Applied Virtual Networks

COMP 4912

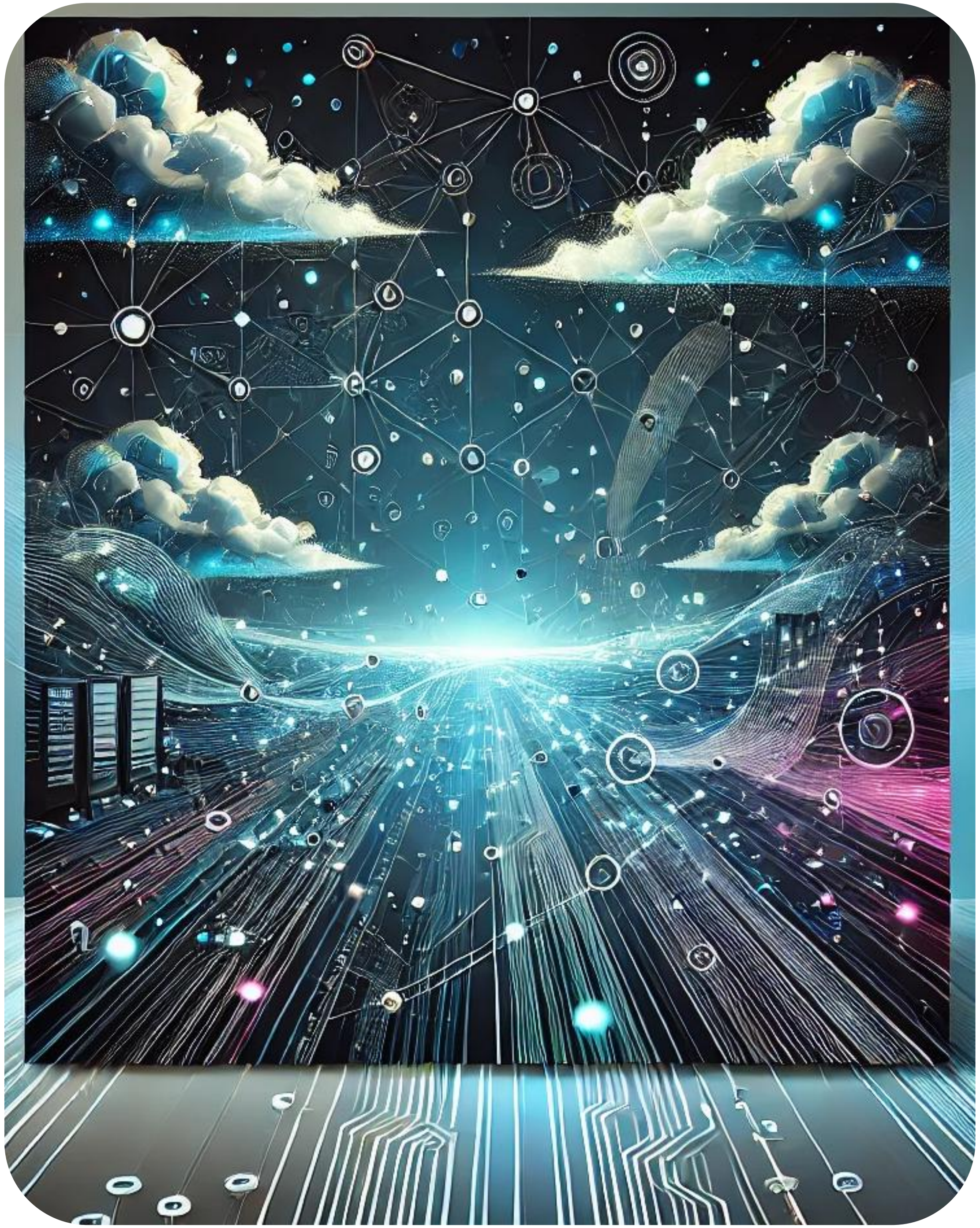
Instructor: **Dawood Sajjadi**

PhD, SMIEEE, CISSP

ssajjaditorshizi@bcit.ca

Winter-Spring 2025

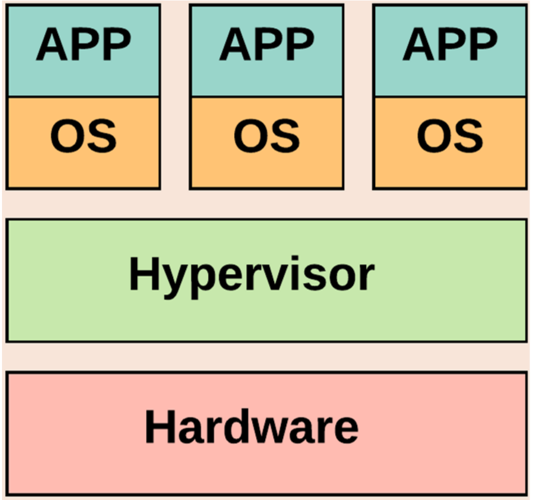
Week #3



Virtual Machines

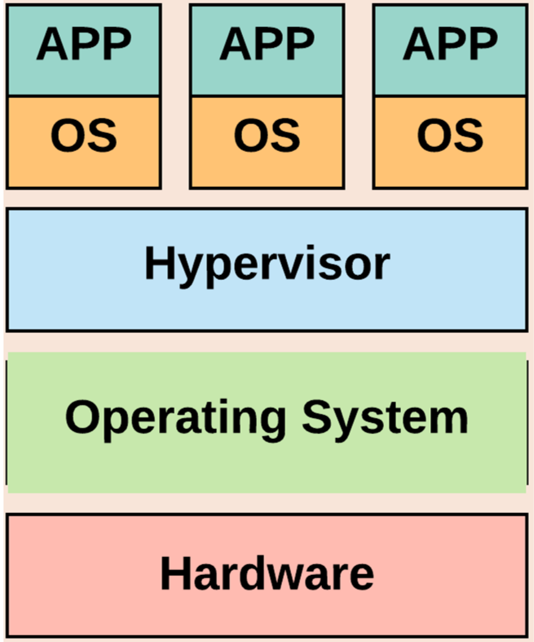
RECAP

Types of Hypervisors



Type-1

Bare-metal

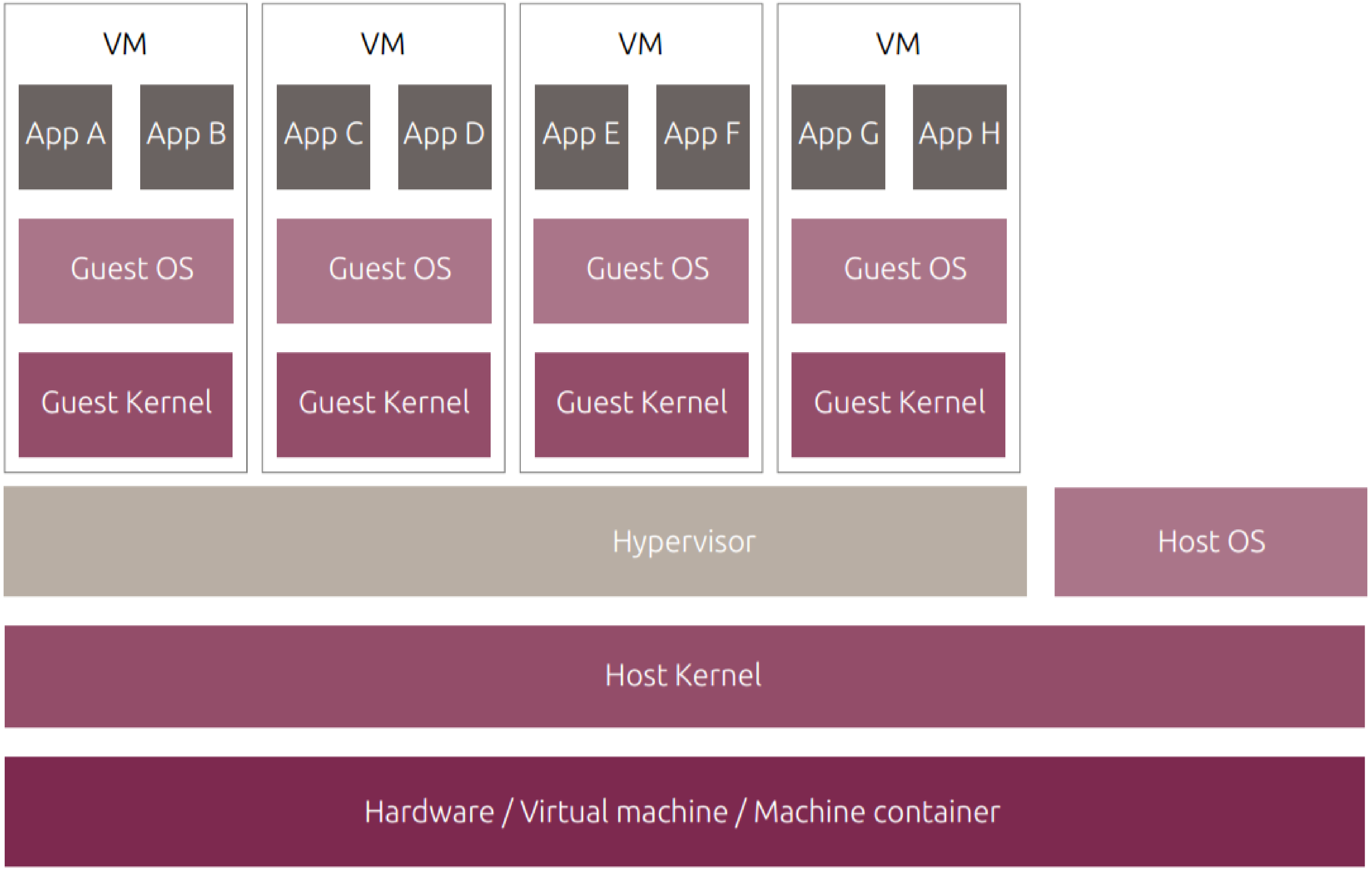


Type-2

Run guest-OS on top of host-OS

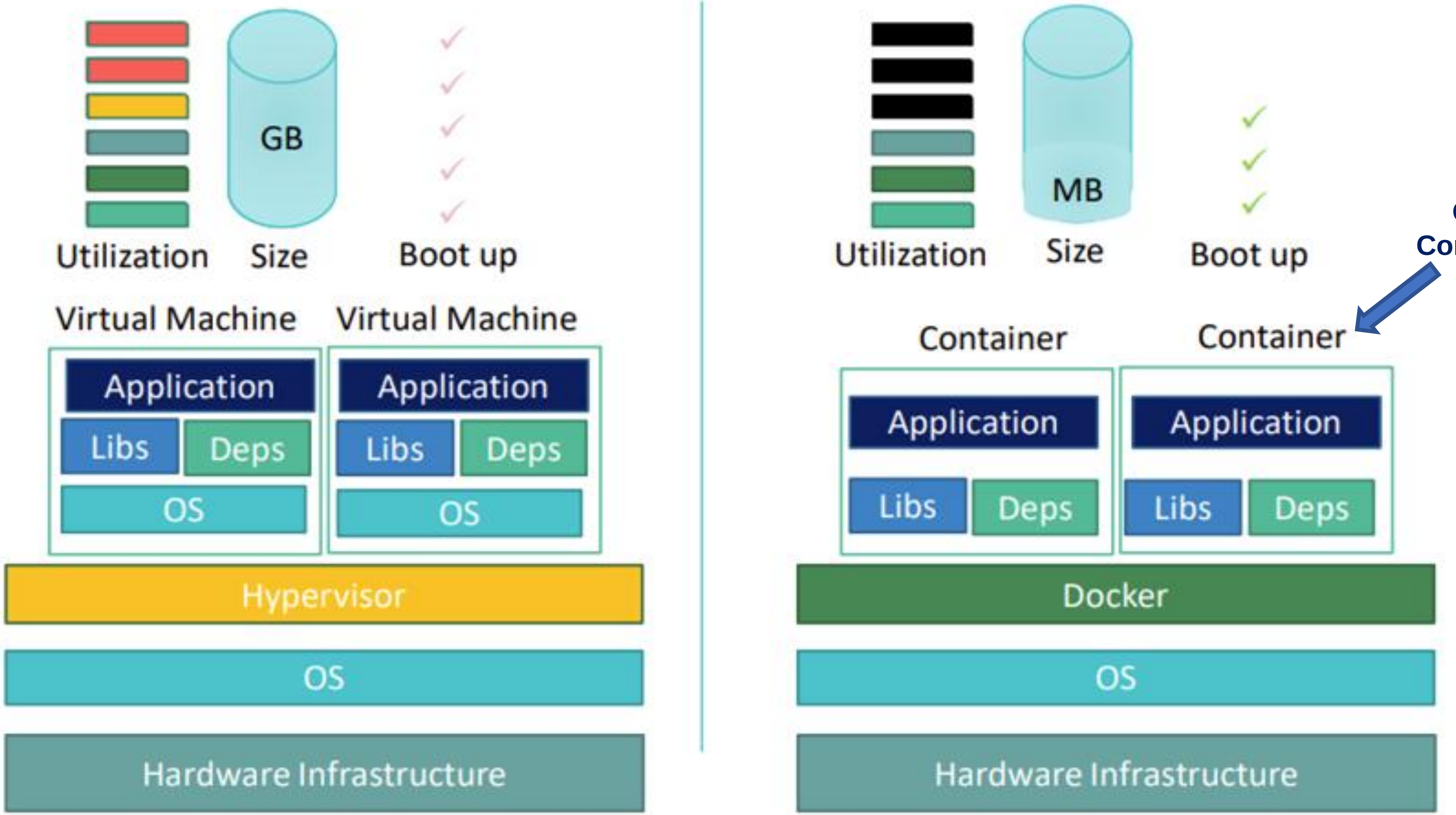
Virtual Machines

RECAP



Containers vs Virtual Machines

RECAP



Created from
Container images

With the same container image, you can reproduce as many containers as you wish.
Think about the image as the recipe, and the container as the cake.
You can make as many cakes as you'd like with a given recipe.

What is Docker and Why use it?

RECAP

docker

from DockerHub

Image file

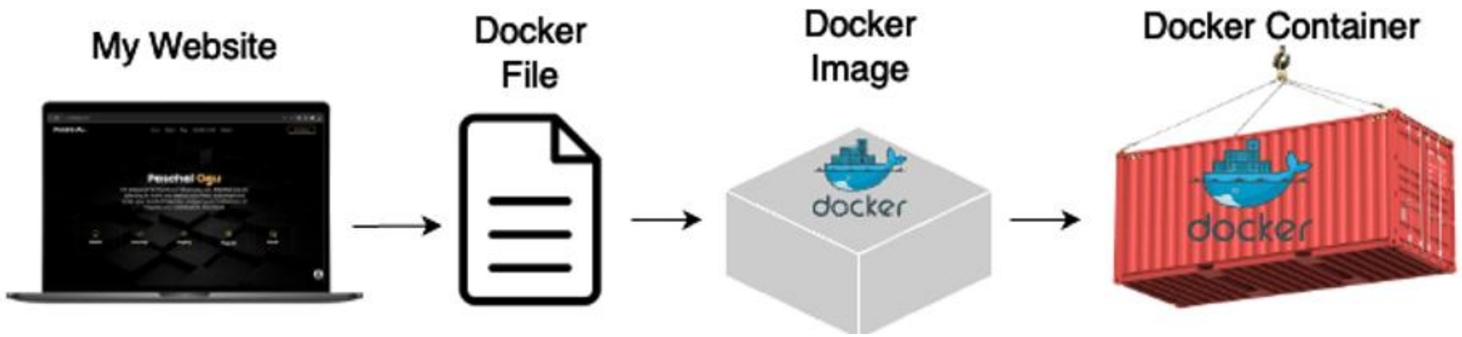
A single file with all the required libraries, package and configuration needed to run a program/app.

Container

Container

Container

Instance of an image that runs a program (process)



Docker makes it very easy to install and run applications and programs without being worrying about <u>software dependencies</u>.

Docker Ecosystem

A platform for running Containers

- Docker Compose
- Docker Server
- Docker Machines
- Docker Client
- Docker Hub
- Docker Images

BCIT™

Docker Commands

RECAP

Running a Simple Nginx Web Server Using Docker

$ docker --version
$ docker search nginx ← Search for an image name
$ docker pull nginx ← Pull an image from Docker Hub
$ docker image ls ← List images pulled to your laptop

$ docker run --name nginx-web-server –d –p 8000:80 nginx ← Run a container from an image
$ docker ps ← List running containers
$ docker ps -s ← Displays approximate size of containers
$ docker logs nginx-web-server ← Check container logs
$ docker exec -it nginx-web-server /bin/sh ← Access a shell inside a sunning container

$ docker stop nginx-web-server ← Stop a running container
$ docker start nginx-web-server ← Start a stopped container

$ docker rm nginx-web-server ← Remove a container
$ docker rmi nginx ← Remove an image

BCIT™

Learning Outcomes of Week #3

1. Understanding YAML language and use it to compose Dockers.

2. Explain How Kernel-based Virtual Machine (KVM) works?

3. Describe the difference between Application and System Containers.

4. Get Familiar with Virsh and its difference with KVM/Qemu.

5. Understanding LXD/LXC to run VMs and Containers.

YAML Intro

```yaml
# What does YAML mean?
YAML:
    - Y: YAML
    - A: Ain't
    - M: Markup
    - L: Language
```

YAML is a human-readable data-serialization format.

Commonly used for configuration files in Cloud Applications.

YAML is designed to be friendly to people working with data and achieves "unique cleanness" by minimizing the use of structural characters, allowing the data to appear in a **natural and meaningful way**.

YAML achieves easy inspection of data's structures by using indentation-based scoping (similar to Python).

XML	JSON	YAML

<Servers>
  <Server>
    <name>Server1</name>
    <owner>John</owner>
    <created>123456</created>
    <status>active</status>
  </Server>
</Servers>
	{
	  Servers: [
	    {
	    name: Server1,
	    owner: John,
	    created: 123456,
	    status: active
	    }
	  ]
	}
		Servers:
		  - name: Server1
		       owner: John
		       created: 123456
		       status: active

YAML Intro

```yaml
# What does YAML mean?
YAML:
    - Y: YAML
    - A: Ain't
    - M: Markup
    - L: Language
```

easy to **read**, **write** and **hierarchical** structure.

- ✓ **Key-Value Pairs**: Data is represented as key-value pairs. (key: value)
- ✓ **Indentation**: Indentation is important! It uses spaces (not tabs).
- ✓ **Lists**: Use dashes (-) for lists.

yaml

```yaml
server:
  host: example.com
  port: 80
  protocol: http
```

- server is a key, and its value is another set of **key-value pairs**.
- **host**, **port**, and **protocol** are nested under server.

BCIT™

YAML Intro

Using Lists in YAML

```yaml
servers:
  - host: server1.example.com
    port: 8080
  - host: server2.example.com
    port: 8081
  - host: server3.example.com
    port: 8082
```

- **Lists** are represented with dashes (-).
- Each item in the list can have multiple key-value pairs.

Nested Data Structures

```yaml
app_config:
  name: myapp
  environments:
    - name: production
      url: https://prod.example.com
      replicas: 3
    - name: staging
      url: https://staging.example.com
      replicas: 2
```

- YAML allows nesting of keys and lists.
- **environments** is a list with dictionaries for each environment

YAML Intro

Scalers and Data Types

- **Strings**: "example" or without quotes.
- **Booleans**: true, false.
- **Numbers**: 100, 3.14.
- **Null**: null or ~.

```yaml
app:
  name: MyService
  enabled: true
  timeout: 30
  retries: 3
```

Example: Service Configuration in YAML

```yaml
service:
  name: database-service
  version: 2.1.0
  replicas: 3
  resources:
    cpu: "1"
    memory: "2Gi"
  health_check:
    path: /healthz
    interval: 30s
    timeout: 5s
```

- Used for configuring the service in a Kubernetes or Docker context.
- Defines replicas, resource limits, and health check intervals

YAML Intro

Mapping

```
map:
  - key1: value1
  - key2: value2
  - key3: value3
```

Block Mapping

```
user:
  username: cinnamon
  name: John
  surname: Doe
  email: cinnamonroll@example.com
  billing-address:
    street: Some Street
    number: 32
    zip-code: 17288
```

Flow String Scalars

```
key: This is a plain flow scalar.
```

```
key: 'This is a single-quoted string with ''single quotes''.'
```

```
key: This is a
flow string scalar
that becomes a
single line.
```

YAML Intro

Using YAML in CI/CD

```yaml
version: '3'
services:
  web:
    image: webapp:latest
    ports:
      - "8080:80"
    environment:
      - NODE_ENV=production
  db:
    image: postgres:13
    environment:
      - POSTGRES_PASSWORD=secret
```

- A docker-compose.yml example for defining services in a CI/CD pipeline.
- Specifies which containers to run and how they communicate

YAML with Kubernetes

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
        - name: myapp-container
          image: myapp:latest
          ports:
            - containerPort: 8080
```

- YAML used to define a Kubernetes Deployment.
- Defines how many replicas, the container image, and port configuration.

YAML Intro

Commenting in YAML

Adding Comments in YAML

- Comments begin with # and continue to the end of the line.
- Can be placed anywhere, but should not appear after a value.

```yaml
# This is a sample service configuration
service:
  name: app-service # The name of the service
  replicas: 3 # Number of instances to run
```

YAML Best Practices

- Use two spaces for indentation (no tabs).
- Always start lists with **-**.
- Be mindful of spaces after **:** in key-value pairs.
- Use quotes for strings with special characters or spaces.
- Keep YAML files small and readable, avoid deep nesting where possible.

YAML Intro

Common mistakes in YAML

- Using tabs instead of spaces.
- Forgetting to indent properly.
- Using incorrect key-value formatting.

```yaml
# Incorrect example:
server:
   host=example.com    # 'key: value' format is needed here
   port 80            # Missing colon
```

A single YAML file can have more than more than one document. Each document can be interpreted as a separate YAML file which means multiple documents can contain the same/duplicate keys. A YAML file with multiple documents would look like below, where each new document is indicated by **---.**

```yaml
---
# document 1
name: John Doe
age: 30

---
# document 2
pets:
  - name: Spot
    breed: Labrador Retriever
  - name: Whiskers
    breed: Siamese

---
# document 3
address:
    street: 123 Main Street
    city: Anytown
    state: CA
    zipcode: 12345
```

Docker Compose and YAML

Docker Compose is a tool for defining and managing multi-container Docker applications.
It uses a YAML file to configure application services.

- ✓ Simplifies multi-container setups
- ✓ Enables service orchestration
- ✓ Allows network and volume configuration
- ✓ Provides portability for application environments

Compose files are written in **YAML** format and follow a standard structure:

- ✓ version: Specifies the Compose file version
- ✓ services: Defines individual container services
- ✓ volumes: Configures persistent storage
- ✓ networks: Specifies custom networking

BCIT™

Docker Compose and YAML

Setup a web server using Nginx and a static HTML website served from a local directory.
This example demonstrates key Docker Compose concepts like defining services, volumes, and ports.

docker-compose.yml

```yaml
version: '3.9'

services:
  web:
    image: nginx:latest
    ports:
      - "8000:80"  # Map port 8000 on the host to port 80 in the container
    volumes:
      - ./html:/usr/share/nginx/html:ro   # Mount the local html/ directory to Nginx's web root
```

Index.html

```html
<!DOCTYPE html>
<html lang="en">
<body>
   <h1>Welcome to My Simple Website!</h1>
   <p>Served using Docker Compose with Nginx.</p>
</body>
</html>
```

$ docker-compose up -d --build → Access your website via http://localhost:8000

Docker Compose and YAML

Setup up a **Load Balancer with multiple backend web servers.**
This demonstrates how Docker Compose can manage service communication, networking, and scalability.

We need to create:
- ✓ a **HAProxy load balancer** that distributes traffic.
- ✓ Two backend **web servers** running Nginx, both serving the same content.

Project Structure

```
load-balancer/
├── docker-compose.yml
├── haproxy/
│   ├── haproxy.cfg
├── web1/
│   └── index.html
├── web2/
│   └── index.html
```

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Web Server 1</title>
</head>
<body>
    <h1>Welcome to Web Server 1</h1>
</body>
</html>
```

web1/index.html

web2/index.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Web Server 2</title>
</head>
<body>
    <h1>Welcome to Web Server 2</h1>
</body>
</html>
```

haproxy.cfg

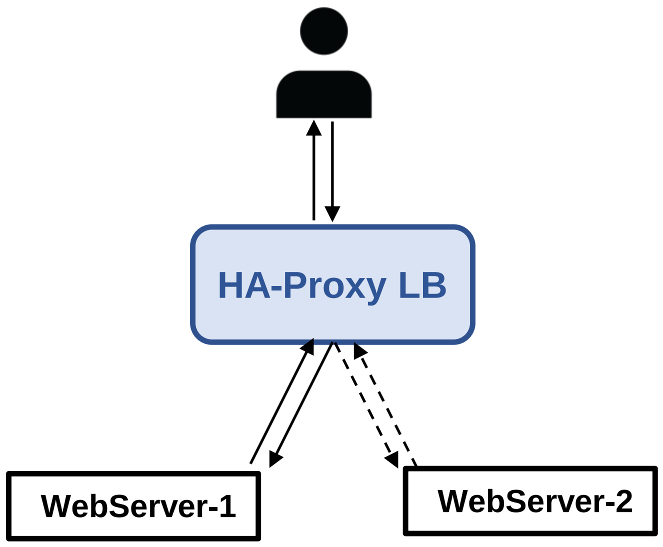
```
frontend http_front
    bind *:80
    default_backend http_back

backend http_back
    balance roundrobin
    server web1 web1:80 check
    server web2 web2:80 check
```

BCIT™

Docker Compose and YAML

```yaml
version: '3.9'

services:
  haproxy:
    image: haproxy:latest
    volumes:
      - ./haproxy/haproxy.cfg:/usr/local/etc/haproxy/haproxy.cfg:ro
    ports:
      - "8000:80"  # Expose HAProxy on port 8000
    depends_on:
      - web1
      - web2
    networks:
      - lb_network

  web1:
    image: nginx:latest
    volumes:
      - ./web1:/usr/share/nginx/html:ro
    networks:
      - lb_network

  web2:
    image: nginx:latest
    volumes:
      - ./web2:/usr/share/nginx/html:ro
    networks:
      - lb_network

networks:
  lb_network:
```

The Compose file orchestrates the load balancer and the two backend web servers.




```
$ docker-compose up -d --build

$ docker-compose down
```

BCIT™

Docker Compose and YAML

Interactive Demo

&

KVM (Kernel-based Virtual Machine)

KVM is a Linux kernel module that turns Linux into a Type-1 hypervisor, enabling full virtualization of hardware.

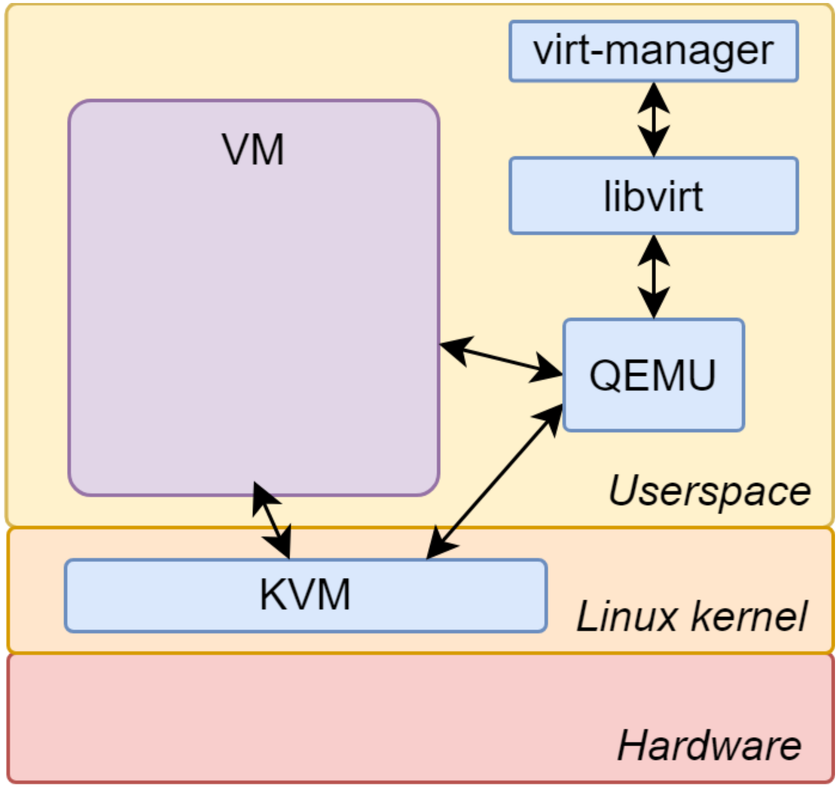
Key Features

- Provides virtualization with **hardware acceleration** (Intel VT-x/AMD-V).
- Can run full virtual machines (VMs) with different OS types (Linux, Windows, etc.).
- Often paired with **QEMU** for device emulation and management.
- Ideal for running resource-heavy applications and legacy OSes in isolated environments.
- Better Performance, Scalability and Security.

QEMU (Quick Emulator) is an open-source emulator and virtualizer that can emulate different architectures (e.g., ARM, x86).

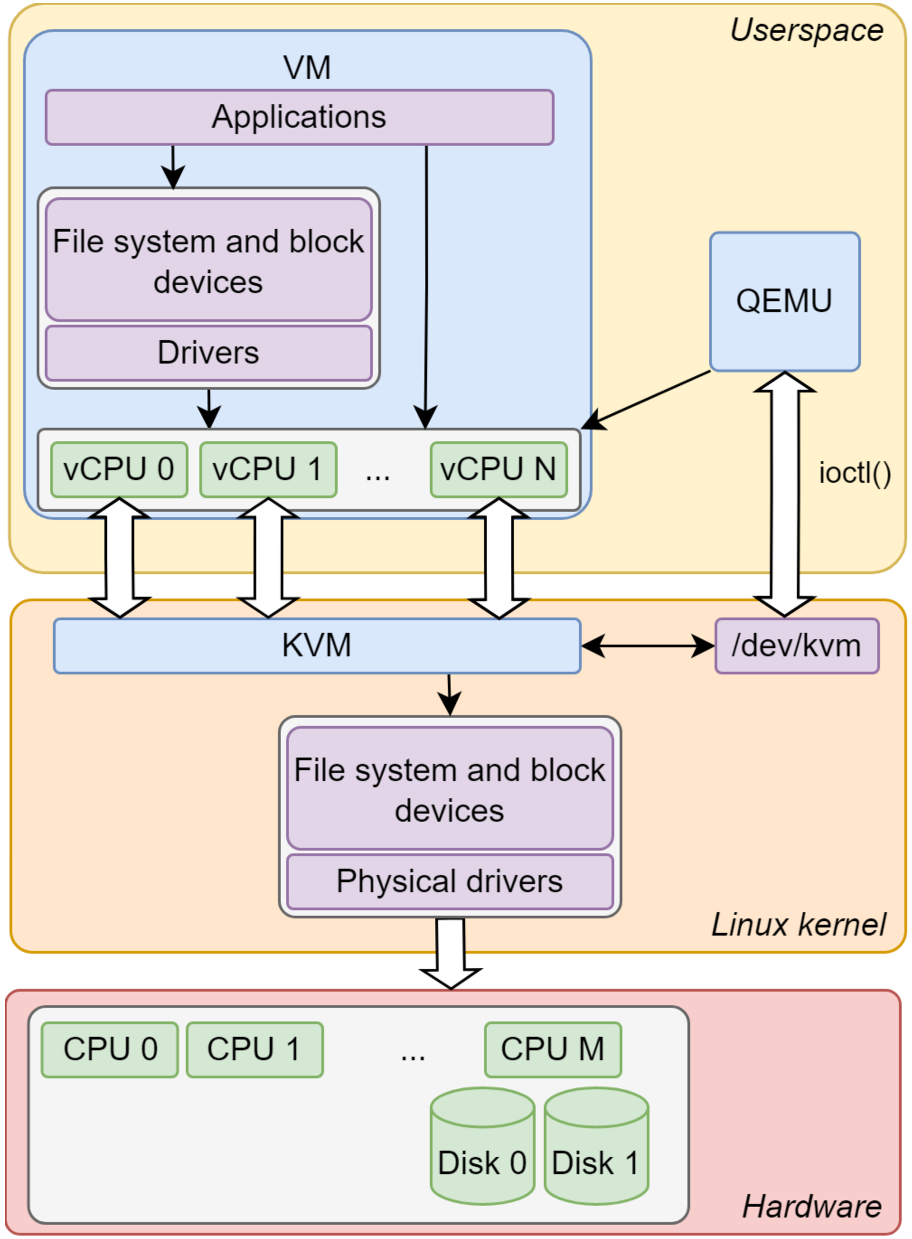
Key Features

- Works alongside KVM to create virtual machines.
- Can emulate a variety of processor architectures.
- Useful for running VMs with non-x86 architectures, such as ARM-based VMs.



KVM (Kernel-based Virtual Machine)

- ✓ **KVM** is the lowest component in the stack and as a Linux kernel module, interacts directly with the hardware.
- ✓ It makes use of hardware-assisted virtualization supported by the CPU, in order to allow **Hardware-based Full Virtualization**.
- ✓ **QEMU** sits above **KVM** and complements its functionality.
- ✓ **QEMU** manages the processes of the guest systems and emulates devices being passed to them.
- ✓ **Libvirt** is the library that is used in order to control **QEMU** and manage the Virtual Machines (VMs).



Linux KVM

$ sudo apt update
$ sudo apt install **cpu-checker**
$ **kvm-ok**

```
my-ubuntu-pc001 ~$ kvm-ok
INFO: /dev/kvm exists
KVM acceleration can be used
```

$ grep –E "(**vmx|svm**)" /proc/cpuinfo

Check if your CPU supports HW Virtualization.

vmx: Indicates **Intel VT-x** hardware virtualization support.
svm: Indicates **AMD SVM** (Secure Virtual Machine) hardware virtualization support.

1. Installing required packages for QEMU/KVM

$ sudo apt install qemu-system-x86 qemu-utils libvirt-daemon-system virt-manager -y
$ sudo systemctl enable --now libvirtd

2. Downloading a Lightweight ISO

$ wget http://tinycorelinux.net/14.x/x86_64/release/TinyCorePure64-current.iso -O tinycore.iso

Linux KVM

Short Demo

Access to Ubuntu running on remote laptop to run Qemu

3. Create a Virtual Disk for VM

$ qemu-img create -f qcow2 tinycore.qcow2 1G

Create a 1 GB virtual disk using qemu-img

4. Run the VM with QEMU/KVM

$ qemu-system-x86_64 -enable-kvm -m 512 -cdrom tinycore.iso -hda tinycore.qcow2 -boot d

- ✓**enable-kvm:** Enables hardware virtualization.
- ✓**m 512:** Allocates 512 MB RAM to the VM.
- ✓**cdrom tinycore.iso:** Boots the ISO file as a live CD.
- ✓**hda tinycore.qcow2:** Uses the virtual disk created earlier.

5. See the list of running VMs

$ ps aux | grep qemu-system

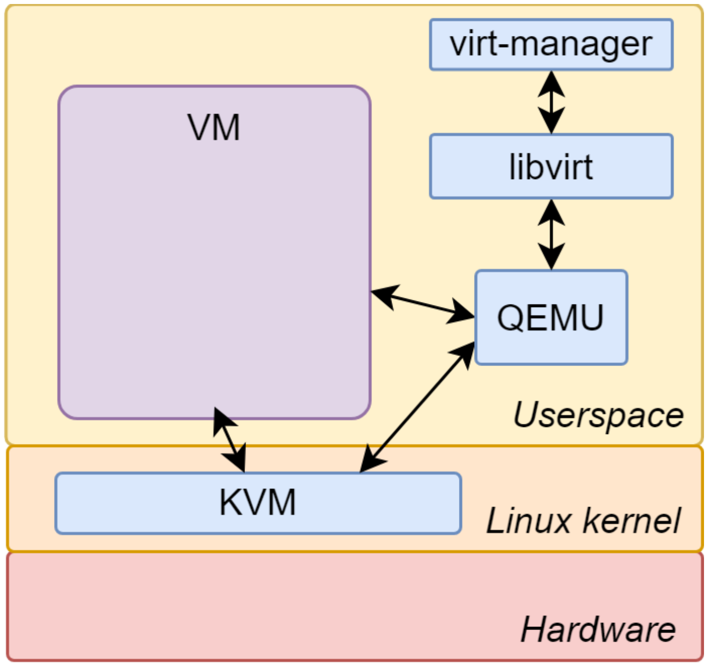
Virsh (Virtual Shell)

✓ **Virsh** is a command-line interface for managing VMs that are handled by the **libvirt** library. It acts as a management layer on top of **KVM**, **QEMU**, or other hypervisors.

✓ **Virsh** communicates with the **libvirt** daemon (**libvirtd**), which interfaces with virtualization back-ends like KVM/QEMU, or LXC. It provides a higher-level, user-friendly way to manage VMs.

KEY FEATURES

- Manage VMs: start, stop, list, pause, destroy.
- Define virtual machines via XML configuration files.
- Connect to remote hypervisors.
- Works with multiple hypervisors (KVM/QEMU, VMware ESXi, etc.).
- Provides more granular control of VMs and their resources.



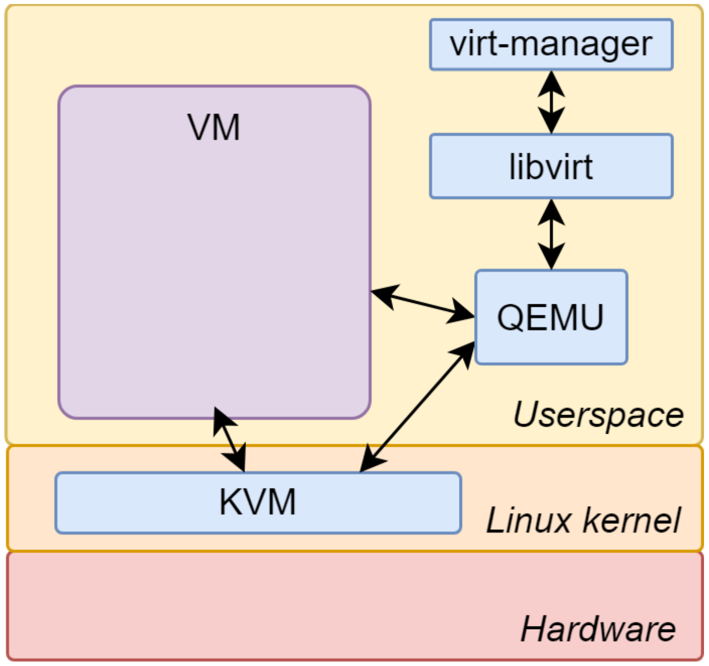
Virsh (Virtual Shell)

Examples

```
$ virsh list --all                      # List all virtual machines
$ sudo virsh define vm.xml              # Define a VM from an XML file
$ sudo virsh undefine vm_name           # Undefine a VM from an XML file
$ sudo virsh start my-vm                # Start the VM named 'my-vm'
$ sudo virsh shutdown my-vm             # Shut down the VM
$ sudo virsh destroy my-vm              # Forcefully shut down the VM
```

How KVM/QEMU/Libvirt/Virsh work together?

- **KVM** provides the virtualization capabilities.
- **QEMU** is often used alongside KVM to create and run VMs.
- **Libvirt** provides a management layer for hypervisors like KVM.
- **Virsh** is a CLI that allows users to interact with libvirt to manage KVM-based VMs.



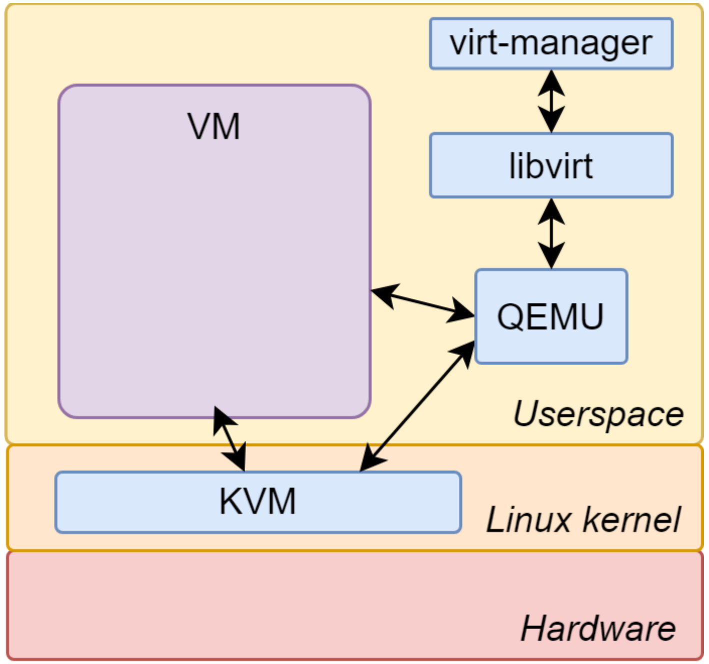
BCIT™

Virsh (Virtual Shell)

vm-config.xml

```xml
<domain type='kvm'>
  <name>my-vm</name>
  <memory unit='MiB'>1024</memory>
  <vcpu>1</vcpu>
  <os>
    <type arch='x86_64'>hvm</type>
  </os>
  <devices>
    <disk type='file' device='disk'>
      <source file='/var/lib/libvirt/images/my-vm.qcow2'/>
      <target dev='vda' bus='virtio'/>
    </disk>
    <interface type='network'>
      <source network='default'/>
    </interface>
    <graphics type='vnc' port='-1' autoport='yes'/>
    <console type='pty'>
      <target type='serial' port='0'/>
    </console>
  </devices>
</domain>
```

https://libvirt.org/formatdomain.html



KVM vs Virsh (Virtual Shell)

	KVM	Virsh
Type	Low-level kernel virtualization module	High-level CLI management tool
Role	Provides HW-accelerated virtualization	Manages VMs via libvirt
Interface	No direct CLI; requires QEMU to interact	Command-line tool for managing VMs
Dependency	Part of the Linux kernel	Depends on the **libvirt** library
Hypervisors	KVM is specific to Linux kernel	Supports multiple Hypervisors (KVM)
VM MGMT	Requires tools like QEMU to manage VMs	Provides commands to define/start/stop VMs
Example	qemu-system-x86_64 -enable-kvm ...	virsh start my-vm

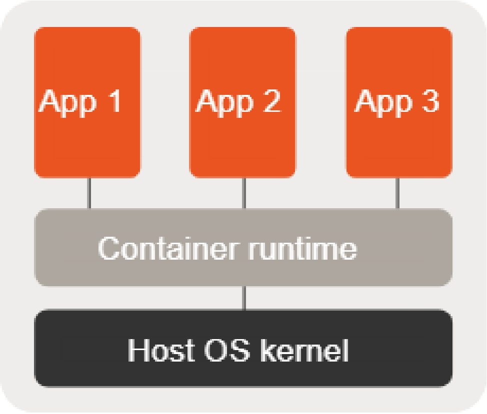
Short Demo (Virsh)

```
$ nano alpine.xml
$ sudo virsh define alpine.xml
$ sudo virsh start alpine-vm
$ sudo virsh shutdown alpine-vm
$ virsh list --all

$ sudo virsh reboot alpine-vm
$ sudo virsh destroy alpine-vm
$ sudo virsh start alpine-vm
$ sudo virsh console alpine-vm

$ sudo virsh net-edit default
$ sudo virsh net-destroy default
$ sudo virsh net-start default
```

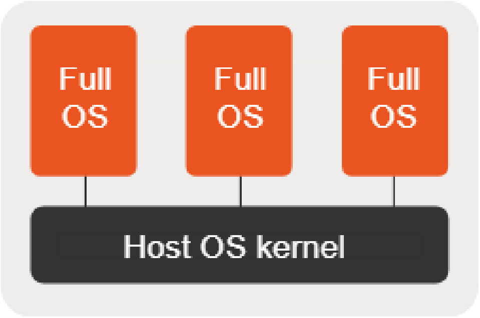
Types of Linux Containers

Application (process) Containers

- The main idea is one application per container, in some cases one process per container
- A container root file system has only necessary Linux distribution-specific binaries and application binaries
- Applications have to be “cloud-native” - different tooling and approaches have to be used
- Mainly non-persistent rootfs, e.g. AUFS or overlayfs, persistence is achieved via other mounted file systems (ephemeral).
- Example: Docker containers



Application containers

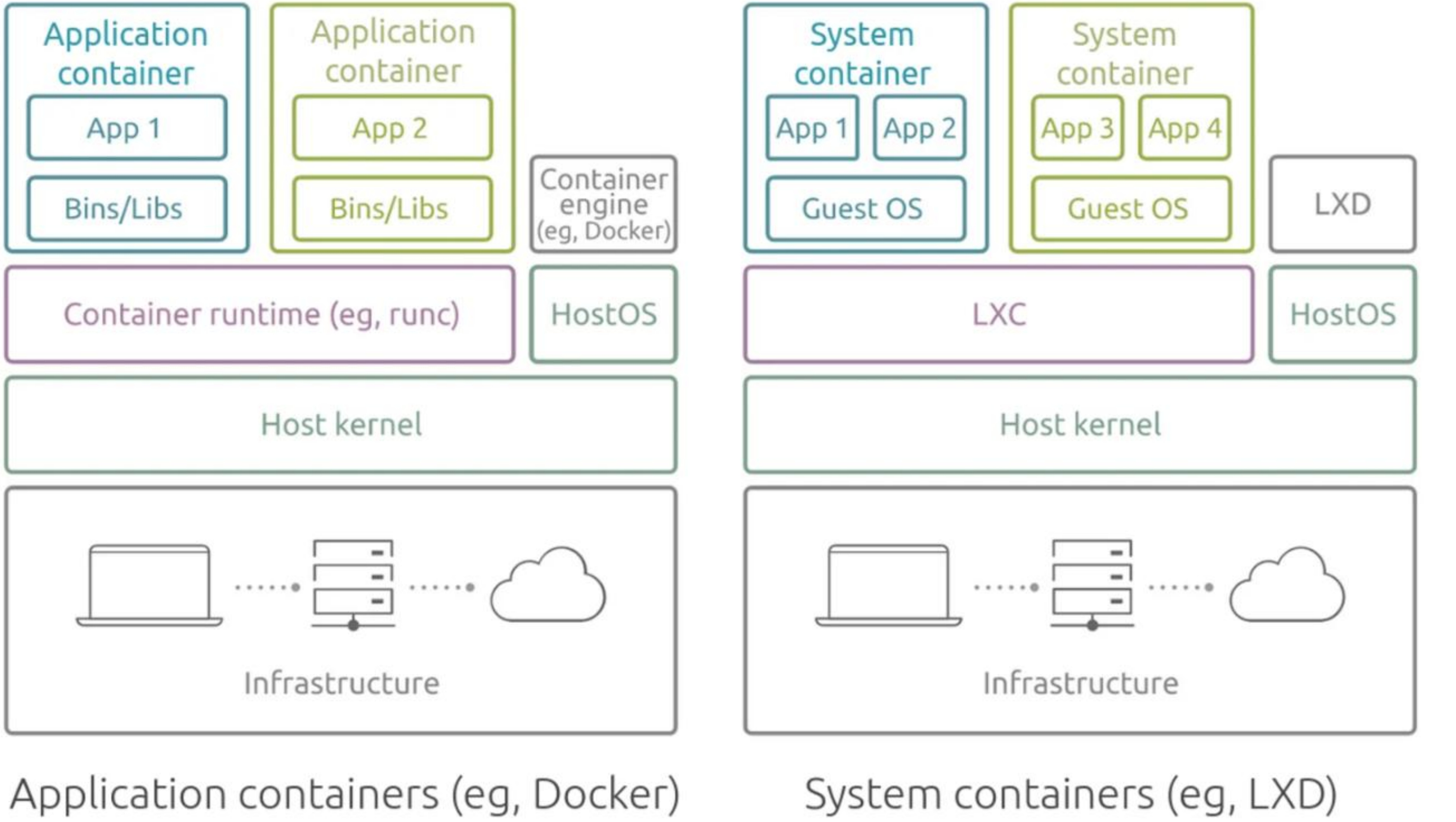


System containers

System (machine) Containers

- Similar to virtual machines - lightweight “logical machines”
- Multiple processes per container
- No need to convert classic user space applications to run them in containers
- Persistent storage (ext4, zfs) for rootfs
- Example: LXC & LXD, OpenVZ.

Application Container vs System Container

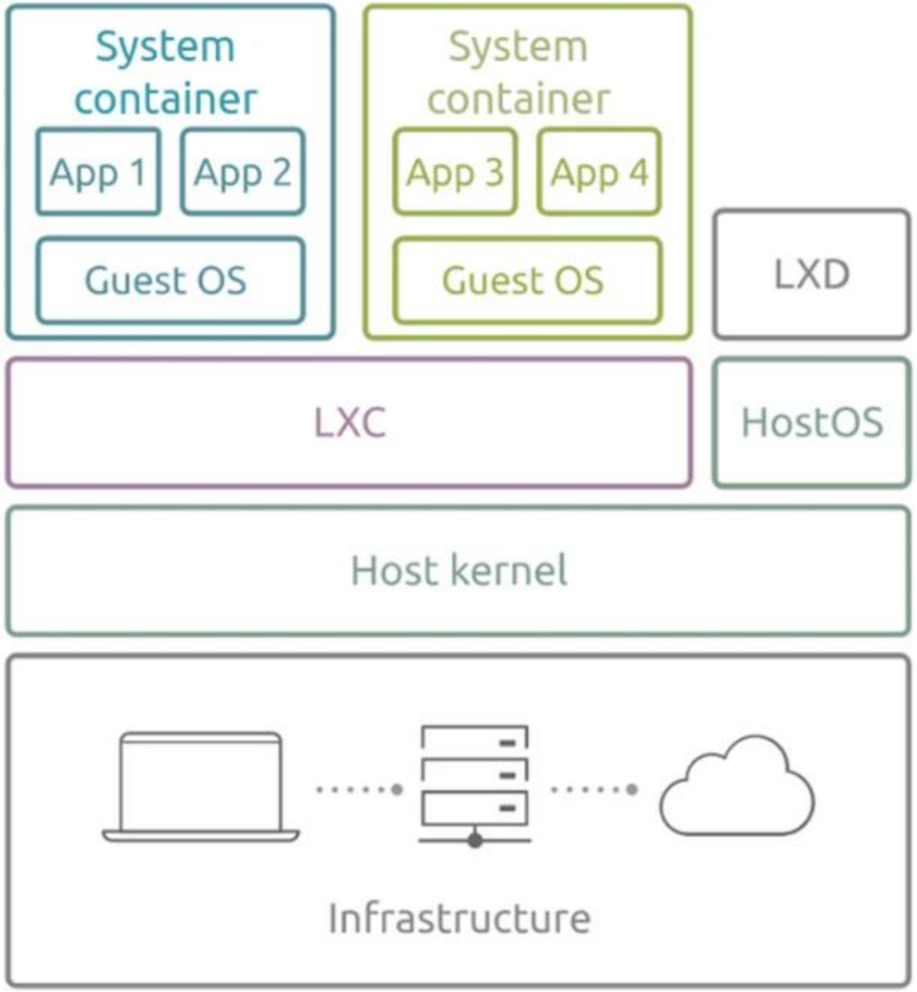


LXC & LXD (LinuX Containers & LinuX Daemon)

LXC: Lightweight Containers

Enables us to run applications or full Linux distributions in isolated environments without the overhead of full virtualization.

- OS-level virtualization.
- Containers share the host kernel but run in isolated user spaces.
- Fast startup times and low resource overhead.
- Ideal for testing or running micro-services.



LXD: A System Container Manager

LXD is a container manager built on top of LXC, designed to provide a simple and easy-to-use interface for managing system containers.

- Provides higher-level management features like live migration/snapshots/cloning.
- Can manage both **containers** and **virtual machines** (using QEMU/KVM).
- Supports **multi-architecture** virtual machines (e.g., x86, ARM).
- REST API available for automation and cloud management.

https://documentation.ubuntu.com/lxd/en/latest/explanation/lxd_lxc/

https://canonical.com/lxd

LXD & LXC (LinuX Daemon & LinuX Containers)

LXC (Linux Containers) use No Hypervisor:

- ✓ LXC is not a virtualization technology in the traditional sense, indeed it is a low-level user space interface that uses Linux kernel features such as namespaces to isolate processes. This means containers share the host kernel, eliminating the need for a hypervisor. Indeed, LXC containers are more lightweight because they directly utilize the Linux kernel without requiring hardware emulation or kernel virtualization.

LXD (Linux Daemon)

- ✓ Similar to LXC, LXD also manages system containers using Linux kernel features (e.g., namespaces, cgroups) and does not rely on a hypervisor for this purpose. LXD is a more intuitive and user-friendly tool aimed at making it easy to work with Linux containers. When LXD is used to run VMs, it relies on QEMU as the underlying hypervisor to provide full hardware virtualization. QEMU works alongside the KVM on Linux systems to enable efficient virtualization with **near-native performance**.

  - For **Containers**, **LXD and LXC** rely on the **Linux kernel**, not a Hypervisor.
  - For **VMs** (via LXD), **LXD** uses **QEMU with KVM as the hypervisor** for HW virtualization.
  - Under the hood, **LXD uses LXC** to create and manage the containers. LXD provides a superset of the features that LXC supports, and it is easier to use.

LXD & LXC (LinuX Daemon & LinuX Containers)

LXD uses LXC for system containers:
LXC is the technology allowing the segmentation of your system into independent containers, whereas LXD is a daemon running on top of LXC which facilitates the creation and management of system containers.

LXD uses QEMU for VMs (need KVM support on Host):
LXD integrates with QEMU for running VMs, allowing it to manage both containers and VMs in a unified environment.

LXC and LXD Roles:
LXC provides the low-level containerization technology, while LXD offers a daemon and a user-friendly interface for managing those containers with additional features.

LXD's features (storage, networking, logging):
LXD supports various storage back-ends, networking configurations, and logging mechanisms, making it highly flexible for a range of use cases.

Image-based and workload flexibility:
LXD leverages image-based deployment, allowing for diverse workloads, from lightweight containerized systems to heavier traditional applications.

While **LXD** shares similarities in managing workloads, it is much more **resource-efficient** (in comparison to VMware or other technologies) and **avoids virtualization overhead**, as it uses system containers for many tasks.

LXD & LXC (LinuX Daemon & LinuX Containers)

LXD daemon

The **lxd** command controls the LXD daemon. Since the daemon is typically started automatically, you hardly ever need to use the lxd command.

LXD client

The **lxc** command is a command-line client for LXD, which you can use to interact with the LXD daemon. You use the lxc command to manage your instances, the server settings, and overall the entities you create in LXD.

```
$ lxd init - -auto
$ lxc image list ubuntu:                      # ubuntu: is officially supported image source
$ lxc image list images:                      # images: is an unsupported source

$ lxc launch ubuntu:22.04 my_container        # using - - vm you can launch a VM but need KVM support on the host.
$ lxc launch images:alpine/edge alpine-container   # install latest available Alpine image
$ lxc delete alpine-container [- -force]      # - -force if it is running
$ lxc copy CONTAINER1 CONTAINER2              # clone

$ lxc list
$ lxc start alpine-container
$ lxc stop alpine-container  [- -force]       # - -force if it doesn't want to stop
$ lxc restart alpine-container  [- -force]
$ lxc pause alpine-container                  # STOP to all container processes
$ lxc exec alpine-container  - -   /bin/sh    # access the container via a BASH shell
$ lxc console alpine-container                # access VM via console
```

BCIT™

https://documentation.ubuntu.com/lxd/en/latest/explanation/lxd_lxc/

https://canonical.com/lxd

Comparison

Feature	VMware	LXC/LXD	KVM/QEMU
Type of Virtualization	Full hardware virtualization	OS-level virtualization (Containers)	Full hardware virtualization (with KVM)
Overhead	High (due to full VM environment)	Low (containers share the host kernel)	Medium (with hardware acceleration)
Performance	Excellent for heavy workloads	Very fast for containerized workloads	High, depending on the configuration
Use Case	Enterprise, cross-platform workloads	Lightweight app deployment, system containers	Full VMs with access to hardware resources
Resource Efficiency	Moderate to high	Very high (containers share kernel)	Moderate (with KVM hardware acceleration)
Ease of Use	GUI-based, user-friendly	CLI-based or LXD GUI, more complex	CLI-based, or using management tools (virt-manager)

BCIT™

LXD & LXC (LinuX Daemon & LinuX Containers)

Interactive Demo

&

2 Breakout Rooms

BCIT™

Lecture #3 Summary

1. Know how to use Docker Compose and YAML language.

2. Explaining the difference between App and Sys Containers.

3. Getting familiar with some of the most popular open source virtualization tools, e.g., KVM/Qemu/Virsh/LXC

4. Utilizing the introduced tools to build VMs and containers, and manage their resources

Review of LAB #2

End of Lecture #3

THANK

Y☺U

- Dawood Sajjadi