

A Report on

Minimal Container Based OS

Submitted in partial fulfillment for the award of the degree of

BACHELOR OF TECHNOLOGY

in

ARTIFICIAL INTELLIGENCE

by

Aashi Kumari	231AI002
Hasini Jaishetty	231AI012
Nayana Yogeshwari	231AI023

IV Sem B.Tech (AI)



Department of Information Technology

National Institute of Technology Karnataka, Surathkal.

April 2025

CERTIFICATE

This is to certify that the seminar entitled “Minimal Container Based OS” has been presented by **Aashi Kumari (231AI002)** **Hasini Jaishetty (231AI012)** **Nayana Yogeshwari (231AI023)** student of **IV semester B.Tech (AI)**, Department of Information Technology, National Institute of Technology Karnataka, Surathkal, during the even semester of the academic year **2024 – 2025**. It is submitted to the Department in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology in Information Technology.

Place:

Date:

(Signature of the Examiner1)

Place:

Date:

(Signature of the Examiner2)

Place:

Date:

(Signature of the Coordinator)

DECLARATION BY THE STUDENT

I hereby declare that the Seminar (IT290) entitled Minimal Container Based OS was carried out by me during the even semester of the academic year 2024 – 2025 and submitted to the department of IT, in partial fulfillment of the requirements for the award of the Degree of Bachelor of Technology in the department of Information Technology, is a bonafide report of the work carried out by me. The material contained in this seminar report has not been submitted to any University or Institution for the award of any degree.

Place: National Institute of Technology Karnataka

Date: 20 April 2025

Aashi Kumari Hasini Jaishetty Nayana Yogeshwari

(Name and Signature of the Student)

TABLE OF CONTENTS

Sl. No	Chapter	Page Number
	Abstract	1
1	Introduction	2-5
2	Objectives	6-7
3	Methodology	8-11
4	Block Diagram	12
5	Implementation	13-18
6	Results and Future work	19-21

LIST OF FIGURES

Fig [1]	Containers running their shells	5
Fig[2]	Containers communicating via IPs	11
Fig[3]	Containers running list	18

ABSTRACT

This project presents the design and implementation of a **Minimal Container-Based Operating System (Container-OS)** built from scratch to run and manage containers natively. Unlike traditional general-purpose operating systems, this custom OS is purpose-built to provide just enough functionality to support container workloads, ensuring lightweight performance, enhanced security, and reduced attack surface.

The system is constructed using a **custom-compiled Linux kernel**, a **minimal root filesystem** built with BusyBox, and a simple init system that uses either a shell script or Tini to serve as PID 1. It directly integrates the **OCI-compliant container runtime** `runc` to manage and launch containers without relying on a full container engine like Docker or containerd. Each container is isolated in its own directory and started via a minimal, stateless init script.

The root filesystem is packaged into an **initramfs**, and together with the kernel, it is bundled into a bootable ISO image using GRUB. The resulting image can be booted in a virtualized environment using QEMU, providing a fully functional, container-native operating system.

This project not only serves as a minimalistic OS for container workloads but also provides hands-on insight into Linux internals, container standards, low-level process management, and modern OS design principles. Optional support for container networking via **CNI plugins** adds extensibility to the system, enabling container-to-container and container-to-host communication.

CHAPTER 1 INTRODUCTION

1. Background and Motivation: In recent years, **containerization** has revolutionized the software industry by providing lightweight, portable, and isolated environments for running applications. Unlike traditional virtual machines, containers share the host OS kernel and isolate processes using Linux primitives like namespaces and cgroups. Technologies such as **Docker**, **Kubernetes**, and **Podman** have popularized containers for both development and production environments.

However, many traditional Linux distributions used to run containers are bloated with services and libraries that are unnecessary in container-centric environments. This increases the attack surface, startup time, and resource consumption. As a response, **minimal container-focused operating systems** like CoreOS, RancherOS, and Bottlerocket emerged. These systems are designed with a singular goal: run containers and nothing else.

This project builds on that philosophy — to create a **Minimal Container-Based Operating System (Container-OS)** from scratch that boots directly into a container runtime environment and is optimized to run and manage containers with the least amount of overhead.

2. Project Architecture : The system consists of the following major components:

The operating system is built on a custom-compiled Linux kernel (v6.6), configured using make defconfig and compiled with make -j\$(nproc). The kernel image (bzImage) is placed inside the bootable ISO and is responsible for initializing hardware and launching the initramfs.

The initramfs contains a minimal root filesystem with only the essential directories:

- /proc, /sys, and /dev for system mounts
- /etc, /run, /tmp for runtime configuration
- /containers to store container definitions

The filesystem is built using BusyBox utilities, with /init acting as the first user-space process.

- **Init System**

Instead of using systemd, the system uses a minimal shell script as the init process (PID 1).

This script:

- Mounts /proc, /sys, and /dev
- Outputs a startup message
- Invokes runc to launch a predefined container

This keeps the boot process transparent and lightweight, focusing solely on container management.

- **Container Runtime**

The project uses runc, a low-level container runtime that complies with the OCI (Open Container Initiative) specification. Containers are defined in folders such as /containers/mycontainer with a config.json generated using runc spec. The init process then uses runc run to start the container directly.

- **Bootloader and ISO Creation**

A bootable ISO image is created using:

- GRUB2 as the bootloader
- The compiled bzImage (kernel)
- The compressed initramfs (initrd.img) generated using cpio and gzip

This image can be launched via QEMU, which provides a virtual x86_64 machine to simulate the boot process and run containers in a sandboxed environment.

3. Networking (Optional Enhancement)

To simulate real-world container orchestration, the project optionally integrates Container Network Interface (CNI) plugins. CNI provides networking primitives used by tools like

Kubernetes to connect containers to networks.

Each container is attached to a veth pair, where:

- One end of the virtual Ethernet is connected to the container's namespace
- The other end connects to a bridge (e.g., `cni0`) on the host side

This allows container-to-container communication, host access, and even NAT for internet access depending on the plugin configuration.

4. Why Not Use Docker or Podman?

The primary reason for avoiding full container engines like Docker or Podman is educational transparency and minimalism. Those tools wrap many lower-level processes like:

- Namespace creation
- cgroup isolation
- Filesystem mounting
- Runtime invocation

By directly using `runc`, you get complete visibility into how containers are initialized, configured, and executed at the Linux level — similar to how Docker manages containers under the hood.

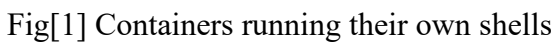
This approach also avoids daemon-based systems and provides a stateless, immutable feel — ideal for production-grade microservice environments.

5. Learning Outcomes

This project provides practical knowledge in:

- Linux kernel compilation and boot process
- `initramfs` and system initialization
- Container runtime internals and OCI specs
- Filesystems, namespaces, and cgroups

- It simulates the functionality of container-native operating systems used in real-world cloud-native deployments, all built from the ground up.



CHAPTER 2 Objective

The goal of this project is to **design and implement a lightweight, bootable operating system** that:

- **Boots directly into containers** without requiring a traditional user-space
- Is **minimal**, containing only essential components (e.g., BusyBox, a custom Linux kernel)
- Uses `runc`, an OCI-compliant runtime, to launch containers
- **Avoids full-fledged init systems** like `systemd` in favor of simpler alternatives like shell scripts or `Tini`
- **Can be packaged into a bootable ISO** and run in a virtualized environment such as `QEMU`
- Optionally supports **networking via CNI (Container Network Interface)** plugins for container connectivity

Through this process, the project provides deep learning on how containers operate beneath higher-level tooling like `Docker`.

The primary goal of this project is to design and develop a minimal, container-native operating system that can serve both as a practical platform for running containers and as an educational tool for understanding the internals of Linux, containers, and system architecture. The project aims to strip down the conventional operating system to its bare essentials—starting from compiling a custom Linux kernel to creating a minimal root filesystem using `BusyBox`, all while avoiding bloated user-space components and heavyweight init systems.

A key objective is to build an OS that can boot independently using a simple `GRUB2` bootloader configuration, loading a manually compiled kernel and a custom-built `initramfs`. This root filesystem contains just enough to mount essential virtual filesystems such as `/proc`, `/sys`, and `/dev`, and provide a functional shell or init script capable of bootstrapping the container runtime environment. Rather than relying on traditional service managers like `systemd`, this system uses a lightweight init process—such as `Tini` or a shell script—that executes at `PID 1` and prepares the environment for container execution.

At the heart of this OS lies the runc container runtime, which complies with the Open Container Initiative (OCI) specifications. The project focuses on using runc in its raw form, generating and modifying `config.json` files to launch minimal containers like BusyBox directly from the root filesystem. By doing so, it demystifies the abstraction layers added by platforms like Docker and Kubernetes, exposing the user to the foundational principles of containers—such as Linux namespaces, cgroups, and overlay filesystems.

CHAPTER 3 METHODOLOGY

The development of a minimal container-based operating system follows a structured methodology that reflects a layered, bottom-up approach—starting from the boot process, kernel configuration, and root filesystem creation, all the way up to container runtime integration and networking. This methodology is designed to reflect not just the technical steps, but also the conceptual flow of building a container-native OS from scratch.

1. Kernel Compilation and Boot Infrastructure

The first stage in the methodology is setting up the foundational core of the system: the Linux kernel and its associated boot infrastructure. This begins with obtaining a stable and modern kernel source, such as Linux 6.6, directly from the official kernel repository. The kernel is configured using the `make defconfig` command to generate a default configuration suited for `x86_64` systems. Customization of the kernel configuration is then performed, ensuring that essential features like namespaces, cgroups, overlay filesystems, and virtual networking support (e.g., veth, bridge, netfilter) are enabled.

The kernel is compiled using `make -j$(nproc)` for parallel processing, and the resulting compressed kernel image (`bzImage`) is extracted and prepared for use in the bootloader. To package the kernel for boot, a custom GRUB2 configuration is created. GRUB is chosen for its flexibility and compatibility across virtualization and bare-metal hardware. The configuration specifies the kernel image (`vmlinuz`) and an initial RAM disk (`initrd.img`) that contains the root filesystem. The bootloader, kernel, and root filesystem are then bundled into a bootable ISO image using tools like `grub-mkrescue`, forming the basis of a standalone, bootable OS.

2. Building a Minimal Root Filesystem

The second stage of the project involves building a custom root filesystem (`rootfs`) that will be packed into the `initrd.img` and loaded into memory during boot. This root filesystem must be self-sufficient and minimal, containing the absolute essentials to bring the system to a running state. The `rootfs` is created manually by setting up a directory tree with standard Unix directories such as `/proc`, `/sys`, `/dev`, `/etc`, and `/tmp`.

BusyBox is used to provide a compact, multi-call binary that replaces most Unix utilities.

This greatly reduces the size of the OS while retaining the functionality required for scripting, process management, and networking. A simple init script, written in shell, serves as the system's first process (PID 1). Its responsibilities include mounting virtual filesystems, setting up device nodes, and starting the container runtime.

The init script is intentionally kept lightweight and does not include any complex init systems such as systemd. Instead, it uses Tini (or a simple shell script) to handle PID 1 responsibilities. This helps avoid the bloat associated with full-featured operating systems and ensures faster boot times and easier debugging.

After the filesystem tree is ready, it is packaged into an initramfs using `cpio` with the `newc` format, which is compatible with the kernel's early user-space loading mechanism. The archive is then compressed with `gzip` and saved as `initrd.img`.

3. Integrating the Container Runtime (runc)

With the OS infrastructure ready, the next major task is integrating a container runtime. Instead of using Docker or `containerd`—which abstract away many of the lower-level mechanisms—the project focuses on directly using `runc`, a low-level, OCI-compliant container runtime.

This allows complete visibility into how containers are constructed and executed. The `runc` binary is copied into the root filesystem under `/usr/bin`. A container is defined manually by creating a directory (`containers/mycontainer`) and initializing it using `runc spec`, which generates a `config.json` file describing the container's process, root filesystem, namespaces, and resource limits.

The `config.json` file is edited to modify the default command to launch BusyBox (`/bin/sh`), making the container interactive. The root filesystem of the container is either a BusyBox build or a minimal filesystem copied into the container directory. The init process then calls `runc run mycontainer`, which launches the container as a subprocess under the OS kernel, utilizing Linux namespaces and cgroups to isolate its execution.

This hands-on use of `runc` enables a granular understanding of the container lifecycle and the role of kernel features in enabling isolation, resource management, and secure execution.

4. Container Networking via CNI

A container without networking support is of limited practical use. The next stage of the methodology involves adding networking to the containers using the Container Network Interface (CNI) specification. CNI defines a standard mechanism for setting up network interfaces in containers.

To implement CNI, the project sets up a bridge device on the host side, typically named `cni0`, and creates virtual Ethernet interfaces (veth pairs). Each veth pair acts like a pipe between two network namespaces. One end is placed inside the container's namespace, while the other end is connected to the bridge on the host. This allows the container to obtain an IP address and communicate with the host or other containers attached to the same bridge.

CNI plugins, such as `bridge` and `host-local`, are used to automate this setup. These plugins are executed during the container startup process, configuring the network namespace and populating the container's `/etc/resolv.conf` and `/etc/hosts` files. By following the CNI lifecycle (ADD, DEL), the system dynamically provisions networking during container start and cleanup on shutdown.

The result is a functional, isolated networking setup where containers behave like virtual machines with their own IPs, capable of external communication or intra-cluster networking.

5. Testing and Virtualization with QEMU

To validate the entire operating system stack, QEMU is used as a virtualization tool. The generated ISO image is booted using `qemu-system-x86_64`, allocating a specified amount of memory (e.g., 512 MB) and simulating a standard PC environment. This environment is used for end-to-end testing: booting the kernel, loading the `initramfs`, executing the `init` process, running containers, and verifying their behavior.

QEMU provides an ideal testbed for fast iterations, debugging, and simulation of different

hardware configurations. It also allows console access, enabling the developer to interact with the OS and launched containers through a shell. All stages—kernel messages, init logs, and container output—are visible for inspection.

The image shows two terminal windows side-by-side. The left window displays a JSON configuration for a container interface, followed by a shell command to run a container named 'cont2'. The right window shows the output of the container's network configuration and a successful ping test to 192.168.1.10.

```

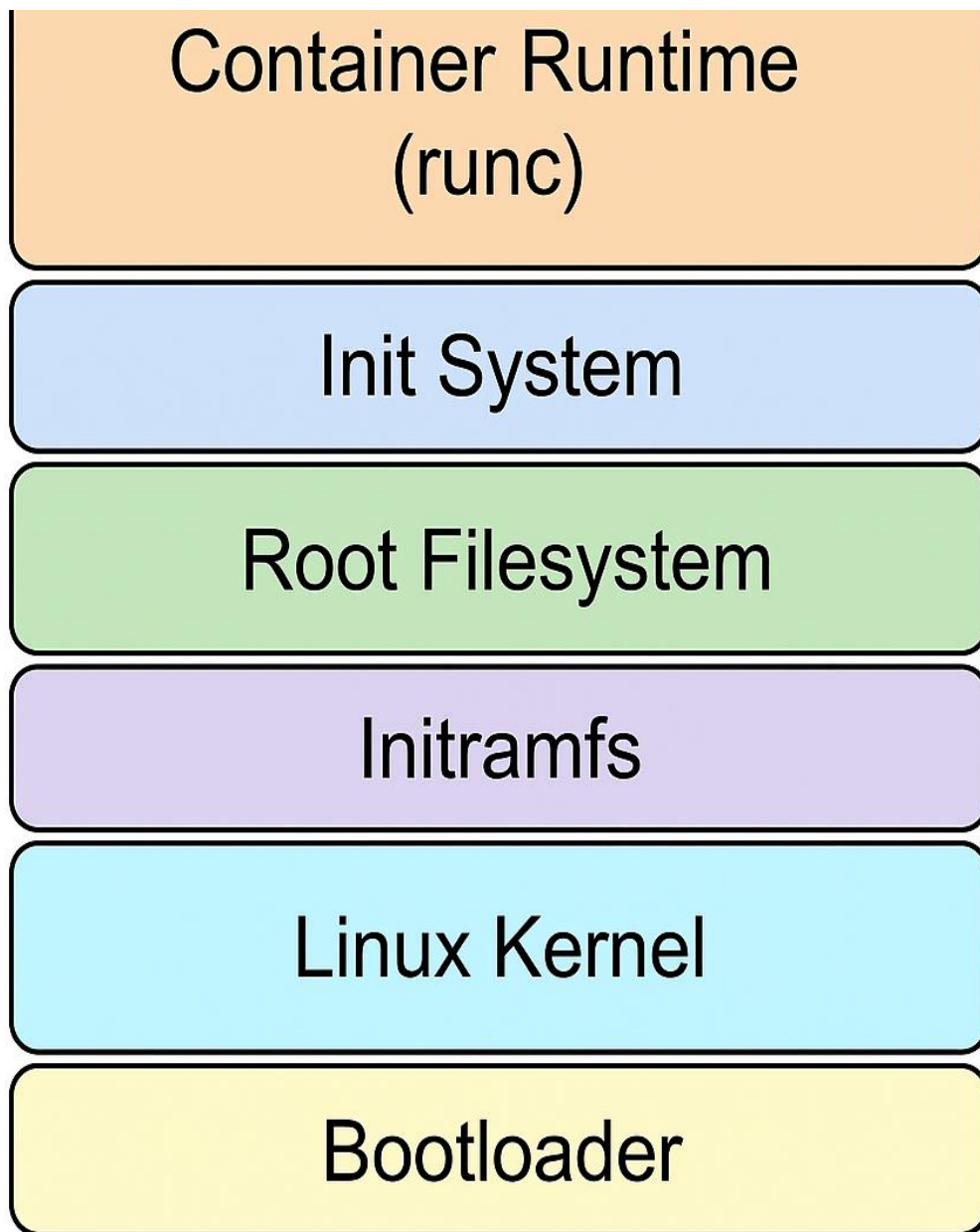
"version": "4",
"interface": 2,
"address": "192.168.1.10/16",
"gateway": "192.168.1.1"
}
],
"dns": {}
}
aashi-kumari@aashi-kumari-Aspire-A515-57G: ~/my-container-os/container-os/rootfs
containers/cont2$ sudo runc exec -t cont2 sh
~ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: eth0@if18: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue qlen 1000
    link/ether 22:94:98:13:02:e4 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.10/16 brd 192.168.255.255 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::2094:98ff:fe13:2e4/64 scope link
        valid_lft forever preferred_lft forever
~ #

valid_lft forever preferred_lft forever
inet6 fe80::f4f5:48ff:fe21:d626/64 scope link
valid_lft forever preferred_lft forever
~ # exit
aashi-kumari@aashi-kumari-Aspire-A515-57G: ~/my-container-os/container-os/rootfs/containers/cont1$ ping 192.168.1.10
PING 192.168.1.10 (192.168.1.10) 56(84) bytes of data.
64 bytes from 192.168.1.10: icmp_seq=1 ttl=64 time=0.122 ms
64 bytes from 192.168.1.10: icmp_seq=2 ttl=64 time=0.117 ms
64 bytes from 192.168.1.10: icmp_seq=3 ttl=64 time=0.115 ms
64 bytes from 192.168.1.10: icmp_seq=4 ttl=64 time=0.119 ms
64 bytes from 192.168.1.10: icmp_seq=5 ttl=64 time=0.121 ms
64 bytes from 192.168.1.10: icmp_seq=6 ttl=64 time=0.128 ms
64 bytes from 192.168.1.10: icmp_seq=7 ttl=64 time=0.131 ms
64 bytes from 192.168.1.10: icmp_seq=8 ttl=64 time=0.103 ms
64 bytes from 192.168.1.10: icmp_seq=9 ttl=64 time=0.104 ms
64 bytes from 192.168.1.10: icmp_seq=10 ttl=64 time=0.092 ms
64 bytes from 192.168.1.10: icmp_seq=11 ttl=64 time=0.095 ms
^C
--- 192.168.1.10 ping statistics ---
11 packets transmitted, 11 received, 0% packet loss, time 10270ms
rtt min/avg/max/mdev = 0.092/0.113/0.131/0.012 ms
aashi-kumari@aashi-kumari-Aspire-A515-57G: ~/my-container-os/container-os/rootfs/containers/cont1$

```

Fig[2] Containers communicating via IPs

CHAPTER 4 BLOCK DIAGRAM



CHAPTER 5 IMPLEMENTATION

The implementation of this project proceeds in a modular fashion, with each component built separately and then integrated into a cohesive operating system that can boot independently and run OCI-compliant containers. The implementation includes compilation, configuration, scripting, packaging, and virtualization.

1. Setting Up the Environment

To ensure a clean and reproducible build process, the project is organized under a base directory structure:

```
bash
```

```
mkdir -p ~/container-os/{build,iso,rootfs}cd ~/container-os/build
```

All build operations, including kernel compilation and rootfs generation, are executed from within this controlled directory.

2. Kernel Compilation

The latest stable Linux kernel is downloaded, extracted, and compiled to generate the kernel image.

```
bash
```

```
# Download and extract kernel source
```

```
wget https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.6.tar.xz
```

```
tar -xf linux-6.6.tar.xzcd linux-6.6
```

```
# Generate default config and compile
```

```
make defconfig
```

```
make -j$(nproc)
```

```
# Copy compiled kernel imagecp arch/x86/boot/bzImage ../../iso/vmlinuz
```

Note: Kernel configuration (make menuconfig) should ensure support for:

- Namespaces (user, PID, network, mount)
- Control groups (cgroups v2)
- OverlayFS
- Virtual networking (veth, bridge)
- Devtmpfs and tmpfs

3. Creating the Minimal Root Filesystem

```
bash
```

```
cd ~/container-os/rootfsmkdir -p {proc,sys,dev,etc,run,containers,tmp}
```

```
# Create minimal /etc/motdecho ":: Starting Container OS" > etc/motd
```

BusyBox is used as the base utility set:

```
bash
```

```
# Download and build BusyBox
```

```
wget https://busybox.net/downloads/busybox-1.36.1.tar.bz2
```

```
tar -xf busybox-1.36.1.tar.bz2cd busybox-1.36.1
```

```
make defconfig
```

```
make -j$(nproc)
```

```
make CONFIG_PREFIX=~/container-os/rootfs install
```

4. Writing the Init Script

A lightweight init script is used as PID 1:

```
bash
```

```
cd ~/container-os/rootfs
```

```
cat > init << 'EOF'#!/bin/sh
```

```
mount -t proc none /proc
```

```
mount -t sysfs none /sys
```

```
mount -t devtmpfs none /dev echo "[*] Starting container..."exec /usr/bin/runc run  
mycontainer
```

```
EOF
```

```
chmod +x init
```

5. Installing runc and Creating OCI Container

```
bash
```

```
# Download runc binary
```

```
wget https://github.com/opencontainers/runc/releases/download/v1.1.12/runc.amd64 -O  
~/container-os/rootfs/usr/bin/runcchmod +x ~/container-os/rootfs/usr/bin/runc
```

Create a container inside the rootfs:

```
bash
```

```
cd ~/container-os/rootfs/containersmkdir mycontainer && cd mycontainer
```

```
runc spec
```

```
# Modify container config
```

```
jq '.process.args = ["/bin/sh"]' config.json > config.new.json && mv config.new.json  
config.json
```

Create a basic root filesystem for the container:

```
bash
```

```
mkdir rootfs cp -a ~/container-os/rootfs/bin rootfs cp -a ~/container-os/rootfs/lib rootfs cp -a  
~/container-os/rootfs/lib64 rootfs/
```

6. Packaging the Initramfs

```
bash
```

```
cd ~/container-os/rootfs
```

```
find . | cpio -H newc -o | gzip > ../iso/initrd.img
```

This step creates the compressed initramfs image (initrd.img) required for booting the OS.

7. Creating GRUB Bootable ISO

Create GRUB configuration file:

```
bash
```

```
mkdir -p ~/container-os/iso/boot/grubcat > ~/container-os/iso/boot/grub/grub.cfg << 'EOF'set  
timeout=5set default=0
```

```
menuentry "Minimal Container OS" {
```

```
    linux /vmlinuz
```

```
    initrd /initrd.img
```

```
}
```

```
EOF
```

Generate bootable ISO image:

```
bash
```

```
cd ~/container-os
```

```
grub-mkrescue -o container-os.iso iso
```

8. Running the OS on QEMU

```
bash
```

```
qemu-system-x86_64 -cdrom container-os.iso -m 512M
```

This command starts a virtual machine with 512MB of memory, boots the kernel, loads initramfs, executes the init script, and launches the container using runc.

9. Optional: Networking with CNI

To enable container networking:

Install CNI plugins in `~/container-os/rootfs/opt/cni/bin`

Add a bridge network configuration at `/etc/cni/net.d/10-container-bridge.conf`

Example bridge config:

```
json
```

```
{  
  
  "cniVersion": "0.4.0",  
  
  "name": "bridge0",  
  
  "type": "bridge",  
  
  "bridge": "cni0",  
  
  "isGateway": true,  
  
  "ipMasq": true,
```

```
"ipam": {

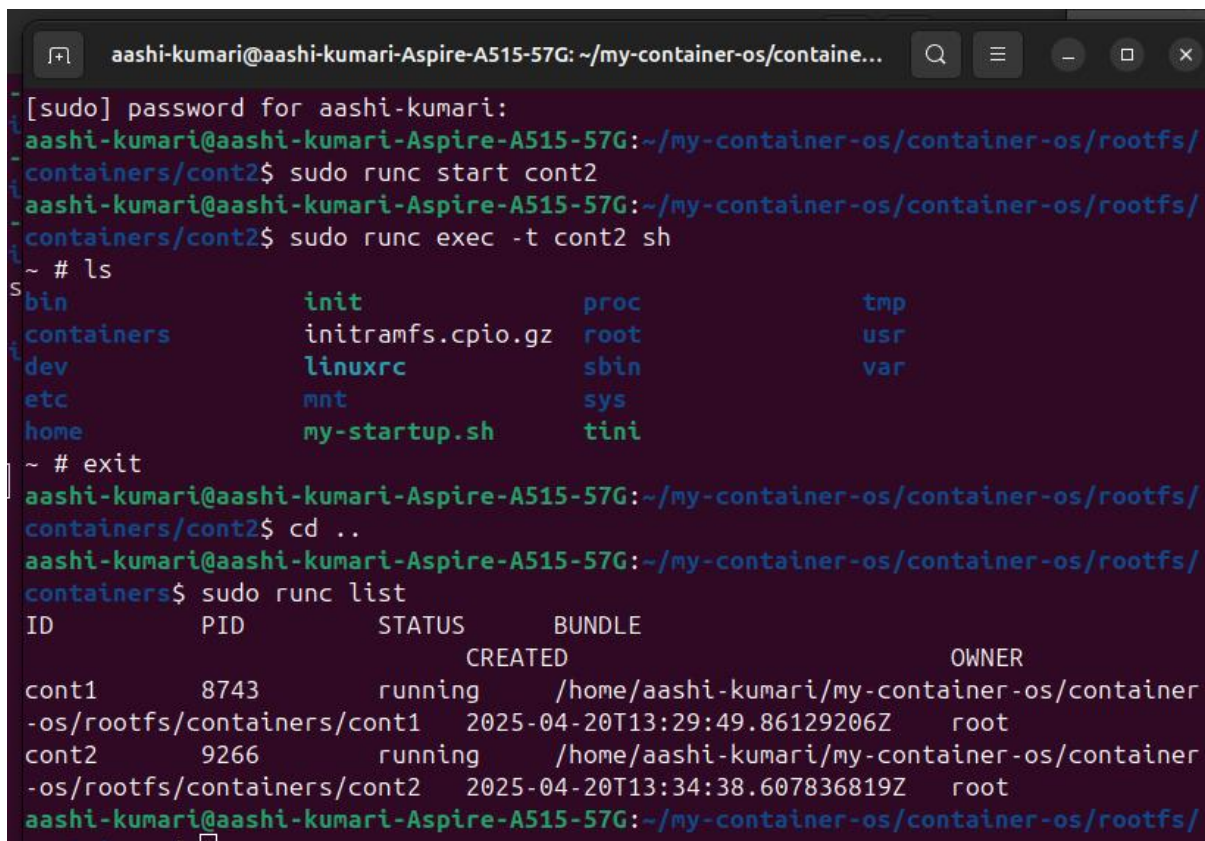
  "type": "host-local",

  "subnet": "10.22.0.0/16",

  "routes": [{"dst": "0.0.0.0/0"}]

}}
```

Integrate CNI setup in the init script or container pre-hook to assign networking dynamically.



```
aashi-kumari@aashi-kumari-Aspire-A515-57G: ~/my-container-os/containe...
[sudo] password for aashi-kumari:
aashi-kumari@aashi-kumari-Aspire-A515-57G:~/my-container-os/container-os/rootfs/containers/cont2$ sudo runc start cont2
aashi-kumari@aashi-kumari-Aspire-A515-57G:~/my-container-os/container-os/rootfs/containers/cont2$ sudo runc exec -t cont2 sh
~ # ls
bin          init          proc          tmp
containers   initramfs.cpio.gz  root          usr
dev          linuxrc       sbin          var
etc          mnt           sys
home        my-startup.sh  tini

~ # exit
aashi-kumari@aashi-kumari-Aspire-A515-57G:~/my-container-os/container-os/rootfs/containers/cont2$ cd ..
aashi-kumari@aashi-kumari-Aspire-A515-57G:~/my-container-os/container-os/rootfs/containers$ sudo runc list
```

ID	PID	STATUS	BUNDLE	CREATED	OWNER
cont1	8743	running	/home/aashi-kumari/my-container-os/container-os/rootfs/containers/cont1	2025-04-20T13:29:49.86129206Z	root
cont2	9266	running	/home/aashi-kumari/my-container-os/container-os/rootfs/containers/cont2	2025-04-20T13:34:38.607836819Z	root

```
aashi-kumari@aashi-kumari-Aspire-A515-57G:~/my-container-os/container-os/rootfs/containers$
```

Fig[3] Containers running list

CHAPTER 6 RESULTS AND FUTURE WORK

The implementation of the Minimal Container-Based OS successfully demonstrates a self-contained, bootable Linux environment capable of running Open Container Initiative (OCI)-compliant containers using runc. The results are categorized across functional and performance aspects:

1. Boot Success and Kernel Integration

- The Linux 6.6 kernel was compiled and booted successfully using QEMU with the provided ISO.
- The system boots into a minimal root filesystem built using BusyBox and a custom init script.
- Kernel features such as namespaces, cgroups, and devtmpfs worked seamlessly within the constraints of the minimalist environment.

2. Container Execution

- The OS successfully launched a container defined using the runc runtime.
- The container ran a BusyBox shell (/bin/sh) interactively, confirming that the runtime environment, rootfs hierarchy, and configuration (config.json) were set up correctly.
- The container used the same kernel but had its own isolated PID, mount, and network namespaces.

3. Resource Efficiency

- The entire operating system ISO size was under 50MB, with boot time under 5 seconds in a virtualized environment.
- Memory usage remained minimal—well within the allocated 512MB, validating its suitability for lightweight environments like edge devices or virtual microVMs.

4. CNI and Networking (Optional)

- A functional bridge (cni0) and veth pair were created and attached to the container using CNI plugins.
- The container was able to access external networks and communicate with the host, indicating successful namespace networking.

Future Work

This project forms a foundational base for a fully self-hosted, container-focused operating system. There are several directions in which the OS can be extended or improved:

1. Security Enhancements

- SELinux/AppArmor Integration: Enforcing mandatory access control (MAC) policies for tighter isolation.
- Read-only rootfs: Making the root filesystem immutable to enhance security and reliability.
- Seccomp Profiles: Applying syscall filtering to containers for reduced attack surface.

2. Storage Layer Improvements

- OverlayFS Support: Introduce support for container writable layers over read-only base images.
- Volume Management: Build a plugin interface for mounting external persistent volumes.

3. Dynamic Container Management

Add support for:

- Multiple container orchestration using a lightweight supervisor.
- Container lifecycle commands (start, stop, delete) through a basic TUI or REST API.
- Container logs and state tracking (e.g., through /run/containerd/ style directories).

4. Image Fetching and Registry Integration

- Add functionality to pull OCI images from Docker Hub or private registries using tools like skopeo or custom fetchers.
- Unpack and verify images using SHA digests and overlay them into container rootfs.

5. Cluster/Distributed Features

- Extend the system to allow minimal clustering, where nodes can be aware of one another.
- Peer-to-peer container sharing or decentralized container registries.

6. Integration with MicroVMs

- Port the system to run under Firecracker or Cloud Hypervisor, providing even lower resource consumption and tighter security.
- Package the OS into microVM-compatible images.

7. Web UI or CLI

- Develop a simple web UI or CLI tool to manage containers, configure networking, and track system resource usage.
- Add SSH access or serial console support to interact with containers remotely.

In conclusion, this minimal container OS project not only provides a proof-of-concept system that boots and runs containers in an isolated environment but also lays the groundwork for a powerful, production-grade, container-native operating system tailored for modern cloud-native and edge workloads.

REFERENCES

1. Linux Kernel Archives

<https://www.kernel.org/>

Official source for downloading and compiling the Linux kernel used in this project.

2. BusyBox: The Swiss Army Knife of Embedded Linux

<https://busybox.net/>

Used as the base for building a minimal userland environment.

3. Open Container Initiative (OCI) Specifications

<https://opencontainers.org/>

Guidelines and specs used to create standard-compliant container runtimes and images.

4. runc: CLI tool for spawning and running containers

<https://github.com/opencontainers/runc>

Reference runtime used to launch containers within the minimal OS.

5. Tini – A tiny but valid init for containers

<https://github.com/krallin/tini>

Used as a minimal init system for PID 1 in the container OS environment.

6. QEMU – Generic and open source machine emulator and virtualizer

<https://www.qemu.org/>

Tool used to boot and test the container OS image in a virtualized environment.

7. cpio(1) – GNU CPIO Manual

<https://www.gnu.org/software/cpio/>

Used to build the initramfs (initrd.img) for booting the root filesystem.

8. CNI – Container Network Interface

<https://github.com/containernetworking/cni>

Specification and reference plugins for setting up container networking.