

Container Runtime: Design Document

Overview

This system implements a minimal container runtime that provides process isolation using Linux namespaces, resource control through cgroups, and layered filesystem support via overlayfs. The key architectural challenge is orchestrating multiple kernel isolation mechanisms while maintaining clean abstractions and handling complex failure modes across namespace boundaries.

This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

Context and Problem Statement

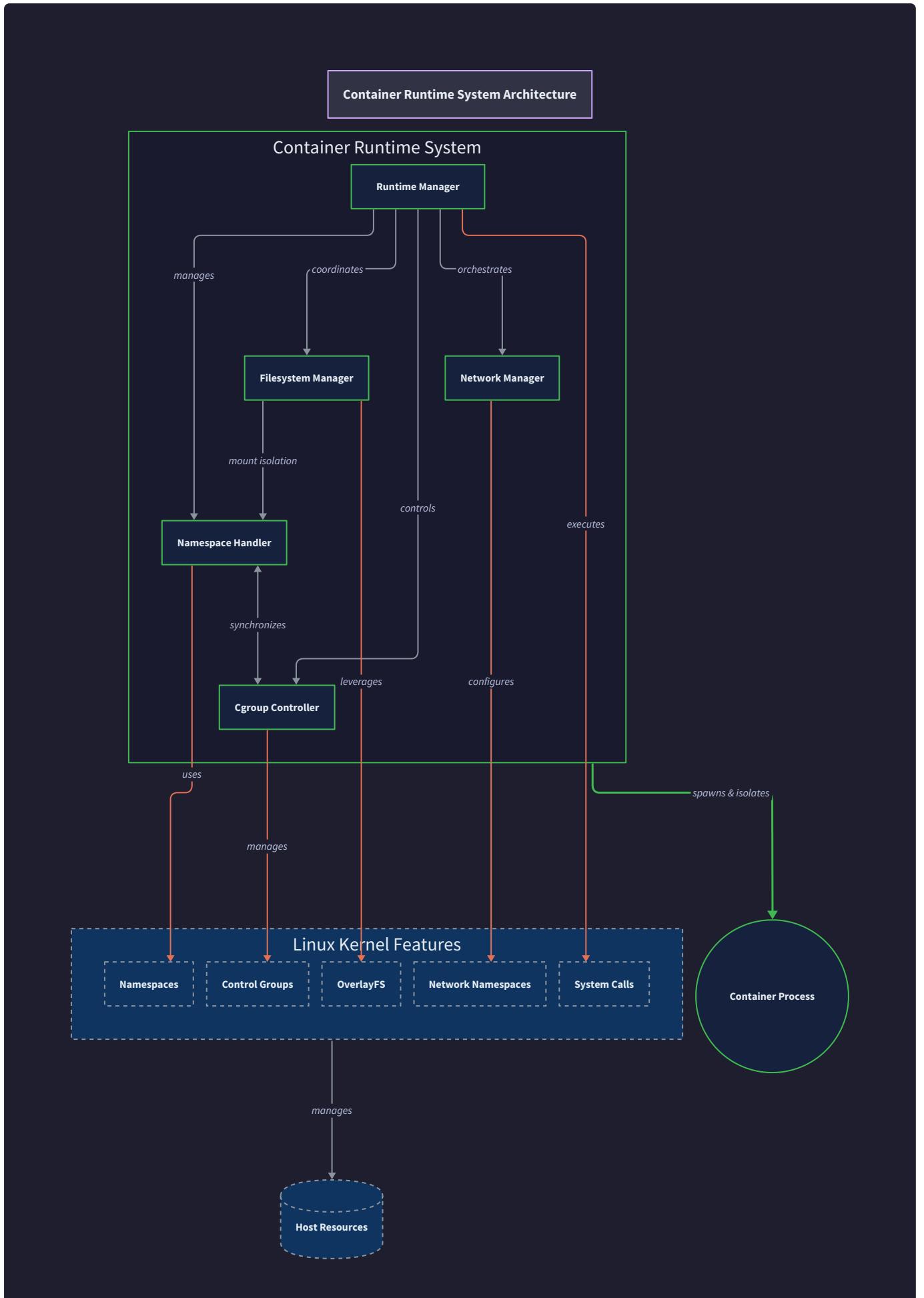
Milestone(s): This section provides foundational understanding for all milestones, establishing the conceptual framework for process isolation, resource control, and filesystem layering.

Modern software deployment relies heavily on containerization, yet many developers treat containers as black boxes—they understand how to use Docker commands but lack insight into the fundamental isolation mechanisms that make containers possible. Building a container runtime from scratch reveals the elegant interplay between Linux kernel features: namespaces for process isolation, cgroups for resource control, and overlay filesystems for efficient image layering. This educational journey transforms abstract concepts like "process isolation" into concrete understanding of system calls, kernel interfaces, and the intricate choreography required to create truly isolated execution environments.

The core challenge in container runtime design lies not in any single isolation mechanism, but in orchestrating multiple kernel subsystems to work together harmoniously. A container must simultaneously exist in its own process tree (PID namespace), have its own filesystem view (mount namespace), control its resource consumption (cgroups), and maintain network connectivity (network namespace)—all while appearing to be a normal process from the host's perspective. This coordination requires deep understanding of Linux kernel interfaces, careful error handling across subsystem boundaries, and robust cleanup procedures when things go wrong.

Mental Model: Apartments in a Building

Before diving into technical details, consider this mental model: **containers are like apartments in a large apartment building, where each apartment provides complete living isolation while sharing the building's infrastructure.**



In this apartment building analogy:

The Building (Host System) represents the physical machine running Linux. The building provides shared infrastructure: electricity (CPU), water and heating (memory and I/O), structural support (kernel), and external connectivity (network). All apartments rely on this shared foundation, but each apartment gets its own allocated portion of these resources.

Individual Apartments (Containers) are completely isolated living spaces. From inside apartment 3B, you cannot see into apartment 2A's living room, rummage through their belongings, or use up all the hot water and leave them with cold showers. Each apartment has its own address (IP address), its own thermostat controls (resource limits), and its own locked door (security boundaries). The residents of apartment 3B might not even know that apartment 2A exists—they see their apartment as if it were a standalone house.

The Building Management (Runtime Manager) handles apartment lifecycle: creating new apartments when tenants move in, setting up utilities connections, enforcing building rules about noise and resource usage, handling maintenance requests, and cleaning up when tenants move out. Management also handles the complex coordination between different building systems—ensuring the electrical system, plumbing, HVAC, and security all work together for each apartment.

Utility Connections (Namespaces) provide each apartment with isolated access to building services. Each apartment gets its own electrical meter (PID namespace), its own water pressure (mount namespace), its own mail delivery (network namespace), and its own thermostat zone (UTS namespace). These systems are carefully isolated—a power surge in apartment 2A doesn't affect apartment 3B's electricity, and a plumbing leak in 3B doesn't contaminate 2A's water supply.

Resource Allocation (Cgroups) ensures fair sharing of building resources. Management enforces rules: apartment 2A gets 30% of the building's heating capacity, apartment 3B gets 40% of the available hot water, and no single apartment can use more than 25% of the total electrical capacity. When apartment 2A tries to exceed its heating allowance, the system automatically regulates their usage rather than letting them freeze out their neighbors.

Shared Infrastructure Layers (Overlay Filesystem) allow apartments to share common elements efficiently while maintaining isolation. All apartments share the same basic floor plan and fixtures (base image layers), but each apartment can customize their space with their own furniture and decorations (writable layer). When a tenant in 3B paints their walls red, it doesn't affect the walls in 2A—each apartment's customizations exist in their own layer.

The key insight is that **isolation is not about physical separation—it's about controlled sharing**.

Apartments share the same building, electrical grid, and water supply, but each apartment gets its own controlled interface to these shared resources.

This mental model illuminates the fundamental challenge of container runtime design: orchestrating multiple isolation mechanisms (utility connections, resource allocation, space management) to create the illusion of

independent environments while efficiently sharing underlying infrastructure. Just as building management must coordinate electricians, plumbers, HVAC technicians, and security to make apartments work, a container runtime must coordinate namespace creation, cgroup configuration, filesystem mounting, and network setup to make containers work.

Existing Container Runtime Approaches

Understanding the design space of container runtimes requires examining how different implementations approach the fundamental challenges of process isolation, resource control, and filesystem management. Each runtime makes different trade-offs between simplicity, performance, security, and feature richness, providing valuable insights into architectural decisions we'll face in our own implementation.

Decision: Runtime Architecture Comparison

- **Context:** Container runtimes exist at different abstraction levels, from low-level kernel interface wrappers to high-level developer platforms. Understanding this landscape helps us position our educational runtime and learn from proven approaches.
- **Options Considered:** Build using existing runtime as foundation, implement OCI-compatible runtime, create minimal educational runtime
- **Decision:** Create minimal educational runtime that implements core concepts directly
- **Rationale:** Direct implementation of kernel interfaces maximizes learning value by exposing all underlying mechanisms without abstraction layers hiding the complexity
- **Consequences:** Higher implementation effort but deeper understanding of fundamental isolation primitives and their interactions

Runtime	Architecture Level	Primary Focus	Implementation Language	Key Design Philosophy
Docker	High-level Platform	Developer Experience	Go	Integrated toolchain with registry, build system, and runtime
containerd	Mid-level Daemon	Industry Standard	Go	Modular architecture with pluggable components
runc	Low-level Runtime	OCI Compliance	Go	Minimal implementation of OCI runtime specification
crun	Low-level Runtime	Performance	C	Fast startup times with lower memory overhead
kata-containers	High-security Runtime	Hardware Isolation	Rust/Go	VM-based containers for enhanced security boundaries

Docker: Integrated Developer Platform

Docker pioneered the modern container experience by providing a complete developer toolchain that abstracts away kernel complexity. The Docker architecture consists of multiple layers: the Docker CLI communicates with the Docker daemon, which manages container lifecycles by delegating to containerd, which in turn uses runc for actual container creation. This layered approach prioritizes developer experience—running `docker run nginx` hides the complexity of namespace creation, cgroup configuration, and overlay filesystem mounting behind a simple command.

The Docker approach teaches us that **developer experience often requires architectural complexity**. Docker's multi-daemon architecture (`dockerd` → `containerd` → `runc`) introduces latency and resource overhead but provides features like image building, registry integration, volume management, and networking that developers expect. For our educational runtime, we'll implement these kernel interfaces directly rather than layering abstractions, maximizing learning value even at the cost of developer convenience.

containerd: Modular Industry Standard

containerd represents the industry's attempt to create a standard, modular container runtime that can serve as a foundation for higher-level tools. Originally extracted from Docker, containerd focuses on the core container lifecycle: image management, container execution, storage management, and network attachment. It provides a gRPC API that tools like Docker and Kubernetes use for container operations.

containerd's architecture demonstrates **separation of concerns through well-defined interfaces**. The containerd daemon manages container metadata and coordinates between specialized subsystems: the image store handles layer management, the snapshotter manages filesystem layers, the runtime manager handles container execution, and the network plugins handle connectivity. Each subsystem can be replaced or extended without affecting others.

Our educational runtime will adopt containerd's principle of component separation while implementing everything in a single process to avoid the complexity of inter-process communication during the learning phase.

runc: OCI-Compliant Minimal Runtime

runc implements the Open Container Initiative (OCI) runtime specification, providing the lowest-level container execution without daemon overhead. It takes a JSON configuration file describing desired container properties (namespaces, cgroups, filesystem mounts, process arguments) and executes the specified process in an isolated environment. runc demonstrates that container creation can be reduced to a deterministic sequence of system calls.

The runc approach reveals that **containers are just processes with fancy isolation**. When you execute `runc run container-id`, runc performs this sequence: creates namespaces via `clone()` system call, sets up cgroup limits, mounts the overlay filesystem, configures networking interfaces, and finally `execv()` the target process. The target process becomes PID 1 in its own PID namespace, completely unaware that it's running in a container.

Our implementation will follow runc's principle of deterministic container creation while exposing each step explicitly for educational purposes rather than hiding them behind a configuration file.

crun: Performance-Optimized Runtime

crun reimplements the OCI runtime specification in C rather than Go, achieving significantly faster container startup times and lower memory overhead. This design choice illustrates the performance trade-offs inherent in container runtime implementation. Go's garbage collector and higher-level abstractions provide development productivity but introduce latency that becomes significant when orchestrators like Kubernetes start thousands of containers per minute.

crun's performance focus teaches us that **system call overhead dominates container creation performance**. The fundamental operations—namespace creation, cgroup setup, filesystem mounting—involve expensive kernel transitions. Optimizing these operations requires careful attention to system call batching, memory allocation patterns, and avoiding unnecessary filesystem operations.

For our educational runtime, we'll prioritize code clarity over performance optimization, but understanding these trade-offs helps explain why production runtimes make certain architectural choices.

Container Runtime Interface (CRI) Integration

Modern container runtimes must integrate with orchestration systems like Kubernetes through the Container Runtime Interface (CRI). CRI defines a gRPC API that Kubernetes uses to manage container lifecycles: `RunPodSandbox` creates isolated environments, `CreateContainer` prepares container specifications, `StartContainer` begins process execution, and `StopContainer` terminates processes cleanly.

CRI Operation	Runtime Responsibilities	Kernel Interfaces Used
RunPodSandbox	Create shared namespace environment for pod containers	clone() with namespace flags, mount() for shared volumes
CreateContainer	Prepare container filesystem and configuration	mount() for overlayfs, cgroup filesystem writes
StartContainer	Execute container process in isolated environment	execve() after namespace/cgroup setup
StopContainer	Terminate process and cleanup resources	kill() signal delivery, umount(), cgroup cleanup
RemoveContainer	Clean up container metadata and temporary files	filesystem cleanup, cgroup removal

Understanding CRI integration reveals that **container runtimes are ultimately process lifecycle managers**. They coordinate kernel isolation mechanisms to create the illusion of independent machines while providing APIs that higher-level tools can use programmatically.

Key Design Insight: All container runtimes, regardless of their high-level architecture, ultimately perform the same fundamental operations: process isolation through namespaces, resource control through cgroups, filesystem layering through overlay mounts, and network connectivity through virtual interfaces. The differences lie in how they package these operations, what additional features they provide, and how they balance performance, security, and usability trade-offs.

Our educational runtime will implement these core operations directly, providing deep understanding of the kernel interfaces that production runtimes build upon. By understanding these fundamentals, developers gain the knowledge needed to debug container issues, optimize containerized applications, and make informed decisions about runtime selection in production environments.

The journey from understanding container concepts to implementing kernel interfaces reveals why containerization became so transformative: it provides process isolation and resource control that previously required separate virtual machines, but with much lower overhead and faster startup times. This efficiency comes from clever use of Linux kernel features that already existed—containers didn't require new kernel functionality, just creative orchestration of existing isolation primitives.

Implementation Guidance

This implementation guidance provides concrete technology choices and starter code to begin building your container runtime while focusing learning effort on the core isolation mechanisms rather than peripheral infrastructure.

Technology Recommendations:

Component	Simple Option	Advanced Option	Our Choice
HTTP API	Standard library <code>net/http</code> with JSON	<code>gRPC</code> with Protocol Buffers	Standard library (learning focus)
Configuration	JSON files with <code>encoding/json</code>	<code>YAML</code> with third-party parser	<code>JSON</code> (minimal dependencies)
Logging	Standard library <code>log</code> package	Structured logging (<code>logrus/zap</code>)	Standard library (simplicity)
Process Management	<code>os/exec</code> with direct system calls	Third-party process libraries	<code>os/exec + unix</code> package (direct kernel access)
Filesystem Operations	Standard library <code>os</code> and <code>filepath</code>	Third-party filesystem abstractions	Standard library (transparency)
Network Configuration	Direct netlink system calls	Third-party network libraries	golang.org/x/sys/unix (learning kernel interfaces)

Recommended File Structure:

```
container-runtime/
├── cmd/
│   ├── runtime/
│   │   └── main.go           ← CLI entry point
│   └── daemon/
│       └── main.go          ← Daemon mode entry point
├── internal/
│   ├── runtime/
│   │   ├── manager.go        ← Runtime Manager (orchestrates components)
│   │   ├── container.go       ← Container specification and state
│   │   └── manager_test.go
│   ├── namespace/
│   │   ├── namespace.go       ← Namespace Handler component
│   │   ├── mount.go           ← Mount namespace operations
│   │   ├── pid.go              ← PID namespace operations
│   │   └── namespace_test.go
│   ├── cgroup/
│   │   ├── controller.go      ← Cgroup Controller component
│   │   ├── memory.go           ← Memory controller operations
│   │   ├── cpu.go                ← CPU controller operations
│   │   └── controller_test.go
│   ├── filesystem/
│   │   ├── overlay.go          ← Filesystem Manager component
│   │   ├── layers.go            ← Layer management operations
│   │   └── overlay_test.go
│   ├── network/
│   │   ├── manager.go          ← Network Manager component
│   │   ├── bridge.go             ← Bridge networking setup
│   │   ├── veth.go                ← Virtual ethernet operations
│   │   └── manager_test.go
│   └── utils/
│       ├── syscalls.go         ← System call wrappers
│       └── paths.go              ← Path utilities
└── pkg/
    └── api/
        ├── types.go             ← Public API types
        └── client.go             ← Client library
├── configs/
│   └── runtime.json           ← Default configuration
├── scripts/
│   ├── setup-cgroups.sh        ← System setup helpers
│   └── cleanup-test-env.sh     ← Test cleanup utilities
└── docs/
    ├── design/
    └── examples/
```

Infrastructure Starter Code:

Since our focus is learning container runtime concepts rather than building JSON parsers or HTTP servers, here's complete starter code for peripheral infrastructure:

Configuration Management (`internal/runtime/container.go`):

```
package runtime

import (
    "encoding/json"
    "os"
)

// ContainerSpec defines what a container should look like

type ContainerSpec struct {

    ID          string      `json:"id"`
    Image       string      `json:"image"`
    Command     []string    `json:"command"`
    Environment map[string]string `json:"environment"`
    WorkingDir  string      `json:"working_dir"`

    // Resource limits

    Memory      int64       `json:"memory"`           // bytes
    CPUShares   int64       `json:"cpu_shares"`        // relative weight
    CPUQuota   int64       `json:"cpu_quota"`         // microseconds per period

    // Networking

    NetworkMode string     `json:"network_mode"`    // "bridge", "host", "none"
    PortMappings []PortMapping `json:"port_mappings"`

}

type PortMapping struct {

    HostPort     int        `json:"host_port"`
    ContainerPort int       `json:"container_port"`

}
```

GO

```
Protocol      string `json:"protocol"` // "tcp" or "udp"

}

// ContainerState tracks runtime information

type ContainerState struct {

    ID          string `json:"id"`

    Status      string `json:"status"` // "created", "running", "stopped"

    PID         int     `json:"pid"`   // container process PID

    ExitCode    int     `json:"exit_code"`

    // Resource handles for cleanup

    NamespacePath string `json:"namespace_path"`

    CgroupPath    string `json:"cgroup_path"`

    OverlayPath   string `json:"overlay_path"`

    NetworkNS    string `json:"network_ns"`

}

func LoadContainerSpec(path string) (*ContainerSpec, error) {

    data, err := os.ReadFile(path)

    if err != nil {

        return nil, err

    }

    var spec ContainerSpec

    if err := json.Unmarshal(data, &spec); err != nil {

        return nil, err

    }

}
```

```
    return &spec, nil
}

func (s *ContainerState) SaveToFile(path string) error {
    data, err := json.MarshalIndent(s, "", " ")
    if err != nil {
        return err
    }

    return os.WriteFile(path, data, 0644)
}
```

System Call Utilities ([internal/utils/syscalls.go](#)):

```
package utils

import (
    "fmt"
    "os"
    "syscall"
    "unsafe"

    "golang.org/x/sys/unix"
)

// NamespaceType represents different Linux namespace types

type NamespaceType int

const (
    CLONE_NEWPID    = 0x20000000
    CLONE_NEWNS     = 0x00020000
    CLONE_NEWNET    = 0x40000000
    CLONE_NEWUTS    = 0x04000000
    CLONE_NEWIPC    = 0x08000000
    CLONE_NEWUSER   = 0x10000000
)

// CreateNamespaces creates new namespaces for container isolation

func CreateNamespaces(namespaces []int) error {
    var flags uintptr

    for _, ns := range namespaces {
        flags |= uintptr(ns)
    }
}
```

GO

```
_ , _, errno := syscall.RawSyscall(syscall.SYS_UNSHARE, flags, 0, 0)

if errno != 0 {

    return fmt.Errorf("unshare failed: %v", errno)
}

return nil
}

// MountOverlayFS mounts overlayfs with specified layers

func MountOverlayFS(lowerDirs []string, upperDir, workDir, target string) error {

    // Build overlay mount options

    var lowers string

    for i, dir := range lowerDirs {

        if i > 0 {

            lowers += ";"

        }

        lowers += dir
    }

    options := fmt.Sprintf("lowerdir=%s,upperdir=%s,workdir=%s",
                           lowers, upperDir, workDir)

    return syscall.Mount("overlay", target, "overlay", 0, options)
}

// WriteToFile is a helper for writing to cgroup control files

func WriteToFile(path, value string) error {
```

```
    return os.WriteFile(path, []byte(value), 0644)

}

// EnsureDir creates directory if it doesn't exist

func EnsureDir(path string) error {

    return os.MkdirAll(path, 0755)

}
```

Core Logic Skeleton:

For the main Runtime Manager, provide skeleton with detailed TODOs mapping to the design concepts:

```
package runtime
```

GO

```
import (
    "fmt"
    "os/exec"
    "path/filepath"
)

// Manager orchestrates all container runtime components

type Manager struct {
    containerStore map[string]*ContainerState
    cgroupRoot     string
    overlayRoot    string
    bridgeName     string
}

func NewManager(cgroupRoot, overlayRoot, bridgeName string) *Manager {
    return &Manager{
        containerStore: make(map[string]*ContainerState),
        cgroupRoot:     cgroupRoot,
        overlayRoot:    overlayRoot,
        bridgeName:     bridgeName,
    }
}

// CreateContainer sets up container environment without starting process

func (m *Manager) CreateContainer(spec *ContainerSpec) error {
    // TODO 1: Validate container specification (check required fields, resource limits)
    // TODO 2: Create container state directory structure under /var/lib/container-runtime/
}
```

```
// TODO 3: Set up overlay filesystem layers (lower dirs from image, upper dir for
container)

// TODO 4: Create cgroup hierarchy for container resource control

// TODO 5: Prepare network namespace (create veth pair, attach to bridge)

// TODO 6: Save container state to persistent storage for later operations

// Hint: Don't start the container process yet - that's StartContainer's job

return fmt.Errorf("not implemented")

}

// StartContainer begins execution of container process in isolated environment

func (m *Manager) StartContainer(containerID string) error {

    // TODO 1: Load container state and validate it's in "created" status

    // TODO 2: Clone new process with namespace isolation flags (CLONE_NEWPID, etc.)

    // TODO 3: In child process: pivot to new root filesystem, mount /proc, setup
environment

    // TODO 4: In child process: join container cgroup, drop to non-root user if specified

    // TODO 5: In child process: exec the container command (this replaces clone child)

    // TODO 6: In parent process: update container state to "running", save PID

    // Hint: Use cmd.SysProcAttr to set namespace clone flags

    return fmt.Errorf("not implemented")

}

// StopContainer terminates container process and performs cleanup

func (m *Manager) StopContainer(containerID string) error {

    // TODO 1: Find container state and validate it's running

    // TODO 2: Send SIGTERM to container process, wait for graceful shutdown

    // TODO 3: If process doesn't exit in 10 seconds, send SIGKILL

    // TODO 4: Wait for process to exit and capture exit code

    // TODO 5: Update container state to "stopped" with exit code
```

```

    // TODO 6: Clean up temporary resources (but keep overlay layers for restart)

    return fmt.Errorf("not implemented")

}

// RemoveContainer performs full cleanup of stopped container

func (m *Manager) RemoveContainer(containerID string) error {

    // TODO 1: Verify container is stopped (not running)

    // TODO 2: Unmount overlay filesystem layers

    // TODO 3: Remove cgroup directory (this also kills any remaining processes)

    // TODO 4: Clean up network namespace and veth interfaces

    // TODO 5: Remove container state directory and all associated files

    // TODO 6: Remove container from in-memory store

    return fmt.Errorf("not implemented")

}

```

Language-Specific Implementation Hints:

Go System Call Interface:

- Use `golang.org/x/sys/unix` package for low-level system calls rather than the standard `syscall` package
- Container process creation requires `exec.Cmd` with `SysProcAttr.Cloneflags` set to namespace flags
- File descriptor handling: always defer `close()` calls and check for errors
- Cgroup operations are just filesystem writes to `/sys/fs/cgroup/` - no special API needed

Error Handling Patterns:

- Wrap system call errors with context: `fmt.Errorf("failed to mount overlay: %w", err)`
- For cleanup operations, continue cleanup even if individual steps fail, but accumulate errors
- Use `defer` functions for resource cleanup, but handle errors explicitly

Concurrency Considerations:

- Container operations can be concurrent, but individual container state changes should be atomic
- Use `sync.Mutex` to protect the container store map from concurrent access

- Child processes inherit parent's file descriptors - be careful about resource leaks

Milestone Checkpoints:

After implementing each component, verify functionality with these concrete tests:

Milestone 1 Checkpoint (Namespaces):

```
# Test PID namespace isolation                                BASH

sudo go run cmd/runtime/main.go create --spec examples/busybox.json

sudo go run cmd/runtime/main.go start busybox-test

# Inside container, 'ps aux' should show only container processes

# Container PID 1 should be your container command, not init

# Test mount namespace isolation

# Inside container, 'mount' should show only container mounts

# Host filesystem should not be visible except through overlay
```

Milestone 2 Checkpoint (Cgroups):

```
# Test memory limits                                         BASH

echo '{"memory": 134217728}' > test-config.json # 128MB limit

sudo go run cmd/runtime/main.go create --spec test-config.json

# Verify limit: cat /sys/fs/cgroup/container-test/memory.max should show 134217728

# Test CPU limits

# Run CPU-intensive process in container, verify it doesn't exceed quota

# Host 'top' should show container process respecting CPU limits
```

Milestone 3 Checkpoint (Overlay Filesystem):

```
# Test copy-on-write behavior                                BASH

sudo go run cmd/runtime/main.go start test-container

# Inside container: echo "test" > /usr/local/test-file

# After container stops, base image should be unchanged

# Check upper layer contains the new file

ls -la /var/lib/container-runtime/overlay/test-container/upper/usr/local/
```

Milestone 4 Checkpoint (Networking):

```
# Test container connectivity                                BASH

sudo go run cmd/runtime/main.go create --network bridge --ports 8080:80 web-test

sudo go run cmd/runtime/main.go start web-test

# Container should get IP address in bridge subnet

# Port 8080 on host should forward to port 80 in container

curl http://localhost:8080 # should reach container service
```

Common Debugging Scenarios:

Symptom	Likely Cause	Diagnosis Command	Fix
"Operation not permitted" on namespace creation	Missing CAP_SYS_ADMIN capability	<code>getcap \$(which your-runtime)</code>	Run with sudo or set capabilities
Container sees host processes	PID namespace not created properly	<code>ls -la /proc/self/ns/pid</code> in container	Check clone flags include CLONE_NEWPID
"No such file or directory" after pivot_root	New root is not a mount point	<code>mount grep new_root</code>	Bind mount new root to itself first
Overlay mount fails	Work directory not empty	<code>ls -la workdir/</code>	Ensure work directory is empty before mount
Network namespace setup fails	Bridge interface doesn't exist	<code>ip link show docker0</code>	Create bridge interface first
Container process becomes zombie	Parent not reaping child	<code>ps aux grep defunct</code>	Add proper signal handling and wait()

This implementation guidance provides working infrastructure code and detailed skeleton functions that map directly to the conceptual understanding built in the main design sections. Focus your learning effort on implementing the TODO sections rather than building peripheral infrastructure from scratch.

Goals and Non-Goals

Milestone(s): This section establishes the scope and boundaries for all milestones (1-4), defining what capabilities our container runtime will implement and what advanced features are intentionally excluded.

Mental Model: Building a Studio Apartment vs. a Luxury Hotel

Think of our container runtime project like deciding between building a studio apartment versus a luxury hotel. A studio apartment has everything you need to live comfortably—a bed, kitchen, bathroom, and living space—but it doesn't have a concierge, room service, or a swimming pool. Similarly, our minimal container runtime will provide all the essential isolation and resource management features that make containers work, but we're deliberately excluding the complex operational features that production container platforms require.

This analogy helps frame our decision-making: every feature request gets evaluated with "Is this essential plumbing (like water and electricity in an apartment) or is this a luxury amenity (like a hotel spa)?" Essential plumbing goes in; luxury amenities are explicitly excluded to keep the learning focused on core concepts.

The key insight is that understanding how to build the studio apartment—the fundamental isolation primitives—gives you the knowledge to later understand how luxury hotels work. But trying to build the hotel first would overwhelm you with operational complexity before you understand why containers need isolation in the first place.

Functional Goals

Our container runtime will implement four core capabilities that demonstrate the fundamental mechanisms underlying all container systems. These align directly with our four milestones and represent the minimum viable functionality for process isolation, resource control, filesystem layering, and network connectivity.

Process Isolation Through Namespaces

The runtime will create isolated process environments using Linux namespaces, demonstrating how containers achieve the illusion of running on separate machines while sharing the same kernel. This capability forms the foundation of container security and process isolation.

Namespace Type	Isolation Provided	Implementation Goal
PID	Process ID space	Container sees itself as PID 1, cannot see host processes
Mount	Filesystem view	Container has isolated filesystem tree, cannot access host mounts
Network	Network interfaces and routing	Container has private network stack with virtual interfaces
UTS	Hostname and domain name	Container can set its own hostname without affecting host
User	User and group ID mapping	Container processes run with mapped UIDs for security
IPC	Inter-process communication	Container has isolated message queues and shared memory

The runtime will handle the complex namespace creation sequence, including proper ordering (user namespace must be created first), filesystem pivoting using `pivot_root`, and the critical `/proc` remounting that many container tutorials skip. This goal specifically targets understanding how processes can have completely different views of the same system resources.

Design Insight: Process isolation is not about virtualization—it's about perspective. The same kernel and hardware serve multiple isolated views of system resources. Understanding this perspective-based isolation is crucial for grasping container security models.

Resource Control Using Cgroups

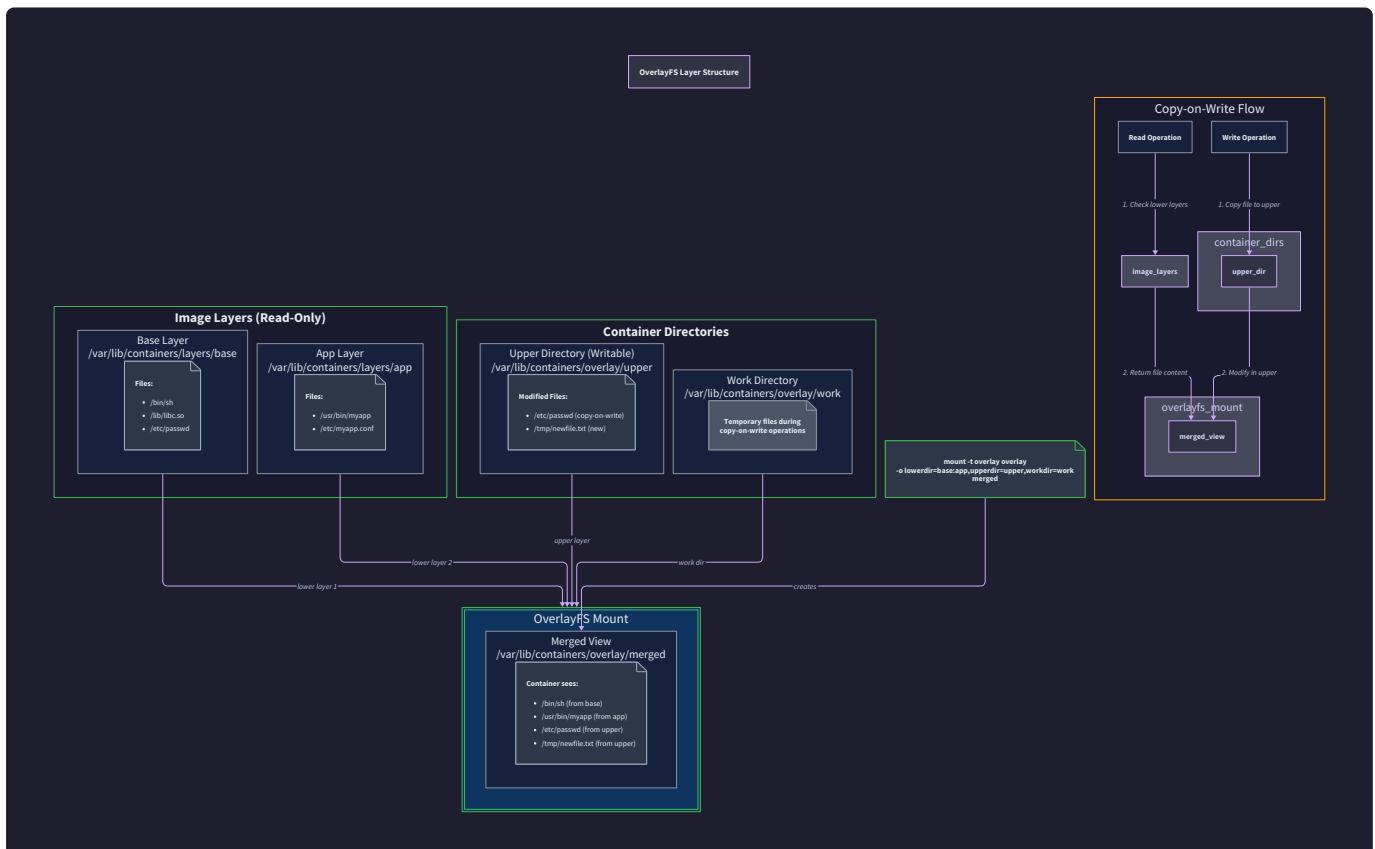
The runtime will enforce CPU, memory, and I/O limits using cgroups v2, demonstrating how containers prevent resource starvation and ensure predictable performance. This capability shows how multi-tenant systems maintain isolation not just for security, but for performance and reliability.

Resource Type	Control Mechanism	Enforcement Goal
Memory	<code>memory.max</code> hard limit	Container killed if it exceeds allocated memory
CPU	<code>cpu.max</code> quota and period	Container CPU time limited to specified percentage
I/O	<code>io.max</code> bandwidth limits	Container disk I/O throttled to prevent interference
PIDs	<code>pids.max</code> process limit	Container cannot fork-bomb the host system

The implementation will include resource usage monitoring through cgroup statistics files, graceful out-of-memory handling with configurable OOM behavior, and proper cgroup hierarchy management. This goal specifically targets understanding how the kernel enforces resource boundaries and how containers achieve predictable performance isolation.

Filesystem Layering with OverlayFS

The runtime will implement copy-on-write filesystem semantics using overlayfs, demonstrating how container images achieve efficient storage and fast startup times. This capability shows how containers can share base images while maintaining isolated filesystem changes.



Layer Type	Purpose	Behavior
Base Image Layers	Read-only image content	Shared across containers, never modified
Upper Layer	Container modifications	Captures all file changes during container lifetime
Work Directory	OverlayFS metadata	Temporary space for overlay operations
Merged View	Combined filesystem	What the container process sees as its root filesystem

The implementation will handle multiple image layer stacking, proper mount option configuration for overlayfs, and complete cleanup of overlay mounts and directories when containers are removed. This goal specifically targets understanding how modern container systems achieve both efficiency (shared layers) and isolation (private changes).

Basic Container Networking

The runtime will provide network connectivity using bridge networking with port mapping, demonstrating how containers communicate with each other and the external world while maintaining network namespace isolation. This capability shows how containers achieve network isolation without sacrificing connectivity.

Network Component	Function	Implementation Goal
veth Pair	Container-to-host connection	Virtual ethernet cable connecting container to host
Bridge	Inter-container networking	Virtual switch allowing containers to communicate
IP Assignment	Container addressing	Automatic IP allocation from configurable subnet
Port Forwarding	External access	NAT rules mapping host ports to container services
DNS Resolution	Name resolution	Configurable nameserver settings for container

The implementation will create and manage veth pairs, configure bridge networking for multi-container communication, implement port mapping using iptables NAT rules, and handle proper cleanup of network interfaces and rules. This goal specifically targets understanding how containers maintain network isolation while enabling controlled connectivity.

Architecture Decision: Bridge Networking Over Host Networking

- **Context:** Containers need network connectivity while maintaining isolation from the host network stack
- **Options Considered:**
 - Host networking (share host network namespace)
 - Bridge networking (isolated namespace with virtual bridge)
 - Overlay networking (software-defined networking across hosts)
- **Decision:** Bridge networking with veth pairs
- **Rationale:** Bridge networking provides the right balance of isolation and simplicity for educational purposes. Host networking sacrifices isolation (containers see all host network interfaces), while overlay networking introduces complex distributed systems concepts that distract from core container isolation mechanisms.
- **Consequences:** Containers get network isolation and controlled connectivity, but we limit ourselves to single-host networking (no multi-host container communication).

Explicit Non-Goals

Understanding what we will **not** build is equally important for keeping the project focused on learning container fundamentals. These non-goals represent production-grade features that would significantly increase complexity without proportionally increasing understanding of core isolation mechanisms.

Image Distribution and Registry Management

We will **not** implement image pulling from remote registries, image caching, or image building capabilities. Our runtime will assume container images already exist as local directory structures or tar files that can be extracted locally.

Feature	Why Excluded	Learning Impact
Remote image pulling	Requires HTTP clients, authentication, and error handling for network failures	Distracts from isolation primitives with network programming
Image caching	Requires content-addressable storage and garbage collection	Adds storage management complexity unrelated to containers
Image building	Requires Dockerfile parsing and build context management	Separate problem domain from runtime execution
Image signing/verification	Requires cryptographic libraries and PKI understanding	Security topic separate from isolation mechanisms

Students will create simple local images by preparing directory structures with the files they want in their container filesystem. This keeps the focus on how overlayfs combines layers rather than how images are distributed across networks.

Container Orchestration and Scheduling

We will **not** implement multi-container coordination, service discovery, rolling updates, or cluster management. Our runtime manages individual containers in isolation without understanding relationships between containers.

Feature	Why Excluded	Learning Impact
Service discovery	Requires distributed coordination and consensus algorithms	Adds distributed systems complexity
Load balancing	Requires traffic routing and health checking	Network management separate from isolation
Rolling updates	Requires deployment strategies and state management	Operations concern, not isolation primitive
Resource scheduling	Requires cluster-wide resource tracking and optimization	Distributed systems problem

Students will start containers individually and manage them through direct runtime commands. This keeps the focus on how a single container achieves isolation rather than how multiple containers work together in production systems.

Production-Grade Security Features

We will **not** implement advanced security policies, capability management, or mandatory access controls. Our runtime will use basic user namespace mapping but exclude enterprise security features.

Feature	Why Excluded	Learning Impact
SELinux/AppArmor integration	Requires understanding mandatory access control systems	Security specialization separate from container basics
Seccomp filters	Requires system call filtering and security policy languages	Advanced security topic
Capability management	Requires deep Linux capability system knowledge	Security detail that obscures main learning objectives
Runtime security scanning	Requires vulnerability databases and policy engines	Operations security concern

Students will learn how user namespaces provide basic privilege separation, but advanced security policies are left for specialized security courses. This keeps the focus on how isolation works rather than how to secure it against sophisticated attacks.

Advanced Networking Features

We will **not** implement overlay networks, service meshes, or multi-host networking. Our networking will be limited to single-host bridge networking with basic port forwarding.

Feature	Why Excluded	Learning Impact
Overlay networks	Requires distributed networking and tunnel management	Network virtualization separate from container isolation
Service mesh integration	Requires sidecar proxy management and service discovery	Microservices architecture concern
CNI plugin support	Requires plugin architecture and network policy management	Standardization complexity
Multi-host networking	Requires cluster networking and distributed state	Distributed systems problem

Students will connect containers through a simple bridge and access them through port forwarding. This demonstrates the core concept of network namespace isolation without the complexity of distributed networking.

Performance Optimization and Scalability

We will **not** optimize for high-performance container startup, massive container density, or enterprise-scale resource management. Our runtime prioritizes code clarity over performance optimization.

Feature	Why Excluded	Learning Impact
Fast container startup	Requires optimization techniques that obscure basic operations	Performance engineering separate from learning isolation
High container density	Requires resource optimization and sharing strategies	Production scalability concern
Resource pooling	Requires complex resource management and sharing algorithms	Operations optimization
Monitoring and metrics	Requires metrics collection and time-series storage	Observability tooling separate from core function

Students will focus on correctness and understanding rather than performance. A container that takes 2 seconds to start but clearly demonstrates namespace creation is better for learning than one that starts in

50ms but hides the setup complexity.

Design Insight: Educational projects require ruthless scope discipline. Every feature that doesn't directly teach the core concepts is a potential distraction that prevents deep understanding of the fundamentals. Production systems need all these excluded features, but learning systems need focus.

Backward Compatibility and Standards Compliance

We will **not** implement full OCI (Open Container Initiative) compatibility, Docker API compatibility, or support for legacy container formats. Our runtime will use simple configuration formats optimized for learning rather than industry standards.

Feature	Why Excluded	Learning Impact
OCI runtime spec compliance	Requires implementing complex specification details	Standards compliance distracts from core concepts
Docker API compatibility	Requires REST API implementation and Docker protocol details	API design separate from isolation mechanisms
Legacy format support	Requires supporting multiple image and config formats	Backward compatibility complexity
Standardized logging	Requires structured logging and log driver architecture	Operational tooling concern

Students will use simple JSON configuration files and direct command-line interaction with the runtime. This removes the cognitive load of learning industry specifications while focusing on the underlying isolation mechanisms that all standards ultimately implement.

Success Criteria and Validation

Our functional goals will be validated through specific behavioral tests that demonstrate successful isolation and resource control. These criteria define what "working" means for each capability.

Goal	Success Criterion	Validation Method
Process Isolation	Container process sees itself as PID 1, cannot see host processes	Run <code>ps aux</code> inside container, verify only container processes visible
Resource Control	Container respects memory limit and is killed when exceeded	Start container with 50MB limit, allocate 100MB, verify OOM kill
Filesystem Layering	Changes inside container don't affect base image	Modify files in running container, verify base layer unchanged
Network Connectivity	Container can reach external network and accept incoming connections	Ping external host from container, curl container service from host

These validation methods provide concrete evidence that our runtime successfully implements container isolation without relying on complex testing infrastructure or production-scale scenarios.

Implementation Guidance

This section provides practical guidance for implementing the goals and respecting the non-goals throughout the development process.

Technology Recommendations

Component	Simple Option	Advanced Option (Avoid)
Configuration Format	JSON files with simple schemas	YAML with complex validation, OCI specs
Namespace Creation	Direct <code>syscall.SyscallN()</code> calls	High-level abstraction libraries
Filesystem Operations	<code>os</code> and <code>syscall</code> packages	Virtual filesystem libraries
Network Management	<code>netlink</code> package for basic operations	CNI plugins, complex networking libraries
Resource Monitoring	Direct cgroup file reading	Prometheus metrics, complex telemetry
Error Handling	Simple error returns and logging	Complex error categorization, retries

Recommended File Structure

Organize the codebase to reflect the four main functional areas while avoiding the complexity that production systems require:

```
container-runtime/
├── cmd/
│   └── container/
│       └── main.go                                ← Simple CLI interface
└── internal/
    ├── manager/
    │   ├── manager.go                            ← ContainerSpec and lifecycle coordination
    │   └── manager_test.go
    ├── namespace/
    │   ├── namespace.go                          ← CreateNamespaces and pivot_root logic
    │   └── namespace_test.go
    ├── cgroup/
    │   ├── cgroup.go                            ← Resource limit enforcement
    │   └── cgroup_test.go
    ├── filesystem/
    │   ├── overlay.go                           ← MountOverlayFS implementation
    │   └── overlay_test.go
    └── network/
        ├── bridge.go                            ← veth and bridge management
        └── bridge_test.go
└── examples/
    ├── alpine-image/                         ← Simple container image for testing
    └── container-config.json                 ← Example ContainerSpec
docs/
└── debugging-guide.md                      ← Common issues and solutions
```

This structure keeps related functionality together while avoiding the multi-layered abstractions that production runtimes use for extensibility.

Core Data Structures

Define simple, focused data structures that capture the essential information without over-engineering for extensibility:

GO

```
// ContainerSpec represents the desired container configuration

type ContainerSpec struct {

    ID          string      `json:"id"`

    Image       string      `json:"image"`           // Local directory path

    Command     []string    `json:"command"`

    WorkingDir  string      `json:"workingDir"`

    Environment map[string]string `json:"environment"`

    Resources   ResourceLimits `json:"resources"`

    Network     NetworkConfig `json:"network"`

}

// ResourceLimits defines cgroup resource constraints

type ResourceLimits struct {

    Memory int64 `json:"memory"` // bytes

    CPU    int64 `json:"cpu"`   // CPU quota in microseconds per 100ms period

    PIDs   int64 `json:"pids"` // maximum number of processes

}

// NetworkConfig specifies container networking

type NetworkConfig struct {

    PortMappings []PortMapping `json:"portMappings"`

    DNS        []string      `json:"dns"`

}
```

Scope Discipline Implementation

Throughout development, apply these filters to avoid scope creep:

Filter 1: Core Learning Objective Test

"Does this feature directly teach namespace isolation, resource control, filesystem layering, or basic networking?"

If the answer is no, the feature belongs in the non-goals category.

Filter 2: Complexity-to-Learning Ratio Test

"Does implementing this feature require more than 50 lines of code unrelated to the core isolation mechanisms?"

If yes, consider whether a simpler alternative achieves the same learning objective.

Filter 3: Production Readiness Test (Reverse Filter)

"Would this feature be required for production use?"

If yes, and it's not in our functional goals, it likely belongs in non-goals. Our runtime should be educational, not production-ready.

Milestone Checkpoints

After implementing each functional goal, validate success using these concrete checkpoints:

Milestone 1 Checkpoint - Process Isolation:

```
# Start container with simple command                                BASH
./container run examples/alpine-image /bin/sh

# Inside container, verify isolation

ps aux      # Should only show container processes
hostname    # Should show container-specific hostname
mount       # Should show isolated mount namespace
```

Milestone 2 Checkpoint - Resource Control:

```
# Start container with memory limit  
  
../container run --memory=50MB examples/alpine-image stress --vm 1 --vm-bytes 100M  
  
# Verify OOM kill occurs  
  
echo $?          # Should be non-zero exit code indicating kill
```

BASH

Milestone 3 Checkpoint - Filesystem Layering:

```
# Start container and modify filesystem  
  
../container run examples/alpine-image /bin/sh  
  
# Inside container: echo "test" > /tmp/container-file  
  
# Verify base image unchanged  
  
ls examples/alpine-image/tmp/    # Should not contain container-file
```

BASH

Milestone 4 Checkpoint - Networking:

```
# Start container with port mapping  
  
../container run --port=8080:80 examples/alpine-image httpd -f  
  
# Test connectivity  
  
curl localhost:8080           # Should reach container service
```

BASH

Common Implementation Pitfalls

⚠ Pitfall: Feature Creep Through "Just One More Thing"

The most common failure mode is gradually adding features that seem "small" but collectively violate the non-goals. Examples include:

- Adding image format auto-detection ("it's just checking file extensions")
- Implementing container logs ("it's just redirecting stdout")
- Adding health checks ("it's just running a command periodically")

Fix: Maintain a strict feature log. Every addition must be explicitly justified against the functional goals and reviewed for scope creep.

⚠ Pitfall: Over-Engineering for Extensibility

Educational code often becomes too abstract when developers anticipate future features that are in the non-goals list.

Fix: Write the simplest code that demonstrates the concept. Prefer concrete implementations over abstract interfaces when the abstraction only serves excluded features.

Pitfall: Incomplete Non-Goal Documentation

Teams often agree on non-goals but don't document them clearly, leading to scope debates during implementation.

Fix: For every rejected feature request, add it to the non-goals list with a clear explanation of why it's excluded. This prevents re-litigation of scope decisions.

High-Level Architecture

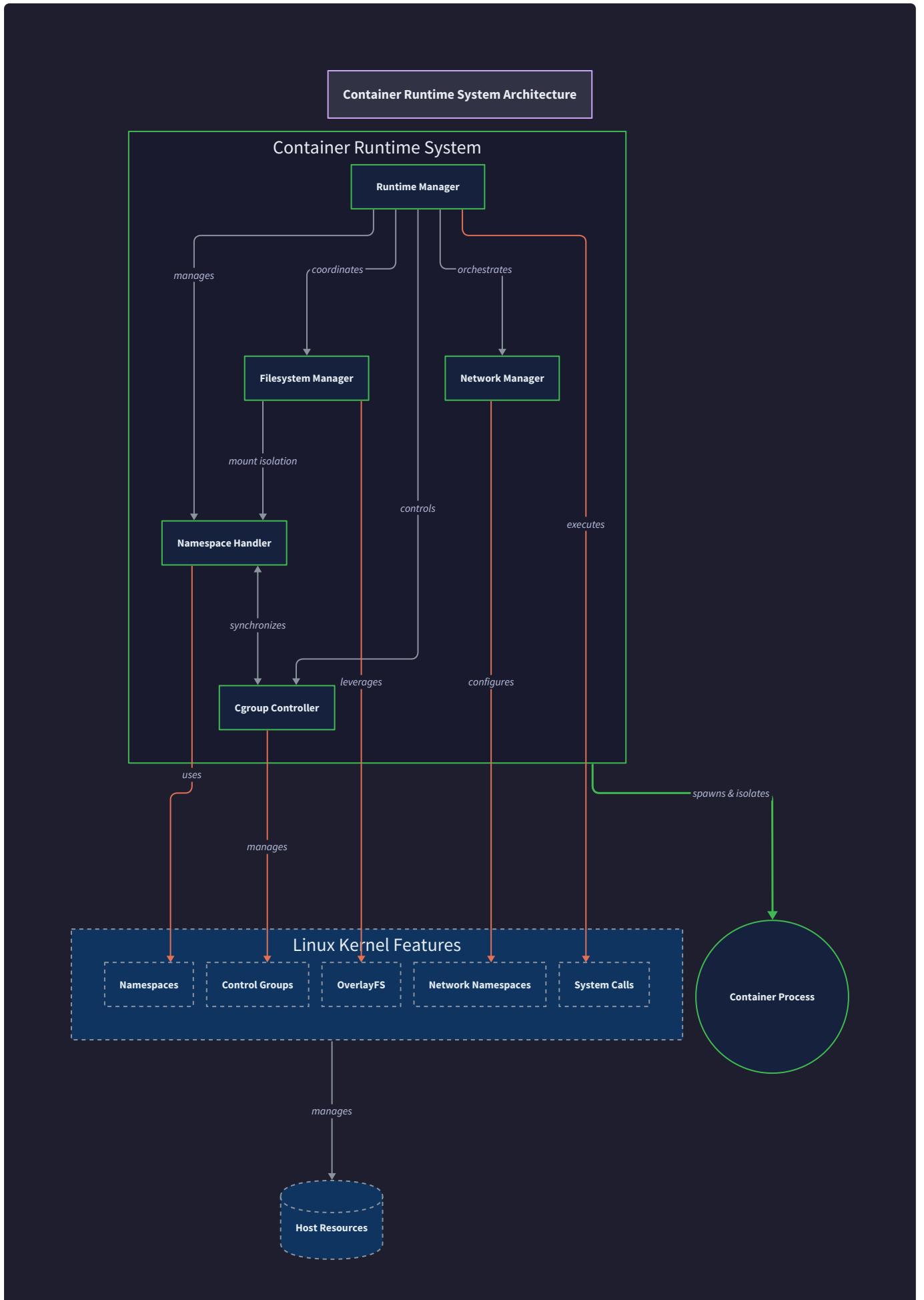
Milestone(s): This section establishes the overall system design for all milestones (1-4), showing how the runtime manager orchestrates namespace creation, resource control, filesystem layering, and networking components.

Mental Model: The Orchestra Conductor

Think of our container runtime as an **orchestra conductor** managing a complex performance. Just as a conductor coordinates different instrument sections—strings, brass, woodwinds, and percussion—our runtime manager coordinates different isolation mechanisms: namespaces for process isolation, cgroups for resource limits, overlayfs for filesystem layering, and network components for connectivity.

The conductor doesn't play any instruments directly, but ensures each section comes in at the right time, plays in harmony, and follows the overall score. Similarly, our runtime manager doesn't implement the low-level kernel features directly, but orchestrates when namespaces get created, when cgroups are configured, when filesystems are mounted, and when network interfaces are established.

Just as a conductor must handle mistakes gracefully—if the violins come in too early or the drums miss their cue—our runtime manager must handle partial failures where some components succeed while others fail, ensuring the entire system reaches a consistent state.



Our container runtime architecture centers around a **Manager** component that orchestrates four specialized subsystems, each responsible for a different aspect of container isolation and resource management. This design follows the principle of separation of concerns, where each component has a clearly defined responsibility and well-defined interfaces for interaction with other components.

The architecture addresses the fundamental challenge of container runtimes: **coordinating multiple kernel-level isolation mechanisms** that must be set up in the correct order, configured consistently, and cleaned up properly when containers are removed. Each kernel feature—namespaces, cgroups, overlayfs, and network interfaces—has its own setup requirements, failure modes, and cleanup procedures.

Decision: Component-Based Architecture with Central Orchestration

- **Context:** Container creation involves coordinating multiple kernel features that must be configured in a specific sequence, with each step depending on the success of previous steps.
- **Options Considered:**
 1. Monolithic design with all logic in a single component
 2. Component-based design with central orchestrator
 3. Pipeline design with sequential processing stages
- **Decision:** Component-based design with central Manager orchestrating specialized components
- **Rationale:** This approach provides clear separation of concerns, making each component testable in isolation while maintaining centralized control over the complex sequencing requirements. The Manager can implement sophisticated error recovery since it has visibility into all component states.
- **Consequences:** Enables easier testing and debugging of individual isolation mechanisms, but requires careful interface design between components and adds complexity to the Manager's coordination logic.

The following table compares our architectural options:

Option	Pros	Cons	Chosen?
Monolithic	Simple control flow, easier debugging	Hard to test individual features, complex error recovery	No
Component-based	Clear separation, testable components, extensible	Interface complexity, coordination overhead	Yes
Pipeline	Natural sequential flow, easy to add stages	Rigid ordering, limited error recovery options	No

Component Responsibilities

Each component in our architecture owns a specific domain of container isolation, with clearly defined responsibilities and interfaces. The **Manager** serves as the central orchestrator, while specialized components handle the technical details of their respective kernel features.

Manager Component

The **Manager** component acts as the central orchestrator for all container operations, implementing the primary API that external clients use to manage container lifecycles. It maintains the authoritative state of all active containers and coordinates the complex sequencing required for container creation, startup, shutdown, and cleanup.

Responsibility	Details	Component Interface
Container Lifecycle	Manages complete container lifecycle from creation through removal	<code>CreateContainer(spec)</code> , <code>StartContainer(id)</code> , <code>StopContainer(id)</code> , <code>RemoveContainer(id)</code>
State Management	Tracks active containers, their PIDs, resource allocations, and cleanup requirements	<code>GetContainer(id)</code> , <code>ListContainers()</code>
Operation Sequencing	Ensures components are called in correct order during container operations	Internal coordination with all components
Error Recovery	Handles partial failures by cleaning up successfully created resources	Rollback procedures for each operation type
Configuration Loading	Loads and validates container specifications from configuration files	<code>LoadContainerSpec(path)</code>

The Manager maintains a **ContainerState** registry that tracks all active containers and their associated resources. This registry serves as the source of truth for cleanup operations and enables the Manager to handle system restarts by detecting orphaned resources.

The critical insight for the Manager design is that container operations are **transactions** across multiple kernel subsystems. Like database transactions, they must either complete fully or be rolled back completely, leaving no partial state that could cause resource leaks or inconsistent behavior.

Namespace Handler Component

The **Namespace Handler** specializes in Linux namespace manipulation, creating isolated execution environments for container processes. It understands the subtle requirements and ordering constraints of different namespace types, handling the complex filesystem operations required for mount namespace isolation.

Namespace Type	Isolation Provided	Setup Requirements	Handler Responsibility
PID	Process ID space isolation	Clone with <code>CLONE_NEWPID</code>	<code>CreatePIDNamespace()</code> , ensure container process becomes PID 1
Mount	Filesystem view isolation	Pivot root operation, /proc mounting	<code>CreateMountNamespace()</code> , <code>PivotRoot()</code> , mount essential filesystems
Network	Network stack isolation	Create network namespace, configure interfaces	<code>CreateNetworkNamespace()</code> , coordinate with Network Manager
UTS	Hostname and domain isolation	Set container hostname independently	<code>CreateUTSNamespace()</code> , <code>SetHostname()</code>
User	User ID mapping and capabilities	Configure UID/GID mapping for unprivileged containers	<code>CreateUserNamespace()</code> , <code>ConfigureUIDMapping()</code>

The Namespace Handler encapsulates the complex requirements for namespace setup, particularly the filesystem operations required for mount namespace isolation. It handles the `pivot_root` system call, which requires careful preparation of mount points and proper sequencing to avoid common failures.

Cgroup Controller Component

The **Cgroup Controller** manages resource limits and monitoring using Linux cgroups v2, providing fine-grained control over CPU, memory, and I/O resources allocated to containers. It abstracts the complexity of the cgroup filesystem interface and provides high-level resource management operations.

Resource Type	Control Mechanism	Configuration Parameters	Controller Responsibility
Memory	Hard and soft limits, OOM behavior	<code>memory.max</code> , <code>memory.high</code> , <code>memory.oom.group</code>	<code>SetMemoryLimit()</code> , <code>GetMemoryUsage()</code> , <code>ConfigureOOMBehavior()</code>
CPU	Quota and period-based scheduling	<code>cpu.max</code> , <code>cpu.weight</code>	<code>SetCPULimit()</code> , <code>GetCPUUsage()</code>
I/O	Bandwidth and IOPS limits	<code>io.max</code> , <code>io.weight</code>	<code>SetIOLimits()</code> , <code>GetIOStats()</code>
PIDs	Maximum process count	<code>pids.max</code>	<code>SetPIDLimit()</code> , <code>GetPIDCount()</code>

The Cgroup Controller handles the hierarchical nature of cgroups, creating per-container cgroup directories under a runtime-managed hierarchy. It provides resource monitoring capabilities that enable the Manager to detect resource pressure and implement proactive management policies.

Decision: Cgroups v2 Unified Hierarchy

- **Context:** Linux systems support both cgroups v1 (legacy) and v2 (unified hierarchy), with different interfaces and capabilities.
- **Options Considered:**
 1. Support only cgroups v1 for maximum compatibility
 2. Support only cgroups v2 for modern features
 3. Dual support with runtime detection
- **Decision:** Target cgroups v2 unified hierarchy exclusively
- **Rationale:** Cgroups v2 provides a cleaner interface, better resource isolation, and is the future direction for Linux. Most modern systems support v2, and the complexity of dual support outweighs compatibility benefits for an educational runtime.
- **Consequences:** Requires Linux 4.5+ with cgroups v2 enabled, but provides cleaner implementation and better resource control features.

Filesystem Manager Component

The **Filesystem Manager** implements image layering using overlayfs, providing copy-on-write semantics that enable efficient container filesystem isolation. It manages the complex directory structure required by overlayfs and handles layer composition from multiple image layers.

Operation	Purpose	Implementation Details	Manager Responsibility
Layer Preparation	Extract and prepare image layers	Create directory structure for lower layers	<code>PrepareImageLayers(imageID)</code>
Overlay Mounting	Combine layers into unified view	Mount overlayfs with proper options	<code>MountOverlayFS(layers, target)</code>
Copy-on-Write	Handle file modifications efficiently	Ensure changes stay in upper layer	Transparent through overlays
Cleanup	Remove overlay mounts and directories	Unmount and clean up layer directories	<code>CleanupContainer(containerID)</code>

The Filesystem Manager maintains a **layer cache** that tracks extracted image layers, enabling layer reuse across multiple containers that share common base images. This significantly reduces storage overhead and container startup time.

The component handles the specific requirements of overlayfs, including the need for separate upper, work, and lower directories, and the requirement that the work directory be on the same filesystem as the upper directory for atomic operations.

Network Manager Component

The **Network Manager** implements container networking using virtual ethernet pairs and bridge networking, providing network isolation while enabling both inter-container communication and external connectivity. It manages IP address allocation and implements port forwarding using iptables NAT rules.

Networking Feature	Implementation	Configuration	Manager Responsibility
Container Isolation	Create network namespace per container	Coordinate with Namespace Handler	<code>CreateContainerNetwork()</code>
Inter-container Communication	Bridge networking with veth pairs	Create and configure bridge interface	<code>SetupBridge()</code> , <code>ConnectContainer()</code>
IP Address Management	Subnet-based IP allocation	CIDR pool for automatic assignment	<code>AllocateIP()</code> , <code>ReleaseIP()</code>
External Connectivity	NAT rules for port forwarding	iptables rules for host-to-container access	<code>ConfigurePortForwarding()</code>
DNS Resolution	Configure container DNS settings	Set up /etc/resolv.conf in container	<code>ConfigureDNS()</code>

The Network Manager coordinates closely with the Namespace Handler to ensure network namespaces are created before network interfaces are configured. It maintains an **IP allocation registry** to prevent address conflicts and enable proper cleanup when containers are removed.

The key architectural challenge for networking is that network configuration spans multiple namespaces—host and container—requiring careful coordination of operations that occur in different network contexts.

Common Pitfalls

⚠ Pitfall: Component Initialization Order Many developers assume components can be initialized independently, but container creation requires strict ordering. For example, the mount namespace must be created before overlays can be mounted, and network namespace must exist before veth pairs can be configured. The Manager must enforce these dependencies explicitly.

⚠ Pitfall: Shared State Between Components Components that share state directly create tight coupling and make error recovery difficult. Each component should own its state completely, communicating with other components only through the Manager's coordination layer.

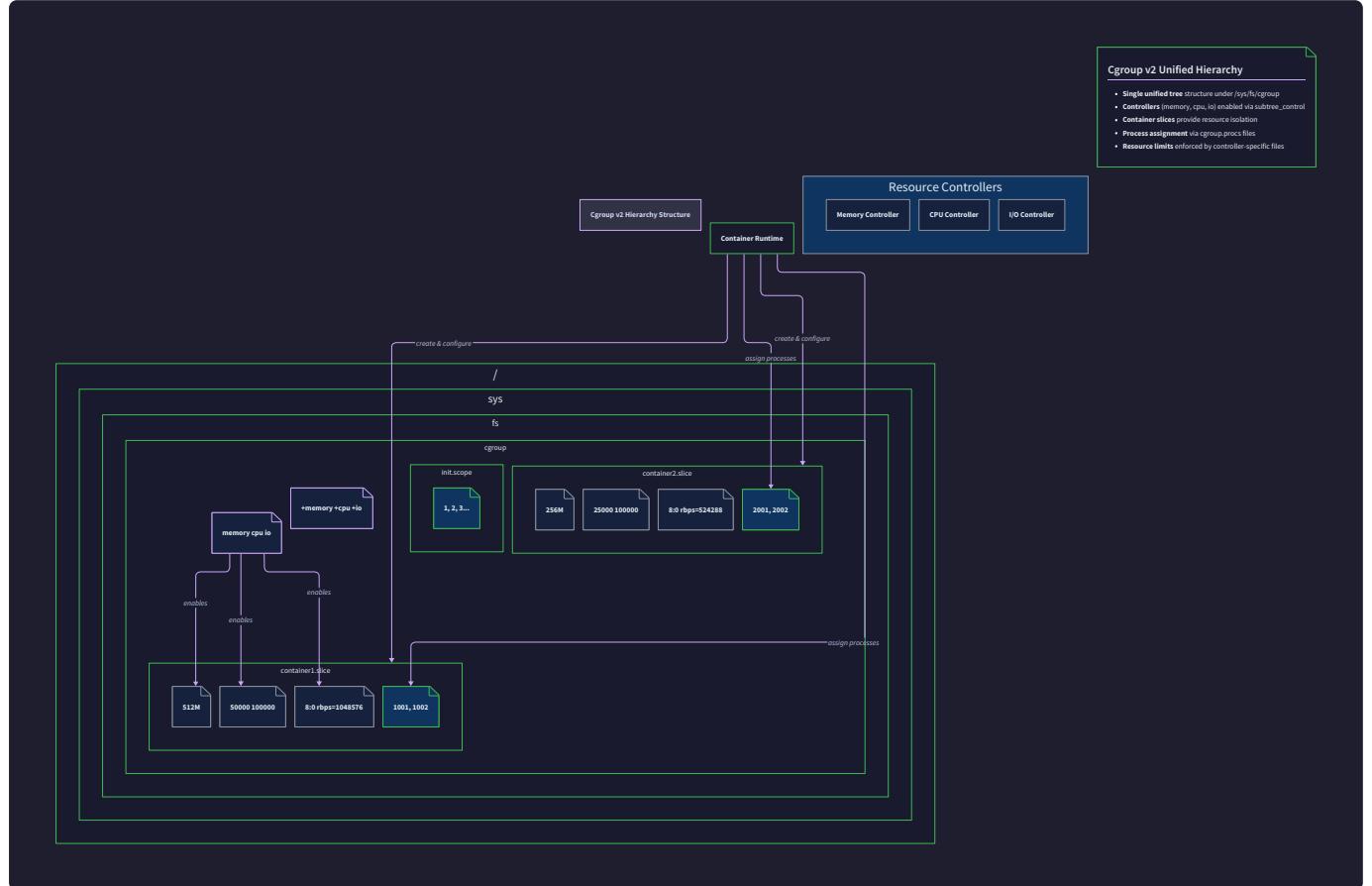
⚠ Pitfall: Incomplete Error Recovery When container creation fails partway through, developers often forget to clean up resources created by earlier components. Each component must provide cleanup methods that the Manager can call during error recovery, and these must be idempotent.

⚠ Pitfall: Resource Leak on Manager Restart If the Manager process crashes, it loses track of active containers and their associated resources (cgroups, mount points, network interfaces). The Manager must

implement startup recovery that scans for orphaned resources and either adopts or cleans them up.

Recommended File Structure

The codebase organization reflects our component-based architecture, with clear module boundaries that align with component responsibilities. This structure makes it easy to develop, test, and maintain each component independently while keeping integration points well-defined.



```
container-runtime/
├── cmd/
│   ├── runtime/
│   │   └── main.go                                # Main runtime daemon entry point
│   └── cli/
│       └── main.go                                # Command-line client for runtime API
└── internal/
    ├── manager/
    │   ├── manager.go                             # Central orchestrator component
    │   ├── container.go                          # Container state management
    │   ├── config.go                            # Configuration loading and validation
    │   └── manager_test.go                      # Integration tests for Manager
    ├── namespace/
    │   ├── handler.go                           # Namespace creation and management
    │   ├── mount.go                            # Mount namespace and pivot_root operations
    │   ├── types.go                            # Namespace type constants and structures
    │   └── handler_test.go                     # Namespace operation unit tests
    ├── cgroup/
    │   ├── controller.go                      # Cgroup v2 resource control
    │   ├── memory.go                           # Memory controller specific operations
    │   ├── cpu.go                             # CPU controller specific operations
    │   └── controller_test.go                 # Cgroup management unit tests
    ├── filesystem/
    │   ├── overlay.go                          # OverlayFS mounting and layer management
    │   ├── layers.go                           # Image layer extraction and caching
    │   ├── cleanup.go                          # Filesystem cleanup operations
    │   └── overlay_test.go                    # Filesystem operation unit tests
    ├── network/
    │   ├── manager.go                          # Network setup and configuration
    │   ├── bridge.go                           # Bridge networking implementation
    │   ├── veth.go                            # Virtual ethernet pair management
    │   ├── iptables.go                         # NAT rules for port forwarding
    │   └── manager_test.go                   # Network configuration unit tests
    └── types/
        ├── container.go                      # ContainerSpec and ContainerState definitions
        ├── resources.go                      # ResourceLimits and monitoring types
        └── network.go                        # NetworkConfig and PortMapping types
└── pkg/
    ├── api/
    │   ├── runtime.go                         # Public API interfaces
    │   └── client.go                          # Client library for runtime API
    └── utils/
        ├── syscalls.go                        # System call wrappers
        └── paths.go                           # Path manipulation utilities
└── configs/
    ├── default.yaml                         # Default runtime configuration
    └── examples/
        ├── alpine.yaml                        # Example Alpine container spec
        └── nginx.yaml                         # Example Nginx container spec
└── test/
    ├── integration/
    │   ├── container_lifecycle_test.go      # End-to-end container tests
    │   └── networking_test.go              # Multi-container networking tests
```

```
|   └── fixtures/
|       ├── test-images/          # Test container images
|       └── configs/            # Test configuration files
└── docs/
    ├── api.md                # API documentation
    └── troubleshooting.md     # Common issues and solutions
```

The **internal** directory contains all implementation components that are not exposed as public APIs. Each component has its own package with clearly defined interfaces, making it easy to develop and test components independently.

The **types** package serves as a central location for all data structures shared between components, preventing circular dependencies while ensuring consistent type definitions across the codebase.

The **pkg** directory contains public APIs that external clients can use to interact with our runtime, following Go conventions for library organization.

Decision: Internal Package Organization

- **Context:** Component interfaces need to be accessible to the Manager for orchestration, but shouldn't be exposed to external clients.
- **Options Considered:**
 1. All components in single package with public interfaces
 2. Separate packages with internal interfaces
 3. Separate packages with public interfaces in pkg/
- **Decision:** Separate internal packages with shared types package
- **Rationale:** This provides clear component boundaries while preventing external dependencies on internal implementation details. The shared types package avoids circular dependencies between component packages.
- **Consequences:** Enables independent component development and testing, but requires careful interface design to minimize coupling between packages.

Implementation Guidance

Our component-based architecture requires careful attention to interface design and dependency management. The following guidance helps implement clean component boundaries while maintaining the flexibility needed for container runtime operations.

Technology Recommendations

Component	Simple Option	Advanced Option
IPC	Direct Go method calls within process	gRPC interfaces for potential distribution
Configuration	YAML files with <code>gopkg.in/yaml.v3</code>	JSON Schema validation with structured configs
Logging	Standard <code>log</code> package with structured output	<code>github.com/sirupsen/logrus</code> with contextual fields
Error Handling	Standard Go error handling with wrapped errors	Custom error types with error codes and recovery hints
State Storage	In-memory maps with file-based persistence	Embedded database like BoltDB for reliable state
Testing	Standard <code>testing</code> package with table-driven tests	<code>github.com/stretchr/testify</code> for assertions and mocks

File Structure Implementation

The recommended file structure supports clean separation between components while maintaining clear integration points. Here's how to implement the core structure:

```
// internal/types/container.go - Central type definitions          GO

package types

// ContainerSpec defines the desired state of a container

type ContainerSpec struct {

    ID      string      `yaml:"id" json:"id"`

    Image   string      `yaml:"image" json:"image"`

    Command []string    `yaml:"command" json:"command"`

    WorkingDir string    `yaml:"workdir" json:"workdir"`

    Environment map[string]string `yaml:"environment" json:"environment"`

    Resources   ResourceLimits `yaml:"resources" json:"resources"`

    Network     NetworkConfig  `yaml:"network" json:"network"`

}

// ContainerState tracks the runtime state of an active container

type ContainerState struct {

    Spec      ContainerSpec `json:"spec"`

    Status    ContainerStatus `json:"status"`

    PID       int           `json:"pid"`

    CreatedAt time.Time    `json:"created_at"`

    StartedAt *time.Time   `json:"started_at,omitempty"`

    CgroupPath string        `json:"cgroup_path"`

    OverlayPath string        `json:"overlay_path"`

    NetworkNS  string        `json:"network_ns"`

    VethHost   string        `json:"veth_host"`

    VethContainer string        `json:"veth_container"`

    IPAddress  string        `json:"ip_address"`

}
```

```
type ContainerStatus string

const (
    StatusCreated ContainerStatus = "created"
    StatusRunning ContainerStatus = "running"
    StatusStopped ContainerStatus = "stopped"
    StatusError   ContainerStatus = "error"
)

// ResourceLimits defines resource constraints for containers

type ResourceLimits struct {

    Memory int64 `yaml:"memory" json:"memory"`           // bytes
    CPU     int64 `yaml:"cpu" json:"cpu"`                  // CPU quota in microseconds per 100ms
    period
    PIDs   int64 `yaml:"pids" json:"pids"`                // maximum number of processes
}

// NetworkConfig defines networking configuration for containers

type NetworkConfig struct {

    PortMappings []PortMapping `yaml:"port_mappings" json:"port_mappings"`
    DNS          []string      `yaml:"dns" json:"dns"`
}

// PortMapping defines a host-to-container port forward

type PortMapping struct {

    HostPort     int      `yaml:"host_port" json:"host_port"`
    ContainerPort int      `yaml:"container_port" json:"container_port"`
    Protocol     string   `yaml:"protocol" json:"protocol"` // "tcp" or "udp"
}
```

}

GO

```
// internal/manager/manager.go - Core orchestration component

package manager

import (
    "container-runtime/internal/types"
    "container-runtime/internal/namespace"
    "container-runtime/internal/cgroup"
    "container-runtime/internal/filesystem"
    "container-runtime/internal/network"
)

// Manager orchestrates container lifecycle operations across all subsystems

type Manager struct {

    containers    map[string]*types.ContainerState
    nsHandler     *namespace.Handler
    cgroupCtrl    *cgroup.Controller
    fsManager     *filesystem.Manager
    netManager    *network.Manager
    configDir     string
    stateDir      string
}

// NewManager creates a new container runtime manager

func NewManager(configDir, stateDir string) (*Manager, error) {
    // TODO: Initialize all component managers
    // TODO: Create necessary directories
    // TODO: Load existing container state from persistence
    // TODO: Perform startup recovery for orphaned resources
}
```

```
    return nil, nil

}

// CreateContainer prepares a new container environment without starting it

func (m *Manager) CreateContainer(spec *types.ContainerSpec) error {

    // TODO: Validate container specification

    // TODO: Check for container ID conflicts

    // TODO: Create container state entry

    // TODO: Prepare image layers through filesystem manager

    // TODO: Create and configure cgroup through cgroup controller

    // TODO: Create namespaces through namespace handler

    // TODO: Set up network namespace through network manager

    // TODO: Mount overlayfs through filesystem manager

    // TODO: Persist container state

    // TODO: Handle errors by cleaning up partial resources

    return nil

}

// StartContainer begins execution of a created container

func (m *Manager) StartContainer(id string) error {

    // TODO: Validate container exists and is in created state

    // TODO: Fork container process in prepared namespaces

    // TODO: Add process to cgroup

    // TODO: Configure network interfaces

    // TODO: Execute container command

    // TODO: Update container state to running

    // TODO: Persist state change

    return nil
```

```
}

// StopContainer gracefully terminates a running container

func (m *Manager) StopContainer(id string) error {

    // TODO: Validate container exists and is running

    // TODO: Send SIGTERM to container process

    // TODO: Wait for graceful shutdown with timeout

    // TODO: Send SIGKILL if timeout exceeded

    // TODO: Update container state to stopped

    // TODO: Persist state change

    return nil
}

// RemoveContainer cleans up all resources for a stopped container

func (m *Manager) RemoveContainer(id string) error {

    // TODO: Validate container exists and is stopped

    // TODO: Unmount overlayfs through filesystem manager

    // TODO: Clean up network configuration through network manager

    // TODO: Remove cgroup through cgroup controller

    // TODO: Clean up namespace resources

    // TODO: Remove container state entry

    // TODO: Clean up persistent state

    return nil
}

// LoadContainerSpec loads container specification from YAML file

func (m *Manager) LoadContainerSpec(path string) (*types.ContainerSpec, error) {

    // TODO: Read YAML file
```

```
// TODO: Parse into ContainerSpec struct  
  
// TODO: Validate required fields  
  
// TODO: Set default values for optional fields  
  
// TODO: Return parsed and validated spec  
  
return nil, nil  
  
}
```

Component Interface Contracts

Each component must implement specific interfaces that the Manager uses for orchestration. This ensures clean separation of concerns while maintaining coordinated behavior:

GO

```
// Component interfaces that each subsystem must implement

type NamespaceHandler interface {

    CreateNamespaces(containerID string, flags NamespaceFlags) error

    EnterNamespaces(containerID string) error

    CleanupNamespaces(containerID string) error

}

type CgroupController interface {

    CreateCgroup(containerID string, limits types.ResourceLimits) error

    AddProcess(containerID string, pid int) error

    GetUsage(containerID string) (*ResourceUsage, error)

    RemoveCgroup(containerID string) error

}

type FilesystemManager interface {

    PrepareContainer(containerID, imageID string) error

    MountOverlayFS(containerID string, layers []string, target string) error

    cleanupContainer(containerID string) error

}

type NetworkManager interface {

    CreateContainerNetwork(containerID string, config types.NetworkConfig) error

    ConnectContainer(containerID string) error

    DisconnectContainer(containerID string) error

    CleanupNetwork(containerID string) error

}
```

Milestone Checkpoints

Checkpoint 1: Architecture Setup After implementing the basic file structure and Manager skeleton:

- Run `go mod tidy && go build ./cmd/runtime/` - should compile without errors
- Create a simple container spec YAML file in `configs/test.yaml`
- Run `go test ./internal/types/` - basic type validation should pass
- The Manager should be able to load container specs from YAML files

Checkpoint 2: Component Integration After implementing component interfaces:

- Each component package should compile independently
- Run `go test ./internal/manager/` - Manager should instantiate all components
- Mock implementations of component interfaces should allow testing Manager orchestration logic
- The Manager should be able to coordinate component method calls in proper sequence

Checkpoint 3: Error Recovery After implementing cleanup and error handling:

- Test partial failure scenarios - create resources with some components, then trigger failures
- Verify that Manager properly cleans up resources from successful components
- Check that the system reaches consistent state after any failure
- No resource leaks should remain after failed container creation attempts

This architecture provides the foundation for implementing each milestone's specific functionality while maintaining clean component boundaries and coordinated behavior across the entire container runtime system.

Data Model

Milestone(s): All milestones (1-4) rely on this data model. The container specification structures support milestone 1 (namespace configuration), milestone 2 (resource limits), milestone 3 (image and filesystem configuration), and milestone 4 (network configuration). The runtime state structures track the lifecycle and cleanup resources across all milestones.

The data model forms the foundation of our container runtime, defining how we represent container configurations, track runtime state, and manage the complex lifecycle of isolated processes. Think of the data model as the **architectural blueprints** for our container system - just as building blueprints specify every room, door, and utility connection before construction begins, our data structures define every piece of information needed to create, manage, and clean up containers before any code executes.

The data model serves two critical purposes in container runtime architecture. First, it provides a **declarative specification interface** where users describe what they want (run this command, with these resource limits, using this image) without needing to understand the complex orchestration of namespaces, cgroups, and filesystem operations underneath. Second, it maintains **runtime state tracking** that allows our system to properly manage the lifecycle of containers, ensuring clean startup sequences and comprehensive cleanup when containers terminate.

Container runtimes must juggle multiple kernel subsystems simultaneously - Linux namespaces for isolation, cgroups for resource control, overlayfs for layered filesystems, and network interfaces for connectivity. Each subsystem has its own configuration parameters, runtime handles, and cleanup requirements. Our data model abstracts this complexity into coherent structures that represent the user's intent and the system's current state.

Container Specification

The `ContainerSpec` structure defines the **declarative blueprint** for what a container should look like when running. This represents the user's intent - the desired end state - rather than the implementation details of how to achieve that state. Think of it as a **recipe card** that specifies all ingredients (image, resources, network settings) and basic instructions (command to run, environment variables) needed to create a container, while leaving the actual cooking process (namespace creation, cgroup setup, filesystem mounting) to the runtime implementation.

The container specification follows the principle of **immutable configuration** - once created, a spec should not change during the container's lifetime. This immutability simplifies reasoning about container behavior and enables reliable cleanup, since the runtime can always refer back to the original specification to understand what resources were allocated and need to be released.

Decision: Separate Specification from Runtime State

- **Context:** Container management requires both user intent (what should run) and current system state (what is actually running)
- **Options Considered:**
 1. Single structure mixing spec and state
 2. Separate spec and state structures
 3. Inheritance hierarchy with base container class
- **Decision:** Use separate `ContainerSpec` and `ContainerState` structures
- **Rationale:** Clear separation of concerns allows immutable specifications while enabling mutable state tracking. Simplifies serialization for persistence and makes the API clearer for users.
- **Consequences:** Requires careful coordination between spec and state, but provides better encapsulation and testing isolation.

Field Name	Type	Description
ID	string	Unique identifier for this container instance. Must be unique across all containers managed by this runtime. Used for filesystem paths, cgroup names, and cleanup tracking.
Image	string	Path to the container image directory or identifier. For our minimal runtime, this points to a directory containing the root filesystem that will become the container's root.
Command	[]string	Command and arguments to execute inside the container. First element is the executable path (relative to container root), remaining elements are arguments passed to the process.
WorkingDir	string	Working directory path inside the container where the command will be executed. Path is relative to the container's root filesystem. Defaults to "/" if not specified.
Environment	map[string]string	Environment variables to set in the container process. Maps variable names to their values. These supplement (and can override) any environment variables from the base image.
Resources	ResourceLimits	Resource constraints to apply to this container via cgroups. Defines memory limits, CPU quotas, and process limits that constrain container resource usage.
Network	NetworkConfig	Network configuration specifying how this container connects to networks and exposes services. Includes port mappings and DNS settings.

The `ID` field serves as the primary key for all container operations and becomes embedded in filesystem paths, cgroup hierarchies, and network interface names. Container IDs must be valid as directory names (no special characters that would break filesystem operations) and should be reasonably short to avoid hitting path length limits when combined with deep directory structures.

The `Image` field in our minimal implementation points to a local directory containing a complete root filesystem. Production container runtimes would extend this to support image references, layer manifests, and remote registries, but for learning purposes we focus on local filesystem images that can be easily inspected and modified during development.

The `Command` array follows POSIX conventions where the first element is the executable and subsequent elements are individual arguments. This differs from shell command lines - arguments containing spaces must be separate array elements rather than quoted strings. The runtime does not perform shell expansion or interpretation of the command elements.

Resource Limits Structure

Resource limits define the **resource budget** allocated to a container, implemented through Linux cgroups. Think of resource limits as setting a **spending allowance** for different types of system resources - just as a parent might give a child separate allowances for food, entertainment, and savings, cgroups let us set separate limits for memory consumption, CPU usage, and process creation.

Decision: Explicit Resource Limit Fields vs Generic Map

- **Context:** Need to represent different types of resource limits that will be enforced via cgroups
- **Options Considered:**
 1. Typed fields for each resource type (Memory, CPU, PIDs)
 2. Generic map[string]interface{} for flexibility
 3. Nested structures per cgroup controller
- **Decision:** Use explicit typed fields for core resource types
- **Rationale:** Type safety prevents configuration errors, clear API makes resource limits obvious, easier validation and documentation
- **Consequences:** Adding new resource types requires schema changes, but provides better user experience and prevents runtime errors

Field Name	Type	Description
Memory	int64	Maximum memory usage in bytes. Enforced via cgroup memory.max. Container processes will be killed if they exceed this limit. Zero means no limit.
CPU	int64	CPU quota in microseconds per 100ms period. Enforced via cgroup cpu.max. Value of 50000 means container gets 50% of one CPU core. Zero means no limit.
PIDs	int64	Maximum number of processes/threads this container can create. Enforced via cgroup pids.max. Prevents fork bombs and runaway process creation. Zero means no limit.

The memory limit directly maps to the cgroups v2 `memory.max` file, which enforces a hard limit on memory usage. When a container approaches this limit, the kernel will start killing processes within the container's cgroup, typically starting with the most recently started processes. This provides strong isolation but can cause abrupt application failures if limits are set too low.

CPU limits use the cgroups v2 bandwidth control system, which implements a quota over a fixed period. The period is typically 100 milliseconds (100000 microseconds), and the quota specifies how many microseconds of CPU time the container can use within each period. A quota of 50000 microseconds in a 100000 microsecond period means the container gets 50% of one CPU core.

Process limits prevent containers from exhausting the system's process table through fork bombs or poorly written applications that leak processes. The limit applies to the total number of tasks (processes and threads) within the container's cgroup, providing protection against both accidental and malicious process creation.

Network Configuration Structure

Network configuration defines how a container connects to networks and exposes services to the outside world. Think of network configuration as specifying the **mailing address and phone system** for your container - it determines how other containers and external systems can reach services running inside the container, and how the container reaches external resources.

Field Name	Type	Description
PortMappings	[]PortMapping	List of port forwarding rules that expose container services on host ports. Each mapping creates NAT rules forwarding traffic from a host port to a container port.
DNS	[]string	DNS nameserver IP addresses for container name resolution. These servers will be configured in the container's /etc/resolv.conf file for hostname lookups.

The port mappings implement **Network Address Translation (NAT)** using iptables rules that forward traffic from host network interfaces to the container's network namespace. This allows external clients to connect to services running inside containers by connecting to host ports, with the traffic automatically forwarded to the appropriate container ports.

DNS configuration ensures that processes inside containers can resolve hostnames to IP addresses. Since containers run in isolated network namespaces, they cannot automatically inherit the host system's DNS configuration and must have explicit nameserver configuration.

Port Mapping Structure

Port mappings define individual **traffic forwarding rules** that expose container services on the host network. Each port mapping creates a tunnel through the network isolation boundary, allowing external traffic to reach specific services inside containers.

Field Name	Type	Description
HostPort	int	Port number on the host where traffic will be received. Must be available (not already bound by another service). Range 1-65535.
ContainerPort	int	Port number inside the container where the service is listening. This is the destination for forwarded traffic. Range 1-65535.
Protocol	string	Network protocol for this port mapping. Typically "tcp" or "udp". Determines which iptables rules and traffic types this mapping handles.

Port mappings create a **bidirectional tunnel** through the network namespace isolation. Inbound traffic to the host port gets forwarded to the container port, and response traffic from the container port gets forwarded back to the original external client. This requires careful iptables rule management to ensure traffic flows correctly in both directions.

The protocol field determines which type of network traffic this mapping handles. TCP mappings create connection-oriented forwarding suitable for HTTP services, databases, and other stream-based protocols. UDP mappings create connectionless forwarding suitable for DNS, DHCP, and other packet-based protocols.

Runtime State

The `ContainerState` structure tracks the **current operational status** of a running container, maintaining all the runtime handles, resource paths, and metadata needed to manage and clean up container instances. Think of runtime state as the **active case file** for a container - it contains all the information needed to monitor, interact with, and eventually clean up a running container instance.

Runtime state is **mutable and ephemeral** - it changes as containers start, run, and stop, and it does not need to persist across system restarts (since containers don't survive reboots in our minimal implementation). The state structure serves as a central registry of all system resources allocated to a container, enabling systematic cleanup when containers terminate.

The runtime state maintains **resource ownership tracking** - for every system resource allocated during container creation (cgroup directories, overlay mounts, network interfaces), the state structure records the identifiers and paths needed to clean up those resources. This prevents resource leaks and ensures containers can be completely removed from the system.

Field Name	Type	Description
Spec	ContainerSpec	The original container specification that created this instance. Provides immutable reference to user intent and configuration parameters.
Status	ContainerStatus	Current lifecycle status of this container (Created, Running, Stopped, etc.). Determines which operations are valid and what cleanup is required.
PID	int	Process ID of the main container process in the host PID namespace. Used for process management and cleanup. Zero if container is not running.
CreatedAt	time.Time	Timestamp when container state was first created. Used for logging, monitoring, and container age calculations.
StartedAt	*time.Time	Timestamp when container process was started, or nil if never started. Pointer allows distinction between "never started" and "started at Unix epoch".
CgroupPath	string	Full filesystem path to this container's cgroup directory (e.g., "/sys/fs/cgroup/container-123"). Used for resource limit enforcement and cleanup.
OverlayPath	string	Full filesystem path to the overlay mount point for this container's root filesystem. Must be unmounted during cleanup to prevent resource leaks.
NetworkNS	string	Path to the network namespace file in /proc (e.g., "/proc/123/ns/net"). Used for network operations and cleanup.
VethHost	string	Name of the veth interface in the host network namespace (e.g., "veth123h"). Must be deleted during cleanup.
VethContainer	string	Name of the veth interface inside the container network namespace (e.g., "eth0"). Automatically cleaned up when namespace is destroyed.
IPAddress	string	IP address assigned to this container's network interface. Used for connectivity and must be returned to the address pool during cleanup.

The embedded `Spec` field provides **configuration consistency** - the runtime can always refer back to the original user intent when making decisions about resource management, cleanup, or status reporting. This embedded spec should never be modified after container creation, ensuring that the original configuration remains available throughout the container lifecycle.

The `PID` field represents the container's main process as seen from the host perspective. Even though the process sees itself as PID 1 inside its PID namespace, the host system assigns it a regular PID that we use for process management operations like sending signals or checking process status.

Path-based fields (`CgroupPath`, `OverlayPath`, `NetworkNS`) provide the **cleanup roadmap** - they contain all the filesystem paths that must be unmounted, removed, or cleaned up when the container terminates.

These paths are determined during container creation and remain stable throughout the container's lifetime.

Network-related fields (`VethHost`, `VethContainer`, `IPAddress`) track the **network plumbing** created for this container. The host veth interface must be explicitly deleted during cleanup, while the container veth interface is automatically cleaned up when the network namespace is destroyed. The IP address must be returned to the available address pool so it can be reused by future containers.

Container Status Enumeration

Container status represents the current **lifecycle stage** of a container instance, determining which operations are valid and what state transitions are possible. This follows a simple state machine that mirrors the typical container lifecycle from creation through cleanup.

Status Value	Description	Valid Transitions	Cleanup Required
Created	Container resources allocated but process not started	→ Running, → Stopped	Full cleanup (cgroups, filesystem, network)
Running	Container process actively executing	→ Stopped	Process termination + full cleanup
Stopped	Container process terminated but resources not cleaned	→ Removed	Full cleanup (cgroups, filesystem, network)
Removed	Container completely cleaned up and removed	None (terminal state)	None

The status progression typically follows: Created → Running → Stopped → Removed, though containers can transition directly from Created to Stopped if they fail to start, or from Running to Removed if they are forcibly deleted while running.

Common Data Model Pitfalls

⚠ Pitfall: Mixing Configuration with Runtime State Many developers initially put runtime-specific fields like PIDs and paths directly into the container specification structure. This breaks the separation between user intent (what should run) and system state (what is actually running). Keep the `ContainerSpec` immutable and put all runtime tracking in `ContainerState`.

⚠ Pitfall: Forgetting Resource Cleanup Tracking The runtime state must track every system resource allocated during container creation. Forgetting to record filesystem paths, network interface names, or cgroup paths leads to resource leaks when containers are removed. Always add cleanup tracking fields to `ContainerState` when introducing new resource types.

⚠ Pitfall: Using Host-Relative Paths in Container Specs Container specifications should use container-relative paths for working directories and command paths, not host system paths. The `WorkingDir` field

should be "/app" not "/var/lib/containers/container-123/rootfs/app" - the runtime handles the translation to host paths during execution.

⚠ Pitfall: Insufficient Container ID Validation Container IDs become embedded in filesystem paths and network interface names, so they must be validated to ensure they don't contain characters that would break these systems. Reject IDs containing "/", ".", "..", or other special characters that could cause path traversal or naming conflicts.

Implementation Guidance

The data model implementation requires careful attention to serialization, validation, and lifecycle management. The structures serve as both API contracts and internal state management, so they need to be robust against invalid input while remaining easy to work with programmatically.

Technology Recommendations

Component	Simple Option	Advanced Option
Serialization	JSON with encoding/json package	Protocol Buffers with schema validation
Validation	Manual field checks in constructors	Struct tags with validator library
State Persistence	In-memory maps with JSON files	Embedded database like BoltDB
ID Generation	UUID v4 with google/uuid	Collision-resistant with timestamp + random

Recommended File Structure

```
internal/
  types/
    container.go      ← Core data structures
    container_test.go ← Structure validation tests
    validation.go     ← Input validation functions
    validation_test.go ← Validation test cases
  runtime/
    state_manager.go  ← Container state tracking
    state_manager_test.go ← State management tests
```

Core Data Structures

```
package types
```

import (
 "fmt"
 "net"
 "path/filepath"
 "regexp"
 "strings"
 "time"
)

// ContainerSpec defines the desired configuration for a container instance.
// This structure is immutable after creation and represents user intent.

```
type ContainerSpec struct {  
  
    // ID uniquely identifies this container instance  
    ID string `json:"id"  
  
    // Image specifies the root filesystem for the container  
    Image string `json:"image"  
  
    // Command and arguments to execute in the container  
    Command []string `json:"command"  
  
    // WorkingDir sets the initial working directory (container-relative path)  
    WorkingDir string `json:"working_dir,omitempty"  
  
    // Environment variables for the container process
```

GO

```
Environment map[string]string `json:"environment,omitempty"`

// Resource limits enforced via cgroups

Resources ResourceLimits `json:"resources,omitempty"`

// Network configuration for connectivity and port exposure

Network NetworkConfig `json:"network,omitempty"`

}

// ResourceLimits defines cgroup-enforced resource constraints

type ResourceLimits struct {

    // Memory limit in bytes (0 = no limit)

    Memory int64 `json:"memory,omitempty"`

    // CPU quota in microseconds per 100ms period (0 = no limit)

    CPU int64 `json:"cpu,omitempty"`

    // Maximum number of processes/threads (0 = no limit)

    PIDs int64 `json:"pids,omitempty"`

}

// NetworkConfig specifies container networking requirements

type NetworkConfig struct {

    // Port mappings for service exposure

    PortMappings []PortMapping `json:"port_mappings,omitempty"`

    // DNS nameservers for hostname resolution

    DNS []string `json:"dns,omitempty"`

}
```

```
}

// PortMapping defines a single port forwarding rule

type PortMapping struct {

    // Host port to listen on

    HostPort int `json:"host_port"`

    // Container port to forward to

    ContainerPort int `json:"container_port"`

    // Protocol ("tcp" or "udp")

    Protocol string `json:"protocol"`

}

// ContainerState tracks the runtime status of a container instance

type ContainerState struct {

    // Original specification (immutable reference)

    Spec ContainerSpec `json:"spec"`

    // Current lifecycle status

    Status ContainerStatus `json:"status"`

    // Main process ID in host namespace (0 if not running)

    PID int `json:"pid"`

    // Creation and start timestamps

    CreatedAt time.Time `json:"created_at"`

    StartedAt *time.Time `json:"started_at,omitempty"`

}
```

```
// Resource cleanup tracking

CgroupPath  string `json:"cgroup_path,omitempty"`
OverlayPath string `json:"overlay_path,omitempty"`
NetworkNS   string `json:"network_ns,omitempty"`
VethHost    string `json:"veth_host,omitempty"`
VethContainer string `json:"veth_container,omitempty"`
IPAddress   string `json:"ip_address,omitempty"`

}

// ContainerStatus represents container lifecycle states

type ContainerStatus string

const (
    ContainerCreated ContainerStatus = "created"
    ContainerRunning ContainerStatus = "running"
    ContainerStopped ContainerStatus = "stopped"
    ContainerRemoved ContainerStatus = "removed"
)
```

Validation Functions

```
// ValidateContainerSpec checks a container specification for common errors GO

func ValidateContainerSpec(spec *ContainerSpec) error {

    // TODO 1: Validate container ID format (alphanumeric, hyphens, underscores only)

    // TODO 2: Check that Image path exists and is accessible

    // TODO 3: Validate Command has at least one element and first element is not empty

    // TODO 4: Validate WorkingDir is absolute path starting with "/"

    // TODO 5: Check Environment variable names don't contain "=" character

    // TODO 6: Validate ResourceLimits are non-negative

    // TODO 7: Validate PortMappings have valid port ranges (1-65535)

    // TODO 8: Check DNS entries are valid IP addresses

    return nil

}

// ValidateContainerID ensures ID can safely be used in filesystem paths

func ValidateContainerID(id string) error {

    // TODO 1: Check ID length is between 1 and 64 characters

    // TODO 2: Verify ID matches pattern: alphanumeric, hyphens, underscores only

    // TODO 3: Ensure ID doesn't start or end with hyphen or underscore

    // TODO 4: Reject reserved names like ".", "..", or system directory names

    return nil

}

// ValidatePortMapping checks a single port mapping for validity

func ValidatePortMapping(pm PortMapping) error {

    // TODO 1: Validate HostPort is in range 1-65535

    // TODO 2: Validate ContainerPort is in range 1-65535

    // TODO 3: Check Protocol is either "tcp" or "udp"
```

```
// TODO 4: Verify HostPort is not in reserved range (1-1023) unless privileged  
  
return nil  
  
}
```

State Management Helper

```
package runtime GO

// StateManager handles container state persistence and lifecycle tracking

type StateManager struct {

    // TODO: Add fields for state storage (map, file path, etc.)

}

// SaveContainerState persists container state to storage

func (sm *StateManager) SaveContainerState(state *types.ContainerState) error {

    // TODO 1: Serialize state to JSON

    // TODO 2: Write to persistent storage (file, database, etc.)

    // TODO 3: Ensure atomic write operation

    // TODO 4: Handle storage errors gracefully

    return nil

}

// LoadContainerState retrieves container state from storage

func (sm *StateManager) LoadContainerState(id string) (*types.ContainerState, error) {

    // TODO 1: Read state data from storage

    // TODO 2: Deserialize JSON to ContainerState struct

    // TODO 3: Validate loaded data integrity

    // TODO 4: Return appropriate error if container not found

    return nil, nil

}

// ListContainers returns all container states matching optional filter

func (sm *StateManager) ListContainers(status types.ContainerStatus)
([]*types.ContainerState, error) {
```

```

    // TODO 1: Load all container states from storage

    // TODO 2: Filter by status if specified (empty status = return all)

    // TODO 3: Sort results by creation time

    // TODO 4: Return slice of matching containers

    return nil, nil

}

// DeleteContainerState removes container state from storage

func (sm *StateManager) DeleteContainerState(id string) error {

    // TODO 1: Verify container exists in storage

    // TODO 2: Remove state data from persistent storage

    // TODO 3: Clean up any related temporary files

    // TODO 4: Ensure deletion is atomic

    return nil

}

```

Language-Specific Implementation Notes

Go-specific considerations:

- Use `encoding/json` struct tags for API serialization control
- Implement `String()` methods on enums like `ContainerStatus` for logging
- Use pointer types (`*time.Time`) to distinguish between zero values and unset fields
- Consider using `sync.RWMutex` for concurrent access to state maps
- Validate file paths using `filepath.Clean()` to prevent directory traversal

Validation patterns:

- Use regular expressions for ID format validation: `^[a-zA-Z0-9]([a-zA-Z0-9_-]*[a-zA-Z0-9])?$/`
- Check port ranges with simple numeric comparisons: `port >= 1 && port <= 65535`
- Validate IP addresses using `net.ParseIP()` from standard library
- Use `os.Stat()` to verify image directory existence and permissions

Milestone Checkpoint

After implementing the data model structures:

1. **Run unit tests:** `go test ./internal/types/...` should pass all validation tests
2. **Test JSON serialization:** Create a container spec, marshal to JSON, unmarshal back, verify equality
3. **Validate error handling:** Try invalid container IDs, port ranges, and paths - should get clear error messages
4. **Check state transitions:** Verify container status can only transition through valid states

Expected behavior:

- Container specs with valid fields serialize/deserialize correctly
- Invalid container IDs (with special characters) are rejected with descriptive errors
- Port mappings outside 1-65535 range are rejected
- Resource limits accept zero (unlimited) and positive values, reject negative values

Signs of problems:

- JSON marshaling panics → Check for circular references in embedded structs
- Validation allows invalid input → Add more comprehensive test cases
- File path operations fail → Ensure proper path cleaning and validation

Namespace Isolation Component

Milestone(s): Milestone 1 (Process Isolation with Namespaces) - This section implements the core isolation mechanisms that allow containers to have their own view of system resources while running on a shared kernel.

Mental Model: Separate Worlds

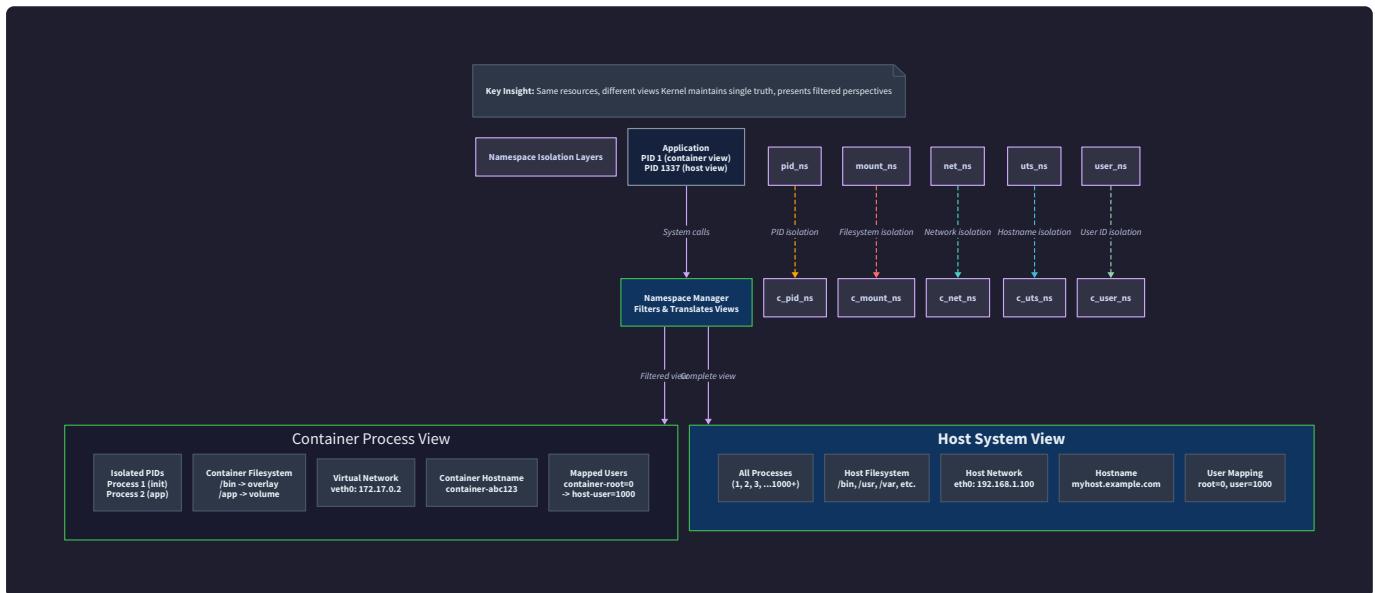
Think of Linux namespaces as creating parallel universes for processes. Imagine you have a magical building where each apartment exists in its own dimension. From inside apartment 3B, the residents can only see their own rooms, their own mailbox labeled "1" (even though it's really mailbox 3B from the outside), and their own view out the windows. They have no idea that apartments 3A and 3C even exist - they think they're the only residents in the entire building.

This is exactly how namespaces work for container processes. When a process runs inside a container, it lives in its own isolated world where:

- It thinks it's the only process running (PID namespace) and sees itself as process ID 1
- It has its own private filesystem that looks like a complete Linux system (mount namespace)
- It has its own network interfaces and IP addresses (network namespace)
- It can set its own hostname without affecting other containers (UTS namespace)
- It maps its user IDs differently from the host system (user namespace)

The crucial insight is that these parallel worlds are created by the Linux kernel, not by copying actual resources. The kernel maintains a single set of processes, files, and network interfaces, but it presents different views of these resources to processes in different namespaces. When a container process asks "what processes are running?", the kernel filters the answer to only show processes in the same PID namespace. When it asks "what files exist in /bin?", the kernel shows files from the container's mount namespace, not the host's /bin directory.

This isolation is both powerful and lightweight. Unlike virtual machines that duplicate entire operating systems, namespaces share the same kernel while providing complete isolation of the user-space view. The container process genuinely believes it's running on its own dedicated system, even though dozens of other containers might be running on the same physical machine.



Namespace Types and Setup

Linux provides several namespace types, each isolating a different aspect of the system. Our container runtime needs to orchestrate the creation and configuration of multiple namespace types to provide complete process isolation.

Decision: Namespace Combination Strategy

- **Context:** Containers need isolation across multiple system resources (processes, filesystems, network, hostname, IPC). We must decide whether to create all namespaces together or incrementally.
- **Options Considered:**
 1. Create all namespaces atomically in a single system call
 2. Create namespaces incrementally as needed
 3. Create base namespaces first, then specialize
- **Decision:** Create all required namespaces atomically using a single `clone()` or `unshare()` call with combined flags
- **Rationale:** Atomic creation prevents race conditions where a process exists in some namespaces but not others. It also simplifies error handling - either all namespaces are created successfully, or the operation fails completely.
- **Consequences:** Simplifies the creation logic and eliminates partial failure states, but requires careful flag combination and may be less flexible for specialized container types.

PID Namespace Isolation

The **PID namespace** creates an isolated process tree where the container's first process becomes PID 1. This is fundamental to container isolation because it prevents containers from seeing or signaling processes on the host system.

Aspect	Description	Implementation Detail
Isolation Scope	Process visibility and PID assignment	Container processes only see processes within their PID namespace
Creation Flag	<code>CLONE_NEWPID</code>	Used with <code>clone()</code> , <code>unshare()</code> , or <code>setns()</code> system calls
First Process	Becomes PID 1 in the new namespace	Critical for proper init semantics and signal handling
Process Tree	Complete isolation from host process tree	<code>/proc</code> shows only namespace-local processes
Signal Delivery	Signals cannot cross namespace boundaries	Prevents container processes from killing host processes
Cleanup Behavior	Namespace destroyed when last process exits	Automatic cleanup prevents resource leaks

When we create a PID namespace, the first process spawned inside it becomes PID 1 from the container's perspective, even though it has a different PID on the host system. This PID 1 process has special responsibilities - it must reap zombie child processes and handle signals properly. If PID 1 dies, the entire namespace is destroyed and all processes within it are killed.

The namespace creation sequence follows this algorithm:

1. The runtime manager calls `clone()` with `CLONE_NEWPID` flag to create a new PID namespace
2. The kernel creates a new PID namespace and assigns the cloned process PID 1 within that namespace
3. The process mounts a new `/proc` filesystem to show only namespace-local processes
4. Any child processes spawned within the namespace receive PIDs 2, 3, etc. from the container's perspective
5. The namespace persists until the last process within it exits

Mount Namespace Isolation

The **mount namespace** provides each container with its own filesystem view, allowing containers to have different root filesystems, mount points, and filesystem hierarchies without affecting the host or other containers.

Aspect	Description	Implementation Detail
Isolation Scope	Filesystem mounts and hierarchy	Each namespace has independent mount table
Creation Flag	<code>CLONE_NEWNS</code>	First namespace type introduced in Linux
Root Filesystem	Container sees its own root directory	Achieved through <code>pivot_root()</code> or <code>chroot()</code>
Mount Propagation	Controls how mounts spread between namespaces	Can be shared, slave, or private
Filesystem View	Complete isolation of filesystem tree	Container <code>/bin</code> different from host <code>/bin</code>
Cleanup Behavior	Mounts automatically unmounted when namespace dies	Prevents mount point leaks

Mount namespaces are particularly complex because they interact with the overlayfs layer management system. The runtime must coordinate between creating the mount namespace, preparing the overlay filesystem layers, and then switching the container's root filesystem to point to the merged overlay view.

The mount namespace setup algorithm proceeds as follows:

1. Create the mount namespace using `CLONE_NEWNS` flag during process creation
2. Prepare the overlay filesystem by mounting the image layers (handled by the filesystem component)

3. Create mount points for essential filesystems (`/proc` , `/sys` , `/dev` , `/tmp`)
4. Execute `pivot_root()` to switch from the host root to the container root
5. Mount essential pseudo-filesystems within the new root
6. Unmount the old root filesystem to complete the isolation

Network Namespace Isolation

The **network namespace** gives each container its own network stack, including network interfaces, IP addresses, routing tables, and firewall rules. This enables containers to have conflicting network configurations without interference.

Aspect	Description	Implementation Detail
Isolation Scope	Complete network stack isolation	Interfaces, routes, iptables rules, sockets
Creation Flag	<code>CLONE_NEWNET</code>	Creates empty network namespace initially
Initial State	Only loopback interface present	All other interfaces must be added explicitly
Interface Assignment	Physical or virtual interfaces moved into namespace	Typically uses veth pairs
IP Configuration	Independent IP addresses and routing	Configured via netlink or ip command
Connectivity	Requires explicit bridge/routing setup	Handled by network management component

A new network namespace starts completely empty except for the loopback interface. This means containers have no network connectivity until we explicitly configure interfaces, IP addresses, and routing. The network management component handles creating veth pairs and connecting them to bridge networks.

Network namespace setup follows this sequence:

1. Create network namespace with `CLONE_NEWNET` during container process creation
2. Network management component creates a veth pair (virtual ethernet cable)
3. One end of veth pair moved into container namespace, other end remains on host
4. Container end configured with IP address from allocated subnet
5. Host end connected to bridge network for inter-container communication
6. Routing rules configured to enable container network access

UTS Namespace Isolation

The **UTS namespace** (UNIX Time Sharing) isolates hostname and domain name, allowing each container to have its own system identity without affecting the host or other containers.

Aspect	Description	Implementation Detail
Isolation Scope	Hostname and NIS domain name	System calls like <code>gethostname()</code> return namespace-local values
Creation Flag	<code>CLONE_NEWUTS</code>	Lightweight namespace with minimal overhead
Initial State	Inherits hostname from parent namespace	Can be changed independently afterward
Configuration	Set via <code>sethostname()</code> system call	Typically set from container specification
Use Cases	Service discovery, logging, application configuration	Many applications use hostname for identification
Cleanup	Automatic when namespace destroyed	No persistent state to clean up

UTS namespaces are relatively simple but important for application compatibility. Many applications expect to run on systems with specific hostnames, and service discovery systems often use hostnames for routing and identification.

User Namespace Isolation

The **user namespace** maps user and group IDs between the container and host system, enabling privilege separation and allowing containers to run as root inside the namespace while being unprivileged on the host.

Aspect	Description	Implementation Detail
Isolation Scope	User ID and group ID mappings	UID 0 inside container may be UID 1000 on host
Creation Flag	<code>CLONE_NEWUSER</code>	Most complex namespace to configure correctly
UID Mapping	Maps container UIDs to host UIDs	Written to <code>/proc/[pid]/uid_map</code>
GID Mapping	Maps container GIDs to host GIDs	Written to <code>/proc/[pid]/gid_map</code>
Capabilities	Grants capabilities within the namespace	Container root has limited capabilities on host
Security Benefit	Unprivileged containers possible	Container breakout results in unprivileged host access

User namespaces are crucial for security but complex to configure. They require setting up UID and GID mappings that define how user identities translate between the container and host perspectives.

Architecture Decision: User Namespace Configuration

- **Context:** User namespaces require complex UID/GID mapping setup and have implications for file permissions and capabilities
- **Options Considered:**
 1. Always enable user namespaces for maximum security
 2. Make user namespaces optional with configuration flag
 3. Skip user namespaces to simplify implementation
- **Decision:** Make user namespaces optional but provide clear configuration interface
- **Rationale:** User namespaces significantly improve security but add complexity and may cause permission issues with mounted volumes. Making them optional allows users to choose the right security/complexity tradeoff.
- **Consequences:** More secure when enabled but requires careful UID/GID mapping configuration. File ownership issues may arise when sharing files between host and container.

Namespace Creation Implementation

The namespace creation process requires careful coordination of multiple system calls and proper error handling to ensure consistent state.

Step	Action	System Call	Error Handling
1	Prepare namespace flags	Bitwise OR of <code>CLONE_NEW*</code> constants	Validate flag combinations
2	Create namespaces	<code>clone()</code> or <code>unshare()</code>	Clean up partial namespaces on failure
3	Configure user mappings	Write to <code>/proc/[pid]/uid_map</code>	Verify mapping is valid and applied
4	Set up network interfaces	<code>netlink</code> socket operations	Remove interfaces on failure
5	Mount essential filesystems	<code>mount()</code> system calls	Unmount on failure
6	Execute <code>pivot_root</code>	<code>pivot_root()</code> system call	Restore original root on failure

The implementation must handle partial failures gracefully. If namespace creation succeeds but subsequent configuration fails, we must clean up the created namespaces to avoid resource leaks.

Filesystem Pivot Process

The filesystem pivot process switches the container's root filesystem from the host root to the container's isolated filesystem. This is more complex than a simple `chroot()` because it must handle overlayfs mounts and ensure proper cleanup of the old root.

Decision: pivot_root vs chroot

- **Context:** Need to change container's root filesystem to isolated overlay mount
- **Options Considered:**
 1. Use `chroot()` system call for simplicity
 2. Use `pivot_root()` system call for proper isolation
 3. Use bind mounts to create filesystem view
- **Decision:** Use `pivot_root()` for complete root filesystem replacement
- **Rationale:** `chroot()` is easily escaped by processes with sufficient privileges and doesn't properly isolate the old root. `pivot_root()` provides complete isolation and allows proper cleanup of the old root filesystem.
- **Consequences:** More complex implementation but much better security isolation. Requires careful mount point setup and cleanup procedures.

Pivot Root Algorithm

The pivot root process must be executed in precise order to avoid mount point conflicts and ensure proper isolation:

1. **Verify mount namespace isolation:** Confirm the container process is running in its own mount namespace, isolated from the host mount table. This prevents `pivot_root` operations from affecting the host filesystem.
2. **Prepare overlay filesystem mount:** The filesystem management component must have already created the overlay mount combining image layers. Verify this mount exists and is accessible at the expected path (typically `/var/lib/container-runtime/overlay/[container-id]/merged`).
3. **Create old root mount point:** Inside the new root filesystem, create a directory that will hold the old root temporarily. This is typically `/old-root` or `/mnt/old-root`. This directory must exist in the overlay filesystem before `pivot_root`.
4. **Execute pivot_root system call:** Call `pivot_root(new_root, old_root)` where `new_root` is the overlay merged directory and `old_root` is the temporary mount point created in step 3. This atomically swaps the root filesystem.
5. **Change working directory:** Immediately after `pivot_root`, change the current working directory to `/` in the new root to ensure no processes are holding references to the old root filesystem.

6. **Mount essential pseudo-filesystems:** Mount `/proc`, `/sys`, `/dev`, and `/tmp` within the new root. These are essential for proper container operation and must be mounted after the pivot to ensure they reflect the container's namespace state.
7. **Unmount old root:** Unmount the old root filesystem that is now accessible under the temporary mount point. This completes the isolation and prevents the container from accessing host filesystem paths.
8. **Remove old root mount point:** Delete the temporary directory used to hold the old root, completing the cleanup process.

Mount Point Requirements

The pivot_root operation has strict requirements that commonly cause failures:

Requirement	Explanation	Common Failure
New root must be a mount point	The target directory must be a filesystem mount, not just a directory	Using a regular directory causes <code>EINVAL</code>
Old root must be under new root	The old root mount point must be a subdirectory of the new root	Placing old root outside new root causes <code>EINVAL</code>
Both must be on different filesystems	New root and old root cannot be the same filesystem	Using bind mounts of same filesystem causes <code>EBUSY</code>
No processes using old root	No process can have working directory or open files in old root	Active processes cause <code>EBUSY</code>
Mount namespace must be private	Mount propagation must not affect other namespaces	Shared mounts can cause <code>EINVAL</code>

Essential Filesystem Mounts

After pivot_root completes, the container needs several pseudo-filesystems mounted to function properly:

Mount Point	Filesystem Type	Purpose	Mount Options
/proc	proc	Process information and system interfaces	nosuid, nodev, noexec
/sys	sysfs	System device and driver information	nosuid, nodev, noexec, ro
/dev	tmpfs	Device nodes for container	nosuid, strictatime, mode=755
/dev/pts	devpts	Pseudo-terminal devices	nosuid, noexec, newinstance, ptmxmode=0666
/tmp	tmpfs	Temporary file storage	nosuid, nodev, noexec

The `/proc` mount is particularly critical because many container processes depend on `/proc/self/` for introspection and `/proc/[pid]/` for process management. Without proper `/proc` mounting, tools like `ps`, `top`, and many application health checks will fail.

Common Namespace Pitfalls

Namespace creation and management involves several subtle failure modes that can cause containers to behave incorrectly or fail to start. Understanding these pitfalls helps avoid hours of debugging mysterious container failures.

⚠️ Pitfall: Mount Point Requirements for pivot_root

Problem: The most common failure when implementing `pivot_root` is violating the kernel's strict requirements for the operation. The error message `EINVAL` from `pivot_root` provides no specific information about which requirement was violated.

Specific Failure Scenarios:

- Attempting `pivot_root` where the new root is not a mount point (just a regular directory)
- Creating the old root directory outside the new root filesystem
- Using the same filesystem for both old and new root (common with bind mounts)
- Running `pivot_root` while processes have the old root as their working directory

Why This Breaks: The kernel enforces these requirements to prevent filesystem inconsistencies and ensure proper cleanup. `pivot_root` must atomically swap roots while maintaining filesystem integrity.

Detection: Check return value of `pivot_root` system call. `EINVAL` typically indicates requirement violation, `EBUSY` indicates active processes using the old root.

Fix Implementation:

1. Always ensure new root is a proper mount (overlayfs mount, not bind mount of directory)
2. Create old root directory inside the new root before calling pivot_root
3. Use `fchdir()` to change working directory immediately before pivot_root
4. Verify no child processes are running with old root as working directory

Pitfall: Missing /proc Mount in PID Namespace

Problem: After creating a PID namespace, many tools and applications break because `/proc` still shows the host's process tree instead of the container's isolated process view.

Specific Symptoms:

- `ps aux` shows all host processes instead of only container processes
- Process tools report wrong PID numbers
- Applications that read `/proc/self/` get incorrect information
- Init systems fail to manage processes properly

Why This Breaks: The `/proc` filesystem is not automatically updated when entering a new PID namespace. The old `/proc` mount continues to show the original namespace's process tree. Applications expect `/proc` to reflect the current namespace state.

Detection: After namespace creation, check if `/proc/1/` exists and points to the container's init process, not a host process.

Fix Implementation:

1. Unmount existing `/proc` after entering PID namespace: `umount("/proc")`
2. Mount new proc for the namespace: `mount("proc", "/proc", "proc", 0, NULL)`
3. Verify mount shows correct PID namespace view by checking `/proc/1/` exists
4. Ensure mount happens after pivot_root to get correct namespace perspective

Pitfall: User Namespace UID Mapping Race Condition

Problem: User namespaces start with no UID/GID mappings configured, during which time the process has no privileges and many operations fail with permission errors.

Specific Failure Scenarios:

- Process creation fails immediately after user namespace creation
- File operations fail with `EACCES` even for files the process should access
- Capability checks fail unexpectedly for operations that should succeed
- Race condition where container process starts before UID mapping is written

Why This Breaks: User namespaces begin in a state where no UIDs or GIDs are valid. The kernel rejects most privileged operations until `/proc/[pid]/uid_map` and `/proc/[pid]/gid_map` are properly configured.

Detection: Check if UID mapping files are empty or missing after user namespace creation. Operations failing with `EACCES` immediately after namespace creation indicate mapping issues.

Fix Implementation:

1. Create user namespace but don't execute container command immediately
2. Parent process writes UID mapping to `/proc/[child-pid]/uid_map`
3. Parent process writes GID mapping to `/proc/[child-pid]/gid_map`
4. Use synchronization (pipe or signal) to ensure child waits for mapping completion
5. Child process verifies mappings are applied before proceeding with container setup

⚠ Pitfall: Network Namespace Connectivity Loss

Problem: New network namespaces contain only a loopback interface, leaving containers with no network connectivity unless explicitly configured.

Specific Symptoms:

- Container cannot reach external networks or other containers
- DNS resolution fails completely
- Network tools show only `lo` interface
- Applications fail with "network unreachable" errors

Why This Breaks: Network namespace isolation is complete - containers get an empty network stack. Unlike other namespaces that inherit some functionality, network namespaces require explicit configuration for connectivity.

Detection: Run `ip link show` inside container - should show only loopback interface if networking not configured properly.

Fix Implementation:

1. Create veth pair before container starts: one end for container, one for host
2. Move container veth end into container's network namespace
3. Configure IP address on container's veth interface
4. Connect host veth end to bridge network for routing
5. Verify connectivity with ping test before declaring container ready

⚠ Pitfall: Incomplete Namespace Cleanup on Failure

Problem: When container creation fails partway through namespace setup, partially created namespaces may persist and leak resources or cause conflicts for subsequent containers.

Specific Failure Scenarios:

- Process dies after creating some namespaces but before joining all namespaces
- Mount operations fail leaving orphaned mount points

- Network interfaces created but not properly associated with namespaces
- Cgroup directories created but not cleaned up when namespace creation fails

Why This Breaks: Namespace creation involves multiple system calls that can fail independently. Each successful step allocates kernel resources that must be explicitly cleaned up if later steps fail.

Detection: Check for orphaned mount points in `/proc/mounts`, unused network interfaces with `ip link show`, and leaked cgroup directories under `/sys/fs/cgroup/`.

Fix Implementation:

1. Track all created namespaces and resources in container state
2. Implement rollback logic that undoes successful operations when later operations fail
3. Use defer-style cleanup in implementation language to ensure cleanup runs
4. Test failure scenarios explicitly to verify cleanup works correctly
5. Monitor for resource leaks in integration tests

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Namespace Creation	Direct system calls with <code>golang.org/x/sys/unix</code>	Higher-level library like <code>github.com/opencontainers/runc/libcontainer</code>
Mount Operations	Standard <code>mount()</code> syscalls	<code>github.com/moby/sys/mount</code> for advanced mount handling
Process Management	<code>os.Process</code> with manual PID tracking	Process group management with <code>golang.org/x/sys/unix</code>
Error Handling	Basic error checking and logging	Structured error types with detailed context

Recommended File Structure

The namespace isolation component should be organized as a focused module within the container runtime:

```
internal/namespace/
├── namespace.go          ← main namespace manager interface
├── isolation.go          ← namespace creation and setup logic
├── pivot.go               ← filesystem pivot_root implementation
├── cleanup.go             ← namespace cleanup and resource management
├── types.go               ← namespace-related type definitions
└── namespace_test.go      ← comprehensive namespace isolation tests

internal/container/
├── manager.go            ← calls namespace component for isolation
└── state.go               ← tracks namespace handles for cleanup

cmd/container-runtime/
└── main.go                ← integrates namespace manager
```

Infrastructure Starter Code

This complete namespace manager provides the foundational structure for namespace isolation:

GO

```
// internal/namespace/types.go

package namespace

import (
    "fmt"
    "os"
    "time"
)

// NamespaceType represents different Linux namespace types

type NamespaceType int

const (
    PIDNamespace NamespaceType = iota
    MountNamespace
    NetworkNamespace
    UTSNamespace
    UserNamespace
    IPCNamespace
)
}

// NamespaceConfig defines which namespaces to create and their configuration

type NamespaceConfig struct {

    EnablePID      bool
    EnableMount    bool
    EnableNetwork  bool
    EnableUTS      bool
    EnableUser     bool
    EnableIPC      bool
}
```

```
// User namespace mappings

UIDMappings []UIDMapping
GIDMappings []GIDMapping


// UTS configuration

Hostname string
Domain   string
}

// UIDMapping defines user ID mapping for user namespaces

type UIDMapping struct {

    ContainerID uint32 // UID inside container

    HostID      uint32 // UID on host system

    Size        uint32 // Range of IDs to map
}

// GIDMapping defines group ID mapping for user namespaces

type GIDMapping struct {

    ContainerID uint32 // GID inside container

    HostID      uint32 // GID on host system

    Size        uint32 // Range of IDs to map
}

// NamespaceHandle tracks created namespaces for cleanup

type NamespaceHandle struct {

    PID          int       // Process ID in new namespaces

    NamespaceTypes []NamespaceType // Which namespaces were created
}
```

```
CreatedAt    time.Time // When namespaces were created  
  
OverlayPath string    // Path to overlay filesystem mount  
  
NetworkNS   string    // Network namespace identifier  
}
```

GO

```
// internal/namespace/namespace.go

package namespace

import (
    "context"
    "fmt"
    "os"
    "golang.org/x/sys/unix"
)

// Manager handles namespace creation and lifecycle

type Manager struct {
    // Configuration
    overlayRoot string
    bridgeName  string

    // Runtime state
    activeNamespaces map[string]*NamespaceHandle
}

// NewManager creates a new namespace manager

func NewManager(overlayRoot, bridgeName string) *Manager {
    return &Manager{
        overlayRoot:      overlayRoot,
        bridgeName:       bridgeName,
        activeNamespaces: make(map[string]*NamespaceHandle),
    }
}
```

```
// CreateNamespaces creates isolated namespaces according to configuration

func (m *Manager) CreateNamespaces(containerID string, config *NamespaceConfig)
(*NamespaceHandle, error) {

    // Build namespace flags

    flags := 0

    var namespaceTypes []NamespaceType

    if config.EnablePID {

        flags |= unix.CLONE_NEWPID

        namespaceTypes = append(namespaceTypes, PIDNamespace)

    }

    if config.EnableMount {

        flags |= unix.CLONE_NEWNS

        namespaceTypes = append(namespaceTypes, MountNamespace)

    }

    if config.EnableNetwork {

        flags |= unix.CLONE_NEWNET

        namespaceTypes = append(namespaceTypes, NetworkNamespace)

    }

    if config.EnableUTS {

        flags |= unix.CLONE_NEWUTS

        namespaceTypes = append(namespaceTypes, UTSNamespace)

    }

    if config.EnableUser {

        flags |= unix.CLONE_NEWUSER

        namespaceTypes = append(namespaceTypes, UserNamespace)

    }

}
```

```
if config.EnableIPC {

    flags |= unix.CLONE_NEWIPC

    namespaceTypes = append(namespaceTypes, IPCNamespace)

}

// Create pipe for parent-child synchronization

readFD, writeFD, err := os.Pipe()

if err != nil {

    return nil, fmt.Errorf("failed to create sync pipe: %w", err)

}

defer readFD.Close()

defer writeFD.Close()

// Fork process with new namespaces

pid, err := unix.ForkExec("/proc/self/exe", []string{"container-runtime", "namespace-init"},

    &unix.ProcAttr{

        Files: []uintptr{0, 1, 2, writeFD.Fd()},
        Sys:   &unix.SysProcAttr{Cloneflags: uintptr(flags)},
    })



if err != nil {

    return nil, fmt.Errorf("failed to create namespaces: %w", err)

}

handle := &NamespaceHandle{
    PID:          pid,
```

```

NamespaceTypes: namespaceTypes,

CreatedAt:      time.Now(),

NetworkNS:      fmt.Sprintf("netns-%s", containerID),

}

// Configure user namespace mappings if enabled

if config.EnableUser {

    if err := m.configureUserNamespace(pid, config.UIDMappings, config.GIDMappings);
err != nil {

        m.CleanupNamespaces(handle)

        return nil, fmt.Errorf("failed to configure user namespace: %w", err)
    }
}

// Signal child process that setup is complete

writeFD.Write([]byte("ready"))

m.activeNamespaces[containerID] = handle

return handle, nil
}

```

Core Logic Skeleton Code

The following functions provide the structure for namespace isolation implementation:

GO

```
// internal/namespace/isolation.go

package namespace

import (
    "fmt"
    "os"
    "path/filepath"
    "golang.org/x/sys/unix"
)

// configureUserNamespace sets up UID/GID mappings for user namespace

func (m *Manager) configureUserNamespace(pid int, uidMaps []UIDMapping, gidMaps []GIDMapping) error {

    // TODO 1: Open /proc/[pid]/uid_map file for writing

    // TODO 2: Write each UID mapping in format "container_id host_id size\n"

    // TODO 3: Open /proc/[pid]/gid_map file for writing

    // TODO 4: Write each GID mapping in format "container_id host_id size\n"

    // TODO 5: Verify mappings were applied by reading back the files

    // Hint: Must write "deny" to /proc/[pid]/setgroups before writing gid_map

    return fmt.Errorf("configureUserNamespace not implemented")
}

// setupEssentialMounts creates required filesystem mounts in container

func (m *Manager) setupEssentialMounts() error {

    // TODO 1: Mount /proc filesystem with mount("proc", "/proc", "proc", 0, "")

    // TODO 2: Mount /sys filesystem read-only for container safety

    // TODO 3: Create /dev as tmpfs and populate essential device nodes

    // TODO 4: Mount /dev/pts for terminal support
```

```
// TODO 5: Create /tmp as tmpfs for temporary file storage

// TODO 6: Verify all mounts succeeded and are accessible

// Hint: Use unix.Mount() with appropriate flags for each filesystem type


return fmt.Errorf("setupEssentialMounts not implemented")

}

// configureHostname sets container hostname in UTS namespace

func (m *Manager) configureHostname(hostname, domain string) error {

    // TODO 1: Validate hostname format (RFC compliance)

    // TODO 2: Call unix.Sethostname() to set hostname in UTS namespace

    // TODO 3: Call unix.Setdomainname() to set domain if provided

    // TODO 4: Verify hostname was set by reading back with unix.Gethostname()

    // Hint: Hostname changes only affect the current UTS namespace


return fmt.Errorf("configureHostname not implemented")

}
```

GO

```
// internal/namespace/pivot.go

package namespace

import (
    "fmt"
    "os"
    "path/filepath"
    "golang.org/x/sys/unix"
)

// ExecutePivotRoot switches container root filesystem using pivot_root

func (m *Manager) ExecutePivotRoot(newRoot string) error {
    // TODO 1: Verify newRoot is a mount point (check /proc/mounts)
    // TODO 2: Create old root directory inside newRoot (typically /old-root)
    // TODO 3: Change current directory to newRoot to avoid EBUSY
    // TODO 4: Call unix.PivotRoot(newRoot, oldRootPath) system call
    // TODO 5: Change directory to / in the new root filesystem
    // TODO 6: Unmount old root filesystem to complete isolation
    // TODO 7: Remove old root directory to clean up mount point
    // Hint: pivot_root requires new root to be mount point, old root under new root

    return fmt.Errorf("ExecutePivotRoot not implemented")
}

// prepareOverlayMount creates overlay filesystem from image layers

func (m *Manager) prepareOverlayMount(containerID string, layers []string) (string, error) {
    // TODO 1: Create overlay directories: lower, upper, work, merged
    // TODO 2: Format overlay mount options with lower layers separated by colons
```

```
// TODO 3: Mount overlayfs with mount("overlay", merged, "overlay", 0, options)

// TODO 4: Verify merged directory shows combined layer content

// TODO 5: Return path to merged directory for use as new root

// Hint: Overlay options format: "lowerdir=layer1:layer2,upperdir=upper,workdir=work"

return "", fmt.Errorf("prepareOverlayMount not implemented")

}

// validateMountPoint checks if path is a filesystem mount point

func (m *Manager) validateMountPoint(path string) error {

    // TODO 1: Get filesystem stat info for the path

    // TODO 2: Get filesystem stat info for path's parent directory

    // TODO 3: Compare device IDs - different devices means mount point

    // TODO 4: Also check /proc/mounts for explicit mount entry

    // Hint: Use unix.Stat() and compare st.Dev fields for device ID check

    return fmt.Errorf("validateMountPoint not implemented")

}
```

GO

```
// internal/namespace/cleanup.go

package namespace

import (
    "fmt"
    "os"
    "syscall"
    "golang.org/x/sys/unix"
)

// CleanupNamespaces removes container namespaces and associated resources

func (m *Manager) CleanupNamespaces(handle *NamespaceHandle) error {
    // TODO 1: Send SIGTERM to container process to request graceful shutdown
    // TODO 2: Wait up to 10 seconds for process to exit voluntarily
    // TODO 3: Send SIGKILL if process hasn't exited after timeout
    // TODO 4: Unmount overlay filesystem if it was mounted
    // TODO 5: Remove overlay directories (upper, work, merged)
    // TODO 6: Clean up any remaining mount points in namespace
    // TODO 7: Remove handle from activeNamespaces map

    // Hint: Use unix.Kill() for signals, unix.Wait4() for process cleanup

    return fmt.Errorf("CleanupNamespaces not implemented")
}

// cleanupMounts removes all mounts associated with container

func (m *Manager) cleanupMounts(overlayPath string) error {
    // TODO 1: Read /proc/mounts to find all mounts under overlay path
    // TODO 2: Unmount each mount point in reverse order (deepest first)
```

```

    // TODO 3: Use MNT_DETACH flag for lazy unmounting if normal unmount fails

    // TODO 4: Remove empty directories after successful unmount

    // TODO 5: Verify no mount points remain under overlay path

    // Hint: Must unmount child mounts before parent mounts to avoid EBUSY


    return fmt.Errorf("cleanupMounts not implemented")
}

// killProcessTree terminates all processes in container namespace

func (m *Manager) killProcessTree(pid int, signal syscall.Signal) error {

    // TODO 1: Find all processes in same PID namespace as container

    // TODO 2: Send specified signal to each process in namespace

    // TODO 3: Wait for processes to exit after SIGTERM

    // TODO 4: Force kill with SIGKILL if processes don't exit gracefully

    // TODO 5: Verify all namespace processes have exited

    // Hint: Read /proc/[pid]/ns/pid to identify processes in same namespace


    return fmt.Errorf("killProcessTree not implemented")
}

```

Milestone Checkpoints

After implementing the namespace isolation component, verify correct behavior with these checkpoints:

Checkpoint 1: PID Namespace Isolation

- Run: `go test -run TestPIDNamespace ./internal/namespace/`
- Expected: Container process sees itself as PID 1, cannot see host processes
- Manual test: Create container and run `ps aux` - should show only container processes
- Failure signs: Seeing host processes indicates PID namespace not created or /proc not remounted

Checkpoint 2: Mount Namespace and Pivot Root

- Run: `go test -run TestMountNamespace ./internal/namespace/`

- Expected: Container has isolated filesystem view with overlay layers
- Manual test: Container `/bin` should differ from host `/bin` directory
- Failure signs: "EINVAL from pivot_root" means mount point requirements not met

Checkpoint 3: Network Namespace Isolation

- Run: `go test -run TestNetworkNamespace ./internal/namespace/`
- Expected: Container starts with only loopback interface
- Manual test: Run `ip link show` in container - should show only `lo` interface initially
- Failure signs: Seeing host network interfaces means network namespace not created

Checkpoint 4: Complete Namespace Integration

- Run: `go test ./internal/namespace/`
- Expected: All namespace types work together without conflicts
- Manual test: Start container with all namespaces and verify isolation
- Failure signs: Resource leaks, mount point conflicts, or partial isolation indicate cleanup issues

Resource Control Component

Milestone(s): Milestone 2 (Resource Limits with Cgroups) - This section implements CPU, memory, and I/O limits using cgroups v2 for container resource control, along with resource monitoring and out-of-memory handling.

Mental Model: Resource Budget Manager

Think of cgroups like a family budget manager that allocates monthly allowances to children. Just as parents set spending limits for each child (clothing budget, entertainment budget, snack budget), cgroups set resource limits for each container (memory budget, CPU budget, disk I/O budget). When a child tries to spend more than their allowance, the budget manager either denies the purchase or finds emergency funds from elsewhere. Similarly, when a container tries to use more resources than allocated, cgroups either deny the request (causing the process to wait or fail) or trigger emergency actions like killing the process.

The budget manager also tracks spending throughout the month, sending warnings when a child approaches their limit ("You've spent 80% of your entertainment budget"). Cgroups provide the same visibility, exposing current resource usage so the container runtime can monitor consumption and warn about approaching limits. Just as the family budget ensures no single child monopolizes the household income, cgroups ensure no single container starves other containers of system resources.

This analogy extends to hierarchical budgets: a family might allocate 40% of income to children's allowances, then subdivide that among individual children. Cgroups work similarly with hierarchical resource allocation,

where you might dedicate 50% of system memory to containers, then divide that pool among individual containers based on their priority and requirements.

Cgroup Controller Configuration

The **cgroup controller configuration** component manages resource limits by interfacing with the Linux cgroups v2 unified hierarchy. Each controller manages a specific resource type and exposes configuration files in `/sys/fs/cgroup` that define limits, track usage, and trigger enforcement actions.

Decision: Cgroups v2 Over Cgroups v1

- **Context:** Linux systems support both cgroups v1 (legacy) and cgroups v2 (modern) interfaces for resource control, with different APIs and capabilities.
- **Options Considered:** Use cgroups v1 for broader compatibility, use cgroups v2 for modern features, support both versions with abstraction layer.
- **Decision:** Use cgroups v2 unified hierarchy exclusively.
- **Rationale:** Cgroups v2 provides unified hierarchy (single tree instead of per-controller trees), better memory accounting, improved CPU isolation, and simplified configuration. Modern distributions default to v2, and v1 is deprecated. The learning value comes from understanding modern kernel interfaces.
- **Consequences:** Requires kernel 4.5+ and systemd systems with cgroups v2 enabled. Simpler implementation with unified interface, but incompatible with older systems using v1-only setups.

Option	Pros	Cons	Chosen
Cgroups v1	Broader compatibility, extensive documentation	Complex multi-hierarchy design, deprecated, limited features	No
Cgroups v2	Modern unified hierarchy, better accounting, future-proof	Requires newer kernels, some tools still v1-only	Yes
Support Both	Maximum compatibility	Complex abstraction, doubles testing burden	No

The cgroup controller system creates a dedicated cgroup for each container under `/sys/fs/cgroup/container-runtime/[container-id]` and enables specific controllers based on the container's `ResourceLimits` specification. The system writes configuration values to controller-specific files and monitors usage through statistics files.

Memory Controller Configuration

The **memory controller** enforces memory limits using the `memory.max` interface, which provides hard memory capping with configurable out-of-memory behavior. When a container approaches its memory limit,

the kernel memory reclaim system attempts to free memory through page cache eviction and swap-out. If reclaim cannot free sufficient memory, the OOM killer terminates processes within the cgroup.

Configuration File	Purpose	Format	Example Value
<code>memory.max</code>	Hard memory limit	Integer bytes or "max"	536870912 (512MB)
<code>memory.high</code>	Soft limit triggering reclaim	Integer bytes or "max"	402653184 (384MB)
<code>memory.swap.max</code>	Swap space limit	Integer bytes or "max"	134217728 (128MB)
<code>memory.oom.group</code>	OOM kill entire cgroup	0 or 1	1
<code>memory.current</code>	Current memory usage (read-only)	Integer bytes	123456789
<code>memory.peak</code>	Peak memory usage since creation	Integer bytes	234567890

The memory controller setup process follows these steps:

1. Enable the memory controller by writing `+memory` to the parent cgroup's `cgroup.subtree_control` file
2. Create the container-specific cgroup directory under the container runtime's cgroup hierarchy
3. Write the hard memory limit from `ResourceLimits.Memory` to `memory.max`
4. Set the soft limit to 75% of hard limit in `memory.high` to trigger early memory pressure
5. Configure swap limits based on container specification or disable swap entirely for predictable behavior
6. Enable group OOM killing to ensure related processes are cleaned up together

Memory limits without corresponding swap limits can allow containers to exceed their intended memory budget by using swap space. Always configure both `memory.max` and `memory.swap.max` for predictable resource enforcement.

CPU Controller Configuration

The **CPU controller** implements CPU bandwidth limiting using the Completely Fair Scheduler (CFS) quota mechanism. Instead of CPU priority or weight-based sharing, the controller enforces hard CPU time limits by allocating specific microseconds of CPU time per scheduling period.

Configuration File	Purpose	Format	Example Value
<code>cpu.max</code>	CPU quota and period	"quota period" or "max"	<code>50000 100000</code> (50% CPU)
<code>cpu.weight</code>	Relative CPU share	Integer 1-10000	<code>100</code> (normal weight)
<code>cpu.stat</code>	CPU usage statistics	Multi-line key-value	<code>usage_usec 1234567</code>
<code>cpuset.cpus</code>	Allowed CPU cores	CPU list format	<code>0-3</code> or <code>0, 2, 4</code>
<code>cpuset.mems</code>	Allowed memory nodes	NUMA node list	<code>0</code>

The CPU quota system works by defining a scheduling period (typically 100ms) and allocating a portion of that period to the container. A container with `cpu.max` set to `50000 100000` receives 50,000 microseconds of CPU time every 100,000 microseconds (50% of one CPU core). The CFS scheduler enforces this by throttling the container's processes when they exhaust their quota within the current period.

CPU controller setup algorithm:

1. Enable the CPU controller by writing `+cpu` to the parent cgroup's `cgroup.subtree_control` file
2. Calculate CPU quota from `ResourceLimits.CPU` value (nanoseconds per second) to microseconds per 100ms period
3. Write the quota and period to `cpu.max` in the format "quota period"
4. Set CPU weight based on container priority (higher weight gets more CPU under contention)
5. Configure CPU affinity using `cpuset.cpus` if the container should run on specific cores
6. Set memory node affinity using `cpuset.mems` for NUMA-aware scheduling

The CPU limit calculation converts from nanoseconds-per-second to the CFS quota format. For example, a `ResourceLimits.CPU` value of 500,000,000 nanoseconds (0.5 CPU cores) becomes a quota of 50,000 microseconds per 100,000 microsecond period.

Device Controller Configuration

The **device controller** restricts container access to device files under `/dev`, preventing containers from accessing host hardware directly unless explicitly permitted. This controller uses allow/deny lists with device type, major/minor numbers, and permission specifications.

Configuration File	Purpose	Format	Example Value
<code>devices.allow</code>	Permit device access	"type major:minor permissions"	<code>c 1:3 rwm</code> (allow /dev/null)
<code>devices.deny</code>	Restrict device access	"type major:minor permissions"	<code>b *:* rwm</code> (deny all block devices)
<code>devices.list</code>	Current device permissions	Multi-line device specs	<code>c 1:3 rwm\nnc 1:5 rwm</code>

The device controller starts with a deny-all policy and selectively permits access to essential devices that containers typically need. The standard device allowlist includes character devices for `/dev/null`, `/dev/zero`, `/dev/urandom`, and `/dev/random`, while denying access to block devices, raw devices, and hardware-specific character devices.

Device controller setup process:

1. Enable the devices controller (automatically enabled in cgroups v2 when processes are added)
2. Apply the default deny-all policy by writing `a *:* rwm` to `devices.deny`
3. Allow essential character devices by writing entries to `devices.allow` for each permitted device
4. Parse any additional device permissions from the container specification
5. Write additional device allow rules based on container requirements (e.g., GPU access for ML containers)

Standard container device allowlist:

- `c 1:3 rwm` - `/dev/null` (null device)
- `c 1:5 rwm` - `/dev/zero` (zero device)
- `c 1:8 rwm` - `/dev/random` (random number generator)
- `c 1:9 rwm` - `/dev/urandom` (non-blocking random)
- `c 5:0 rwm` - `/dev/tty` (controlling terminal)
- `c 5:2 rw` - `/dev/ptmx` (pseudoterminal master)
- `c 136:/* rw` - `/dev/pts/*` (pseudoterminal slaves)

Cgroup Creation and Assignment Algorithm

The cgroup creation process establishes the resource control hierarchy and assigns the container process to the appropriate cgroup for enforcement. This process must handle race conditions, permission requirements, and cleanup on failure.

Container cgroup setup algorithm:

1. **Validate cgroups v2 availability** by checking that `/sys/fs/cgroup/cgroup.controllers` exists and contains required controllers

2. **Create runtime cgroup hierarchy** if it doesn't exist: `/sys/fs/cgroup/container-runtime/` with appropriate permissions
3. **Enable controllers** by writing `+memory +cpu +pids` to `/sys/fs/cgroup/container-runtime/cgroup.subtree_control`
4. **Create container cgroup** directory at `/sys/fs/cgroup/container-runtime/[container-id]/`
5. **Configure memory limits** by writing `ResourceLimits.Memory` to `memory.max`
6. **Configure CPU limits** by calculating and writing quota/period to `cpu.max`
7. **Configure PID limits** by writing `ResourceLimits.PIDs` to `pids.max`
8. **Configure device access** by setting up device allow/deny rules
9. **Assign container process** by writing the container's PID to `cgroup.procs`
10. **Verify assignment** by reading `cgroup.procs` and confirming the PID appears in the list

The process assignment step occurs after the container process starts but before it begins executing the user-specified command. The container runtime forks the container process, performs namespace setup, assigns it to the cgroup, then executes the target command. This ensures resource limits apply from the beginning of command execution.

⚠️ Pitfall: Process Assignment Timing Assigning a process to a cgroup after it has already consumed significant resources allows it to exceed limits during startup. Always assign processes to cgroups immediately after fork, before any significant work begins.

Resource Usage Monitoring

The **resource usage monitoring** system tracks current resource consumption by reading statistics from cgroup controller files and detecting when containers approach their configured limits. This monitoring enables the container runtime to implement early warning systems, automatic scaling decisions, and proactive resource management.

Decision: Pull-Based Monitoring Over Event-Based

- **Context:** Cgroups provide both polling-based monitoring (reading stat files) and event-based monitoring (using eventfd for threshold notifications).
- **Options Considered:** Periodic polling of cgroup stat files, event-based notifications using cgroup.events, hybrid approach with polling plus critical event notifications.
- **Decision:** Use periodic polling with configurable intervals for primary monitoring.
- **Rationale:** Polling provides consistent monitoring behavior, easier implementation, and better integration with existing monitoring systems. Event-based monitoring requires complex eventfd handling and may miss gradual resource increases. The polling overhead is minimal compared to container workloads.
- **Consequences:** Slight monitoring latency (up to polling interval) but simpler implementation. Requires tuning polling frequency to balance responsiveness with overhead.

Monitoring Approach	Pros	Cons	Chosen
Periodic Polling	Simple implementation, consistent behavior, integrates with standard monitoring	Fixed latency, potential overhead with frequent polling	Yes
Event Notifications	Low latency, minimal overhead, kernel-driven	Complex eventfd handling, can miss gradual changes	No
Hybrid Approach	Best of both worlds	Complex implementation, harder to debug	No

Memory Usage Monitoring

Memory usage monitoring tracks both current consumption and memory pressure indicators to detect containers approaching their limits before OOM conditions occur. The monitoring system reads multiple memory statistics to build a comprehensive view of container memory behavior.

Memory Statistic	File Path	Description	Warning Threshold
Current Usage	<code>memory.current</code>	Total memory currently allocated	> 80% of <code>memory.max</code>
Peak Usage	<code>memory.peak</code>	Highest memory usage since container start	> 90% of <code>memory.max</code>
Cache Usage	<code>memory.stat</code> (cache field)	Page cache memory that can be reclaimed	< 10% of total usage
Swap Usage	<code>memory.swap.current</code>	Current swap space consumption	> 50% of <code>memory.swap.max</code>
Memory Events	<code>memory.events</code>	OOM kills, memory pressure events	Any <code>oom_kill</code> events

Memory monitoring algorithm:

1. **Read current memory usage** from `memory.current` and calculate percentage of `memory.max`
2. **Check memory events** by parsing `memory.events` for `oom`, `oom_kill`, `oom_group_kill` counters
3. **Analyze memory composition** by parsing `memory.stat` to understand cache vs anonymous memory ratio
4. **Detect memory pressure** by comparing current usage trends with historical patterns
5. **Calculate memory efficiency** by examining the ratio of productive memory (heap, stack) to overhead (page tables, kernel buffers)
6. **Generate alerts** when usage exceeds 80% of limit or memory pressure events indicate thrashing
7. **Update monitoring metrics** for integration with external monitoring systems

The memory monitoring system maintains a sliding window of usage samples to detect trends and predict when containers might hit memory limits. Rapid memory growth patterns trigger early warnings, while sustained high usage with frequent page reclaim indicates memory pressure requiring attention.

Memory usage can spike temporarily during garbage collection or memory allocation bursts. Use moving averages over 30-60 second windows rather than instantaneous readings for alerting thresholds.

CPU Usage Monitoring

CPU usage monitoring tracks both absolute CPU consumption and throttling statistics to understand how containers utilize their allocated CPU quotas. The monitoring system distinguishes between CPU usage (actual compute time) and CPU throttling (time spent waiting due to quota limits).

CPU Statistic	File Path	Description	Threshold
Usage Time	<code>cpu.stat</code> (usage_usec)	Total CPU microseconds consumed	Compare to quota allocation
User Time	<code>cpu.stat</code> (user_usec)	CPU time spent in user space	High ratio indicates compute-bound
System Time	<code>cpu.stat</code> (system_usec)	CPU time spent in kernel space	High ratio indicates I/O-bound
Throttled Time	<code>cpu.stat</code> (throttled_usec)	Time spent waiting due to quota limits	> 20% indicates quota too low
Throttled Periods	<code>cpu.stat</code> (nr_throttled)	Number of periods where quota was exhausted	Increasing trend indicates pressure

CPU monitoring algorithm:

1. **Sample CPU statistics** by reading and parsing the multi-line `cpu.stat` file
2. **Calculate CPU utilization** by comparing `usage_usec` delta with time elapsed and quota available
3. **Detect CPU throttling** by monitoring `throttled_usec` and `nr_throttled` increases between samples
4. **Analyze CPU efficiency** by examining user/system time ratios to understand workload characteristics
5. **Track CPU quota effectiveness** by comparing actual usage patterns with allocated quotas
6. **Generate CPU pressure alerts** when throttling exceeds thresholds or containers consistently hit quota limits
7. **Maintain CPU usage history** for capacity planning and right-sizing recommendations

The CPU monitoring system calculates CPU utilization as a percentage by taking the difference in `usage_usec` between samples, dividing by the time elapsed, and comparing to the allocated quota. For example, if a container uses 45,000 microseconds of CPU time in a 100,000 microsecond period with a 50,000 microsecond quota, it's running at 90% of its allocation ($45,000 / 50,000$).

Resource Monitoring Data Structures

The resource monitoring system maintains current usage state, historical trends, and alert thresholds for each container using structured data that enables efficient monitoring and reporting.

Field Name	Type	Description
ContainerID	string	Unique identifier for the monitored container
CgroupPath	string	Filesystem path to the container's cgroup directory
LastSampleTime	time.Time	Timestamp of the most recent monitoring sample
MemoryUsage	ResourceUsage	Current memory consumption and trends
CPUUsage	ResourceUsage	Current CPU utilization and throttling statistics
IOUsage	ResourceUsage	Disk and network I/O statistics
AlertThresholds	AlertConfig	Warning and critical thresholds for each resource
UsageHistory	[]ResourceSample	Sliding window of historical usage samples

Field Name	Type	Description
Current	int64	Current resource usage value
Peak	int64	Highest usage since monitoring started
Limit	int64	Configured resource limit from cgroup
UtilizationPercent	float64	Current usage as percentage of limit
TrendSlope	float64	Rate of change in usage over time
PressureIndicators	map[string]int64	Pressure metrics (throttling, OOM events, etc.)

Field Name	Type	Description
WarningThreshold	float64	Utilization percentage triggering warnings
CriticalThreshold	float64	Utilization percentage triggering critical alerts
TrendWarningSlope	float64	Rate of increase triggering trend-based warnings
EventCountThreshold	int64	Number of pressure events triggering alerts

Out-of-Memory Handling

The **out-of-memory handling** system detects and manages memory exhaustion scenarios when containers exceed their allocated memory limits. The OOM handling mechanism combines kernel-level process termination with application-level cleanup and recovery procedures to maintain system stability.

Decision: Cgroup-Spaced OOM Killing Over System-Wide

- **Context:** When memory is exhausted, the Linux kernel can kill processes either system-wide (based on global OOM scores) or within the specific cgroup that exceeded limits.
- **Options Considered:** Allow system-wide OOM killer to select victims, enable cgroup-scoped OOM killing, implement custom OOM handling with memory pressure monitoring.
- **Decision:** Use cgroup-scoped OOM killing with `memory.oom.group` enabled.
- **Rationale:** Cgroup-scoped killing ensures only the offending container's processes are terminated, protecting other containers and the host system. Group killing ensures related processes (parent/child, shared memory) are cleaned up together, preventing orphaned resources.
- **Consequences:** Container processes may be killed more aggressively than with system-wide OOM, but system stability is preserved. Requires proper application design to handle sudden termination gracefully.

OOM Handling Approach	Pros	Cons	Chosen
System-Wide OOM	Natural Linux behavior, well-tested	Can kill host processes, unpredictable victim selection	No
Cgroup-Spaced OOM	Precise isolation, predictable behavior	More aggressive killing, requires container awareness	Yes
Custom Memory Pressure	Fine-grained control, graceful degradation	Complex implementation, potential memory leaks	No

OOM Detection and Response

The OOM detection system monitors memory events and usage patterns to identify when containers are approaching or experiencing memory exhaustion. The system distinguishes between predictable memory pressure (gradual increase toward limits) and sudden memory spikes that trigger immediate OOM conditions.

OOM detection algorithm:

1. **Monitor memory events** by reading `memory.events` and tracking increases in `oom`, `oom_kill`, and `oom_group_kill` counters
2. **Detect memory pressure** by comparing `memory.current` with `memory.max` and monitoring rate of increase
3. **Check swap exhaustion** by comparing `memory.swap.current` with `memory.swap.max` if swap is enabled
4. **Analyze allocation failures** by monitoring `memory.events` for `high`, `max`, and other pressure indicators
5. **Predict OOM conditions** using usage trends and allocation patterns to warn before limits are reached

6. **Trigger preventive actions** when usage exceeds warning thresholds (typically 90% of memory limit)
7. **Handle OOM events** by detecting process termination and initiating cleanup procedures

The OOM response system implements a tiered approach based on memory pressure severity:

Warning Level (80-90% memory usage):

- Log memory pressure warnings with usage statistics
- Notify monitoring systems about approaching memory limits
- Optionally trigger garbage collection hints for supported runtimes
- Increase monitoring frequency for more responsive detection

Critical Level (90-95% memory usage):

- Generate critical alerts to monitoring and logging systems
- Implement emergency memory reclaim by dropping non-essential caches
- Prepare for potential OOM by saving critical application state
- Consider triggering application-specific memory reduction mechanisms

OOM Event (process killed by kernel):

- Detect OOM kills by monitoring `memory.events` counters
- Log detailed OOM information including memory usage at time of kill
- Initiate container cleanup procedures to free resources
- Update container state to reflect OOM termination cause
- Optionally implement restart policies based on OOM frequency

OOM Recovery and Cleanup

The OOM recovery system handles the aftermath of memory exhaustion events, ensuring proper resource cleanup and implementing restart policies that prevent repeated OOM cycles. The recovery process must distinguish between application bugs (memory leaks) and insufficient memory allocation.

OOM Recovery Action	Trigger Condition	Description	Implementation
Immediate Cleanup	Process killed by OOM killer	Remove container resources, update state	Clean cgroups, namespaces, mounts
Diagnostic Logging	Any OOM event	Record memory usage patterns and triggers	Write detailed logs with usage history
Restart Decision	Container marked for restart	Evaluate restart policies and backoff	Check restart count, implement delays
Resource Adjustment	Repeated OOM within window	Increase memory limits or reduce requests	Update ResourceLimits if policy allows

OOM recovery algorithm:

1. **Detect OOM termination** by monitoring container process exit status and memory events
2. **Preserve diagnostic information** by capturing memory usage statistics, allocation patterns, and timeline leading to OOM
3. **Initiate resource cleanup** by removing cgroups, unmounting filesystems, and cleaning network resources
4. **Update container state** to reflect OOM termination with detailed exit reason and resource usage
5. **Evaluate restart eligibility** based on restart policies, backoff intervals, and OOM frequency
6. **Implement restart backoff** to prevent rapid restart cycles that repeatedly trigger OOM
7. **Optionally adjust resources** if restart policies allow automatic memory limit increases
8. **Generate alerts** for persistent OOM patterns that require manual intervention

The restart backoff system implements exponential delays to prevent OOM thrashing:

- First OOM: immediate restart if restart policy allows
- Second OOM within 5 minutes: 10-second delay before restart
- Third OOM within 15 minutes: 60-second delay before restart
- Subsequent OOMs: exponential backoff up to 300-second maximum delay

⚠️ Pitfall: OOM Loop Prevention Containers that immediately trigger OOM after restart can create rapid resource allocation/deallocation cycles that destabilize the host system. Always implement restart backoff and consider automatic memory limit adjustments for chronic OOM containers.

Memory Pressure Prevention

The memory pressure prevention system implements proactive measures to avoid OOM conditions by monitoring memory trends and triggering early intervention when containers approach their limits. This system focuses on graceful degradation rather than hard termination.

Memory pressure prevention strategies:

Trend-Based Prediction:

- Monitor memory allocation rate over 60-second windows
- Extrapolate current trends to predict when memory limits will be reached
- Trigger warnings when projected exhaustion time falls below 5 minutes
- Account for allocation bursts and garbage collection patterns

Application-Level Cooperation:

- Send SIGUSR1 signals to applications supporting memory pressure notifications
- Implement cgroup memory.pressure interface for pressure stall information
- Allow applications to implement graceful memory reduction strategies
- Provide memory usage APIs for application self-monitoring

Cache and Buffer Management:

- Monitor page cache usage through memory.stat analysis
- Trigger early page cache reclaim when memory pressure builds
- Implement buffer size reductions for network and disk I/O
- Coordinate with kernel memory management for optimal reclaim timing

Dynamic Resource Adjustment:

- Temporarily increase memory limits during predicted short-term spikes
- Implement memory borrowing from unused container allocations
- Coordinate with cluster-level resource management for dynamic scaling
- Revert temporary adjustments after memory pressure subsides

The prevention system maintains effectiveness by learning from historical patterns and adjusting prediction algorithms based on application behavior. Containers with predictable memory allocation patterns receive more accurate warnings, while applications with erratic memory usage receive more conservative early warnings.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Cgroup Interface	Direct filesystem operations with os.OpenFile	go-cgroups library with type safety
Resource Monitoring	Periodic file reads with time.Ticker	Prometheus client library with metrics
OOM Detection	Poll memory.events file	Use eventfd for kernel notifications
Data Persistence	JSON files in /var/lib/container-runtime	SQLite database with structured schema

Recommended File Structure

```
container-runtime/
  internal/cgroups/
    cgroups.go          ← cgroup controller interface
    memory.go          ← memory controller implementation
    cpu.go              ← CPU controller implementation
    devices.go          ← device controller implementation
    monitor.go          ← resource usage monitoring
    oom.go              ← out-of-memory handling
    cgroups_test.go     ← unit tests for all controllers
  internal/resources/
    limits.go           ← ResourceLimits validation and conversion
    monitor.go          ← resource monitoring coordinator
    stats.go            ← usage statistics collection
```

Infrastructure Starter Code

Complete cgroup filesystem interface for reading and writing cgroup files:

GO

```
// pkg/cgroups/filesystem.go

package cgroups

import (
    "fmt"

    "io/ioutil"

    "os"

    "path/filepath"

    "strconv"

    "strings"
)

// CgroupFS provides filesystem operations for cgroup management

type CgroupFS struct {

    Root string // Root cgroup filesystem path (usually /sys/fs/cgroup)
}

// NewCgroupFS creates a new cgroup filesystem interface

func NewCgroupFS() *CgroupFS {
    return &CgroupFS{Root: "/sys/fs/cgroup"}
}

// WriteFile writes content to a cgroup file with proper error handling

func (c *CgroupFS) WriteFile(cgroupPath, filename, content string) error {
    fullPath := filepath.Join(c.Root, cgroupPath, filename)

    return ioutil.WriteFile(fullPath, []byte(content), 0644)
}

// ReadFile reads content from a cgroup file
```

```
func (c *CgroupFS) ReadFile(cgroupName string) (string, error) {

    fullPath := filepath.Join(c.Root, cgroupName, filename)

    data, err := ioutil.ReadFile(fullPath)

    if err != nil {

        return "", err

    }

    return strings.TrimSpace(string(data)), nil

}

// CreateCgroup creates a cgroup directory

func (c *CgroupFS) CreateCgroup(cgroupName string) error {

    fullPath := filepath.Join(c.Root, cgroupName)

    return os.MkdirAll(fullPath, 0755)

}

// RemoveCgroup removes a cgroup directory

func (c *CgroupFS) RemoveCgroup(cgroupName string) error {

    fullPath := filepath.Join(c.Root, cgroupName)

    return os.Remove(fullPath)

}

// ParseKeyValue parses multi-line key-value format from cgroup stat files

func (c *CgroupFS) ParseKeyValue(content string) map[string]int64 {

    result := make(map[string]int64)

    lines := strings.Split(content, "\n")

    for _, line := range lines {

        if line == "" {

            continue

        }

        key, value := strings.Split(line, "=")

        result[key] = strconv.ParseInt(value, 10, 64)

    }

    return result

}
```

```
}

parts := strings.Fields(line)

if len(parts) == 2 {

    if value, err := strconv.ParseInt(parts[1], 10, 64); err == nil {

        result[parts[0]] = value

    }
}

return result
}
```

Resource monitoring data structures and utilities:

GO

```
// internal/resources/types.go

package resources

import (
    "time"
)

// ResourceUsage tracks current and historical resource consumption

type ResourceUsage struct {
    Current          int64      `json:"current"`
    Peak             int64      `json:"peak"`
    Limit            int64      `json:"limit"`
    UtilizationPercent float64   `json:"utilization_percent"`
    TrendSlope       float64   `json:"trend_slope"`
    PressureIndicators map[string]int64 `json:"pressure_indicators"`
}

// ResourceSample represents a point-in-time resource measurement

type ResourceSample struct {
    Timestamp time.Time `json:"timestamp"`
    Value     int64     `json:"value"`
    Metadata  map[string]interface{} `json:"metadata,omitempty"`
}

// AlertConfig defines thresholds for resource usage alerts

type AlertConfig struct {
    WarningThreshold float64 `json:"warning_threshold"`
    CriticalThreshold float64 `json:"critical_threshold"`
    TrendWarningSlope float64 `json:"trend_warning_slope"`
}
```

```
    EventCountThreshold int64 `json:"event_count_threshold"`

}

// ContainerResourceState maintains monitoring state for a container

type ContainerResourceState struct {

    ContainerID      string          `json:"container_id"`

    CgroupPath       string          `json:"cgroup_path"`

    LastSampleTime   time.Time       `json:"last_sample_time"`

    MemoryUsage      ResourceUsage   `json:"memory_usage"`

    CPUUsage         ResourceUsage   `json:"cpu_usage"`

    IOUsage          ResourceUsage   `json:"io_usage"`

    AlertThresholds AlertConfig     `json:"alert_thresholds"`

    UsageHistory     []ResourceSample `json:"usage_history"`

}

// CalculateUtilization computes utilization percentage with bounds checking

func (r *ResourceUsage) CalculateUtilization() {

    if r.Limit > 0 {

        r.UtilizationPercent = float64(r.Current) / float64(r.Limit) * 100.0

    } else {

        r.UtilizationPercent = 0.0

    }

    // Clamp to reasonable bounds

    if r.UtilizationPercent > 100.0 {

        r.UtilizationPercent = 100.0

    }

    if r.UtilizationPercent < 0.0 {
```

```
r.UtilizationPercent = 0.0  
}  
}
```

Core Logic Skeleton Code

Cgroup controller implementation with memory, CPU, and device management:

GO

```
// internal/cgroups/controller.go

package cgroups

import (
    "context"
    "fmt"
    "path/filepath"
    "strconv"
    "time"
)

// Controller manages cgroup v2 controllers for container resource limits

type Controller struct {

    fs      *CgroupFS

    basePath string // Base path for container runtime cgroups
}

// NewController creates a new cgroup controller

func NewController() *Controller {
    return &Controller{
        fs:      NewCgroupFS(),
        basePath: "container-runtime",
    }
}

// CreateContainerCgroup sets up cgroup hierarchy and limits for a container

func (c *Controller) CreateContainerCgroup(containerID string, limits ResourceLimits) error {
    // TODO 1: Validate that cgroups v2 is available by checking
    // /sys/fs/cgroup/cgroup.controllers
```

```

    // TODO 2: Create runtime base cgroup if it doesn't exist: /sys/fs/cgroup/container-
runtime/

    // TODO 3: Enable required controllers (+memory +cpu +pids) in base cgroup
subtree_control

    // TODO 4: Create container-specific cgroup directory: /sys/fs/cgroup/container-
runtime/[containerID]/

    // TODO 5: Configure memory limits by writing limits.Memory to memory.max

    // TODO 6: Configure CPU limits by converting limits.CPU to quota/period format for
cpu.max

    // TODO 7: Configure PID limits by writing limits.PIDs to pids.max

    // TODO 8: Set up device controller with standard container device allowlist

    // TODO 9: Verify cgroup creation by reading back the written configuration files

    // Hint: Use c.fs.CreateCgroup() and c.fs.WriteFile() for filesystem operations

    // Hint: CPU limits in nanoseconds/second convert to microseconds/100ms for cpu.max

}

// AssignProcessToCgroup moves a process into the container's cgroup for resource
enforcement

func (c *Controller) AssignProcessToCgroup(containerID string, pid int) error {

    // TODO 1: Validate that the container cgroup exists before assignment

    // TODO 2: Write the PID to the container's cgroup.procs file

    // TODO 3: Verify assignment by reading cgroup.procs and confirming PID is listed

    // TODO 4: Handle assignment failures due to process state or permissions

    // Hint: Convert PID to string before writing to cgroup.procs

    // Hint: Assignment may fail if process is already exiting

}

// GetResourceUsage reads current resource consumption from cgroup statistics

func (c *Controller) GetResourceUsage(containerID string) (*ContainerResourceState, error)
{

    // TODO 1: Read memory usage from memory.current and memory.peak
}

```

```

// TODO 2: Read memory limits from memory.max for utilization calculation

// TODO 3: Read CPU usage from cpu.stat (parse usage_usec field)

// TODO 4: Read CPU throttling statistics from cpu.stat (throttled_usec, nr_throttled)

// TODO 5: Read memory pressure events from memory.events

// TODO 6: Calculate utilization percentages for each resource type

// TODO 7: Detect resource pressure indicators (high throttling, OOM events)

// TODO 8: Build and return ContainerResourceState with all collected metrics

// Hint: Use c.fs.ParseKeyValue() to parse multi-line stat files

// Hint: Handle missing or unreadable files gracefully (container may be gone)

}

// RemoveContainerCgroup cleans up cgroup when container is removed

func (c *Controller) RemoveContainerCgroup(containerID string) error {

    // TODO 1: Ensure no processes remain in the cgroup before removal

    // TODO 2: Read cgroup.procs and verify it's empty

    // TODO 3: Remove the cgroup directory using rmdir (not rm -rf)

    // TODO 4: Handle removal failures due to remaining processes or resources

    // TODO 5: Log warnings but don't fail if cgroup is already gone

    // Hint: Cgroup removal will fail if any processes are still assigned

    // Hint: Use c.fs.RemoveCgroup() which calls os.Remove() (equivalent to rmdir)

}

```

Memory controller with OOM detection and handling:

GO

```
// internal/cgroups/memory.go

package cgroups

import (
    "fmt"
    "strconv"
    "strings"
)

// MemoryController manages memory cgroup settings and OOM handling

type MemoryController struct {
    fs *CgroupFS
}

// SetMemoryLimit configures memory.max and related memory settings

func (m *MemoryController) SetMemoryLimit(cgroupPath string, limitBytes int64) error {
    // TODO 1: Write hard memory limit to memory.max file
    // TODO 2: Calculate and set soft limit (75% of hard limit) in memory.high
    // TODO 3: Configure swap limit in memory.swap.max (same as memory limit or disabled)
    // TODO 4: Enable group OOM killing by writing 1 to memory.oom.group
    // TODO 5: Verify limits are applied by reading back the written values
    // Hint: Convert limitBytes to string before writing
    // Hint: Use "0" for memory.swap.max to disable swap for the container
}

// GetMemoryUsage reads current memory consumption and pressure indicators

func (m *MemoryController) GetMemoryUsage(cgroupPath string) (*ResourceUsage, error) {
    // TODO 1: Read current memory usage from memory.current
    // TODO 2: Read peak memory usage from memory.peak
}
```

```

// TODO 3: Read memory limit from memory.max

// TODO 4: Parse memory.stat to get detailed memory breakdown (cache, anonymous, etc.)

// TODO 5: Read memory pressure events from memory.events

// TODO 6: Calculate utilization percentage and trend indicators

// TODO 7: Build ResourceUsage struct with pressure indicators map

// Hint: memory.events contains counters for low, high, max, oom, oom_kill events

// Hint: Use m.fs.ParseKeyValue() to parse memory.stat and memory.events

}

// CheckOOMEvents detects out-of-memory kills and returns event details

func (m *MemoryController) CheckOOMEvents(cgroupPath string) (bool, map[string]int64,
error) {

    // TODO 1: Read memory.events file to get OOM event counters

    // TODO 2: Check for increases in oom_kill and oom_group_kill counters

    // TODO 3: Compare with previously stored counter values to detect new events

    // TODO 4: Return true if new OOM events occurred since last check

    // TODO 5: Include all event counters in returned map for detailed logging

    // Hint: Store previous counter values to detect increases

    // Hint: Any non-zero oom_kill count indicates the container has experienced OOM

}

```

Resource monitoring with trend analysis and alerting:

GO

```
// internal/resources/monitor.go

package resources

import (
    "context"
    "sync"
    "time"
)

// ResourceMonitor coordinates resource usage monitoring across all containers

type ResourceMonitor struct {

    controller      *cgroups.Controller
    containers      map[string]*ContainerResourceState
    mutex           sync.RWMutex
    alertChannel    chan ResourceAlert
    stopChannel     chan struct{}
    monitoringInterval time.Duration
}

// ResourceAlert represents a resource usage alert condition

type ResourceAlert struct {

    ContainerID    string      `json:"container_id"`
    ResourceType   string      `json:"resource_type"`
    AlertLevel     string      `json:"alert_level"` // warning, critical, oom
    CurrentUsage   int64       `json:"current_usage"`
    Limit          int64       `json:"limit"`
    Message         string      `json:"message"`
    Timestamp       time.Time  `json:"timestamp"`
}
```

```
}

// StartMonitoring begins periodic resource usage monitoring for all containers

func (r *ResourceMonitor) StartMonitoring(ctx context.Context) {

    // TODO 1: Start background goroutine with ticker for monitoring interval

    // TODO 2: On each tick, iterate through all registered containers

    // TODO 3: Collect current resource usage for each container

    // TODO 4: Update usage history with new samples (maintain sliding window)

    // TODO 5: Calculate trends and detect approaching resource limits

    // TODO 6: Generate alerts for containers exceeding warning/critical thresholds

    // TODO 7: Check for OOM events and generate immediate critical alerts

    // TODO 8: Handle context cancellation for graceful shutdown

    // Hint: Use time.NewTicker(r.monitoringInterval) for periodic monitoring

    // Hint: Protect concurrent access to r.containers map with r.mutex

}

// RegisterContainer adds a container to the monitoring system

func (r *ResourceMonitor) RegisterContainer(containerID, cgroupPath string, thresholds AlertConfig) error {

    // TODO 1: Create ContainerResourceState for the new container

    // TODO 2: Initialize usage history slice with appropriate capacity

    // TODO 3: Store alert thresholds configuration

    // TODO 4: Add container to monitoring map with proper locking

    // TODO 5: Perform initial resource usage reading to establish baseline

    // Hint: Use r.mutex.Lock() for write access to r.containers map

    // Hint: Initialize UsageHistory with make([]ResourceSample, 0, historyCapacity)

}

// calculateUsageTrend computes the rate of change in resource usage
```

```

func (r *ResourceMonitor) CalculateUsageTrend(history []ResourceSample) float64 {
    // TODO 1: Validate that history has at least 2 samples for trend calculation

    // TODO 2: Use linear regression or simple slope calculation across sample window

    // TODO 3: Calculate bytes-per-second or percentage-per-minute change rate

    // TODO 4: Handle edge cases (empty history, identical timestamps, etc.)

    // TODO 5: Return trend slope (positive=increasing, negative=decreasing, zero=stable)

    // Hint: Simple slope = (lastValue - firstValue) / (lastTime - firstTime)

    // Hint: Convert time difference to seconds for rate calculation

}

```

Milestone Checkpoint

After implementing the Resource Control Component:

Test Command: `go test ./internal/cgroups/... -v`

Expected Behavior:

- Cgroup creation succeeds for valid container specifications
- Memory limits are enforced when containers exceed allocation
- CPU throttling occurs when containers exceed CPU quotas
- Resource usage monitoring returns accurate utilization statistics
- OOM events are detected and logged appropriately

Manual Verification:

1. Create a test container with 100MB memory limit: `echo 'stress --vm 1 --vm-bytes 200M' | ./container-runtime run --memory 104857600 alpine`
2. Container should be killed when memory usage exceeds limit
3. Check cgroup files: `cat /sys/fs/cgroup/container-runtime/[container-id]/memory.max` should show 104857600
4. Monitor resource usage: `./container-runtime stats [container-id]` should show current memory/CPU utilization
5. Verify cleanup: after container removal, cgroup directory should be deleted

Signs of Problems:

- "Operation not permitted" when writing to cgroup files → Check that cgroups v2 is enabled and process has sufficient privileges

- Memory limit not enforced → Verify memory.max file contains correct limit and memory.oom.group is enabled
- Resource monitoring returns zero values → Check cgroup file permissions and paths
- Cgroup removal fails → Ensure all processes are terminated before attempting removal

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Container uses unlimited memory despite limits	Memory limits not applied or enforced	Check memory.max file exists and contains expected value	Verify cgroups v2 enabled, ensure memory controller active
CPU usage exceeds configured quotas	CPU limits not configured or incorrect quota calculation	Check cpu.max file and verify quota/period values	Convert nanoseconds correctly to microsecond quota
Cgroup creation fails with permission denied	Insufficient privileges or cgroups v2 not available	Check /sys/fs/cgroup mount and user permissions	Run with appropriate privileges or enable user namespaces
Resource monitoring returns stale data	Cgroup files not updating or monitoring stopped	Check if container process still exists and cgroup populated	Verify process assignment and restart monitoring
OOM kills not detected	Event monitoring not working or wrong file format	Read memory.events manually and check counter values	Implement proper event file parsing and counter tracking

Filesystem Layering Component

Milestone(s): Milestone 3 (Overlay Filesystem) - This section implements image layering using overlayfs for copy-on-write filesystem support with proper layer management and cleanup.

Mental Model: Transparent Sheets

Before diving into the technical complexities of overlayfs, imagine you're working with a stack of transparent sheets on an overhead projector. Each sheet represents a filesystem layer that contains some files and directories. When you stack multiple sheets on top of each other, the audience sees a combined view where content from all sheets appears merged together.

The **bottom sheets** are your base image layers - they contain the operating system files, libraries, and application dependencies. These sheets are read-only; you never modify them directly. You might have a base

Ubuntu sheet, then an application runtime sheet on top of it, then your application code sheet on top of that.

The **top sheet** is special - it's a blank, writable transparency where you can draw new content or place opaque stickers to "hide" content from the sheets below. When you write a file in the container, it's like drawing on this top sheet. When you modify an existing file, it's like placing an opaque sticker over the old content and writing the new content on the sticker.

This analogy captures the essence of **copy-on-write semantics**: the original sheets (base image layers) remain untouched, but the combined view shows your modifications. If you need to "edit" a file that exists in a lower layer, the filesystem copies it up to the writable layer first, then modifies the copy. The original remains pristine in the lower layer, but the upper copy masks it in the merged view.

The **projector** itself represents the overlayfs mount that combines all these layers into a single, coherent filesystem tree that the container process sees as its root filesystem. Just as the audience sees one combined image despite the multiple sheets, the container sees one unified filesystem despite the multiple underlying layers.

Layer Stacking and Management

The core challenge in filesystem layering is **orchestrating the overlayfs mount** to combine multiple read-only image layers with a single writable upper layer. This process involves careful directory structure setup, mount option configuration, and proper handling of layer ordering.

Decision: Directory Structure for OverlayFS

- **Context:** OverlayFS requires specific directory layout with lower directories, upper directory, work directory, and mount target
- **Options Considered:**
 1. Flat structure with all directories in same parent
 2. Nested structure mirroring image layer hierarchy
 3. Separate trees for layers, upper, work, and mounts
- **Decision:** Separate trees organized by function under container storage root
- **Rationale:** Cleaner separation of concerns, easier cleanup, better debugging visibility, matches Docker/containerd patterns
- **Consequences:** More directory traversal but clearer organization and easier troubleshooting

Directory Type	Purpose	Example Path	Permissions	Cleanup Timing
Lower Layers	Read-only image layer content	/var/lib/runtimes/layers/sha256:abc123...	755 (readable)	On image removal
Upper Directory	Container's writable layer	/var/lib/runtimes/containers/{id}/upper	755 (writable)	On container removal
Work Directory	OverlayFS internal operations	/var/lib/runtimes/containers/{id}/work	755 (writable)	On container removal
Merged Mount	Combined view for container	/var/lib/runtimes/containers/{id}/merged	755 (mount target)	On container stop

The **layer stacking algorithm** follows a precise sequence to ensure proper overlay mounting:

1. **Validate layer prerequisites** by checking that each lower layer directory exists and contains valid filesystem content. Empty layers are permitted but must have proper directory structure.
2. **Create container-specific directories** for the upper layer, work directory, and merge mount point. The work directory must be on the same filesystem as the upper directory to support atomic operations.
3. **Construct the lowerdir mount option** by joining all image layer paths with colon separators, ordered from most-specific to most-general (reverse chronological order of layer creation). This ordering ensures that files in later layers override files in earlier layers.
4. **Prepare mount options** combining lowerdir, upperdir, workdir, and any additional overlayfs-specific flags like index=on for better performance on supported kernels.
5. **Execute the overlay mount** using the mount system call with MS_MGC_VAL flag and "overlay" filesystem type, binding the merged view to the container's mount point.
6. **Verify mount success** by checking that the merged directory shows expected content from all layers and that test write operations succeed in creating files that appear only in the upper directory.

The critical insight here is that overlayfs requires the work directory to be empty and on the same filesystem as the upper directory. Many mount failures stem from violating these requirements.

Layer ordering is crucial for proper file precedence. When multiple layers contain the same file path, overlayfs uses the **first match wins** principle, searching from the upper directory down through the lower

directories in the order specified. This means the mount option construction must reverse the chronological layer order:

Layer Type	Creation Order	Mount Order	Precedence
Upper (container changes)	N/A	First (implicit)	Highest
App layer	3rd (most recent)	2nd in lowerdir	Second highest
Runtime layer	2nd	3rd in lowerdir	Third highest
Base OS layer	1st (oldest)	Last in lowerdir	Lowest

Container filesystem manager coordinates the entire layer stacking process through a well-defined interface:

Method	Parameters	Returns	Description
PrepareContainerFilesystem	containerID string, imageLayers []string	error	Creates directories and mounts overlay
MountOverlayFS	layers []string, target string	error	Core overlay mount operation
GetMergedPath	containerID string	string	Returns path to merged filesystem
GetUpperPath	containerID string	string	Returns path to writable layer
ListLayerContent	layerPath string	[]FileInfo, error	Inspects layer contents for debugging
ValidateLayerIntegrity	layerPath string	error	Verifies layer filesystem consistency

Copy-on-Write Semantics

Copy-on-write (CoW) is the fundamental mechanism that makes container filesystem layering efficient and safe. When a container process attempts to **modify a file that exists in a lower read-only layer**, overlays automatically copies the file to the upper writable layer before allowing the modification to proceed. This ensures that the original file in the base image remains unchanged while the container sees its modified version.

The **CoW trigger conditions** determine when overlayfs initiates a copy operation:

Operation Type	Triggers CoW	Behavior	Performance Impact
Read existing file	No	Direct read from lower layer	Minimal overhead
Write new file	No	Write directly to upper layer	Minimal overhead
Modify existing file	Yes	Copy to upper, then modify	One-time copy cost
Delete existing file	Yes	Create whiteout in upper layer	Minimal overhead
Change permissions	Yes	Copy to upper, then change attrs	One-time copy cost
Create hard link	Yes	Copy target to upper if needed	Depends on target size

Whiteout files are overlayfs's mechanism for handling file deletions without modifying lower layers. When a container deletes a file that exists in a lower layer, overlayfs creates a special **character device file** with device number 0/0 in the upper layer at the same path. This whiteout file masks the lower layer file in the merged view, making it appear deleted to the container while preserving the original in the base image.

The **copy-up process** follows a specific algorithm to ensure atomicity and consistency:

1. **Create temporary file** in the upper layer with a unique name (typically the target filename with a random suffix) to avoid conflicts during the copy operation.
2. **Copy file content** from the lower layer source to the temporary file in chunks, preserving all data exactly while handling large files efficiently.
3. **Copy metadata** including permissions, ownership, timestamps, and extended attributes from the original file to maintain exact semantics.
4. **Atomic rename** the temporary file to the final target name in the upper layer, making the copied file visible in the merged view instantaneously.
5. **Verify copy integrity** by comparing file size and checksums between source and destination to detect any corruption during the copy process.

Key Design Insight: The copy-up operation is atomic from the container's perspective - the file either appears unchanged (before copy-up) or fully updated (after copy-up), never in a partial state.

Container process perspective shows how CoW remains transparent to applications running inside containers:

Container Action	Filesystem Events	Container Observes	Actual Storage
<code>cat /etc/passwd</code>	Read from lower layer	File contents	No upper layer copy
<code>echo "test" > /tmp/new</code>	Write to upper layer	New file created	File only in upper
<code>sed -i 's/bash/zsh/' /etc/passwd</code>	Copy-up, then modify	Modified file	Original in lower, copy in upper
<code>rm /bin/ls</code>	Create whiteout	File deleted	Original in lower, whiteout in upper
<code>ls /etc/</code>	Merge lower + upper - whiteouts	Combined directory	Files from all layers

Performance characteristics of copy-on-write operations vary significantly based on file size and access patterns:

File Size Range	Copy-up Latency	Storage Overhead	Optimization Strategy
< 4KB (small configs)	< 1ms	Minimal	Acceptable overhead
4KB - 1MB (libraries)	1-10ms	Low	Monitor for hot files
1MB - 100MB (binaries)	10ms - 1s	Moderate	Consider image layer optimization
> 100MB (databases)	> 1s	High	Use volume mounts instead

⚠ Pitfall: Large File Copy-up Performance A common mistake is placing large, frequently-modified files (like databases or log files) in the container's overlay filesystem. When the container first modifies a 500MB database file from the base image, overlayfs must copy the entire file to the upper layer before allowing any writes. This creates a several-second pause and doubles the storage requirement. **Solution:** Use volume mounts for large, mutable data files, keeping only application code and configuration in the overlay layers.

Layer Cleanup Process

Proper cleanup of overlay filesystem layers is critical for preventing resource leaks and maintaining system stability. The cleanup process must handle **partial failure scenarios** gracefully while ensuring that shared base image layers are only removed when no longer referenced by any containers.

Decision: Cleanup Ordering Strategy

- **Context:** Failed cleanup can leave mount points active, directories locked, or storage space consumed
- **Options Considered:**
 1. Aggressive cleanup that forces unmount and removes all directories
 2. Conservative cleanup that fails fast on any error
 3. Best-effort cleanup with detailed error reporting
- **Decision:** Best-effort cleanup with rollback capability
- **Rationale:** Maximizes successful cleanup while providing diagnostics for manual intervention when needed
- **Consequences:** More complex cleanup logic but better operational reliability

The **cleanup sequence** follows a carefully ordered set of operations designed to minimize the chance of leaving the system in an inconsistent state:

1. **Identify cleanup scope** by determining which overlay mount points, directories, and layer references belong to the container being removed.
2. **Unmount overlay filesystem** using the umount system call with MNT_DETACH flag if normal umount fails, allowing cleanup to proceed even if processes hold open file descriptors.
3. **Remove container-specific directories** including the merged mount point, upper directory, and work directory, which contain only this container's changes.
4. **Update layer reference counts** to track how many containers are using each base image layer, enabling safe cleanup of unused layers.
5. **Remove unreferenced image layers** only after confirming no other containers or cached images depend on them.
6. **Clean up mount namespace entries** to prevent stale mount information from accumulating in the kernel's mount table.

Cleanup Phase	Target	Error Handling	Rollback Action
Overlay unmount	/var/lib/runtime/containers/{id}/merged	Retry with MNT_DETACH	Mark for later cleanup
Upper directory	/var/lib/runtime/containers/{id}/upper	Log and continue	Manual removal required
Work directory	/var/lib/runtime/containers/{id}/work	Log and continue	Manual removal required
Mount point	/var/lib/runtime/containers/{id}/merged	Log and continue	Manual removal required
Layer references	Reference count files	Log and continue	May leak storage space
Image layers	/var/lib/runtime/layers/sha256:*	Skip if referenced	Preserve for safety

Reference counting prevents premature removal of shared base image layers that might be used by other containers:

Reference Type	Storage Location	Update Trigger	Cleanup Trigger
Container references	/var/lib/runtime/refs/containers/{id}	Container create/remove	Container removal
Image references	/var/lib/runtime/refs/images/{digest}	Image pull/remove	Image removal
Layer references	/var/lib/runtime/refs/layers/{digest}	Layer first use/last use	Reference count reaches zero

Cleanup verification ensures that the cleanup process completed successfully and didn't leave orphaned resources:

Verification Check	Expected State	Detection Method	Recovery Action
Mount points cleared	No overlay mounts for container	Parse <code>/proc/mounts</code>	Force unmount
Directories removed	Container directory tree gone	<code>stat()</code> system call	Recursive removal
References updated	Counts reflect removal	Read reference files	Recalculate from existing containers
Storage reclaimed	Disk space freed	<code>du</code> on upper directory	Manual space recovery

⚠ Pitfall: Unmount Failures with Open Files A frequent issue occurs when container cleanup attempts to unmount the overlay filesystem while processes still have open file descriptors to files in the merged view. This can happen if the container process didn't terminate cleanly or if background processes are still running.

Detection: The `umount` system call returns `EBUSY`. **Solution:** Use `MNT_DETACH` flag to create a lazy unmount that removes the mount from the namespace immediately but defers the actual cleanup until all references are closed.

⚠ Pitfall: Work Directory Reuse OverlayFS requires the work directory to be empty for each mount operation. If cleanup fails to remove the work directory completely, subsequent container creation will fail with cryptic mount errors. **Detection:** Mount fails with "work directory not empty" or similar message. **Solution:** Always verify work directory is empty before mounting, and remove any stale files if found.

Cleanup state machine tracks the progress of cleanup operations and enables recovery from partial failures:

Cleanup State	Description	Next States	Recovery Action
CleanupStarted	Cleanup initiated	OverlayUnmounted, CleanupFailed	Retry from beginning
OverlayUnmounted	Overlay filesystem unmounted	DirectoriesRemoved, CleanupFailed	Remove directories
DirectoriesRemoved	Container dirs removed	ReferencesUpdated, CleanupFailed	Update reference counts
ReferencesUpdated	Reference counts updated	LayersChecked, CleanupFailed	Check for unused layers
LayersChecked	Unused layers identified	CleanupCompleted, CleanupFailed	Remove unused layers
CleanupCompleted	All cleanup finished	N/A	Success
CleanupFailed	Error occurred	CleanupStarted	Manual intervention or retry

Implementation Guidance

Technology Recommendations:

Component	Simple Option	Advanced Option
Filesystem Operations	<code>syscall.Mount()</code> + <code>os.RemoveAll()</code>	<code>sys/mount</code> with detailed error handling
Directory Management	<code>os.MkdirAll()</code> + <code>filepath.Walk()</code>	Custom directory tree with atomic operations
Layer Storage	Simple directory hierarchy	Content-addressable storage with deduplication
Reference Counting	JSON files on disk	Embedded database (BoltDB)
Mount Verification	Parse <code>/proc/mounts</code>	Use <code>statfs()</code> and mount namespace APIs

Recommended File Structure:

```
internal/filesystem/
  manager.go           ← main filesystem manager
  overlay.go           ← overlayfs operations
  layers.go            ← layer management
  cleanup.go           ← cleanup operations
  references.go        ← reference counting
  manager_test.go      ← integration tests
  overlay_test.go      ← overlay-specific tests
  testdata/
    base-layer/
    app-layer/
    expected-merged/
```

Infrastructure Starter Code:

```
package filesystem

import (
    "fmt"

    "os"

    "path/filepath"

    "strings"

    "syscall"

    "encoding/json"

    "time"
)

// LayerInfo represents metadata about a filesystem layer

type LayerInfo struct {

    Digest      string      `json:"digest"`

    Size        int64       `json:"size"`

    CreatedAt   time.Time   `json:"created_at"`

    RefCount    int         `json:"ref_count"`

}

// FilesystemManager handles all overlay filesystem operations

type FilesystemManager struct {

    storageRoot    string

    layersDir      string

    containersDir string

    refsDir        string

}

// NewFilesystemManager creates a new filesystem manager
```

GO

```
func NewFilesystemManager(storageRoot string) (*FilesystemManager, error) {

    fm := &FilesystemManager{

        storageRoot:    storageRoot,
        layersDir:      filepath.Join(storageRoot, "layers"),
        containersDir:  filepath.Join(storageRoot, "containers"),
        refsDir:        filepath.Join(storageRoot, "refs"),
    }

    // Create required directory structure

    dirs := []string{fm.layersDir, fm.containersDir, fm.refsDir}

    for _, dir := range dirs {

        if err := os.MkdirAll(dir, 0755); err != nil {

            return nil, fmt.Errorf("failed to create directory %s: %v", dir, err)
        }
    }

    return fm, nil
}

// LayerExists checks if a layer is available locally

func (fm *FilesystemManager) LayerExists(digest string) bool {

    layerPath := filepath.Join(fm.layersDir, digest)

    _, err := os.Stat(layerPath)

    return err == nil
}

// GetLayerPath returns the filesystem path for a layer

func (fm *FilesystemManager) GetLayerPath(digest string) string {
```

```
    return filepath.Join(fm.layersDir, digest)

}

// GetContainerPaths returns all paths for a container's overlay setup

func (fm *FilesystemManager) GetContainerPaths(containerID string) (upper, work, merged
string) {

    containerDir := filepath.Join(fm.containersDir, containerID)

    return filepath.Join(containerDir, "upper"),
           filepath.Join(containerDir, "work"),
           filepath.Join(containerDir, "merged")
}

// ReferenceManager handles layer reference counting

type ReferenceManager struct {

    refsDir string
}

// AddContainerReference records that a container uses specific layers

func (rm *ReferenceManager) AddContainerReference(containerID string, layers []string)
error {

    refFile := filepath.Join(rm.refsDir, "containers", containerID+".json")

    if err := os.MkdirAll(filepath.Dir(refFile), 0755); err != nil {

        return err
    }

    data, err := json.Marshal(layers)

    if err != nil {

        return err
    }
}
```

```
    return os.WriteFile(refFile, data, 0644)

}

// RemoveContainerReference removes a container's layer references

func (rm *ReferenceManager) RemoveContainerReference(containerID string) error {

    refFile := filepath.Join(rm.refsDir, "containers", containerID+".json")

    return os.Remove(refFile)

}
```

Core Logic Skeleton Code:

```
// PrepareContainerFilesystem creates and mounts overlay filesystem for container      GO
func (fm *FilesystemManager) PrepareContainerFilesystem(containerID string, imageLayers []string) error {
    // TODO 1: Validate that all required image layers exist locally
    // Hint: Use LayerExists() for each layer digest

    // TODO 2: Create container-specific directories (upper, work, merged)
    // Hint: Use GetContainerPaths() and os.MkdirAll()

    // TODO 3: Ensure work directory is empty (overlayfs requirement)
    // Hint: Remove and recreate work directory if it exists

    // TODO 4: Add container reference to track layer usage
    // Hint: Use ReferenceManager.AddContainerReference()

    // TODO 5: Mount the overlay filesystem
    // Hint: Call MountOverlayFS() with properly ordered layers

    return nil
}

// MountOverlayFS mounts overlay filesystem with given layers and target
func (fm *FilesystemManager) MountOverlayFS(layers []string, target string) error {
    // TODO 1: Validate all layer directories exist and are accessible
    // Hint: Check each layer path with os.Stat()

    // TODO 2: Construct lowerdir option by joining layer paths with ":""
    // Hint: Use strings.Join() - order matters! Most recent layer first
```

```
// TODO 3: Build complete mount options string

// Format: "lowerdir=/path1:/path2,upperdir=/upper,workdir=/work"

// Hint: Get upper/work paths from target directory structure


// TODO 4: Execute overlay mount using syscall.Mount()

// Hint: syscall.Mount("overlay", target, "overlay", 0, options)


// TODO 5: Verify mount succeeded by checking target directory contents

// Hint: List files in target and compare with expected layer contents


return nil

}

// HandleCopyOnWrite demonstrates Cow behavior (for testing/debugging)

func (fm *FilesystemManager) HandleCopyOnWrite(containerID, filePath string) error {

    // TODO 1: Check if file exists in lower layers but not upper

    // Hint: Check merged view vs upper directory directly


    // TODO 2: If file is in lower layer, demonstrate that modification triggers copy-up

    // Hint: Write to file and verify copy appears in upper directory


    // TODO 3: Verify original file in lower layer remains unchanged

    // Hint: Compare checksums before and after modification


    return nil

}
```

```
// CleanupContainerFilesystem removes overlay mount and directories

func (fm *FilesystemManager) CleanupContainerFilesystem(containerID string) error {

    // TODO 1: Get container paths (upper, work, merged)

    // Hint: Use GetContainerPaths() method


    // TODO 2: Unmount overlay filesystem from merged directory

    // Hint: Use syscall.Unmount() with 0 flags, retry with MNT_DETACH if busy


    // TODO 3: Remove container-specific directories (upper, work, merged)

    // Hint: Use os.RemoveAll() for recursive removal


    // TODO 4: Remove container's layer references

    // Hint: Use ReferenceManager.RemoveContainerReference()


    // TODO 5: Check if any image layers can be cleaned up (reference count = 0)

    // Hint: Implement reference counting logic to identify unused layers


    return nil
}

// VerifyOverlayMount checks that overlay mount is working correctly

func (fm *FilesystemManager) VerifyOverlayMount(containerID string) error {

    // TODO 1: Check that merged directory is actually a mount point

    // Hint: Parse /proc/mounts or use statfs() system call


    // TODO 2: Verify that files from all layers are visible in merged view

    // Hint: List expected files and check they appear in merged directory
```

```
// TODO 3: Test that writes go to upper directory

// Hint: Create test file in merged view, verify it appears in upper only


// TODO 4: Test that whiteouts work for file deletion

// Hint: Delete a file from lower layer, verify whiteout created in upper


return nil

}
```

Language-Specific Hints (Go):

- Use `syscall.Mount("overlay", target, "overlay", 0, options)` for overlay mounting
- Handle `syscall.EBUSY` errors on unmount by retrying with `syscall.MNT_DETACH`
- Use `filepath.Join()` for cross-platform path construction
- Check mount success by parsing `/proc/mounts` or using `unix.Statfs()`
- Use `os.MkdirAll()` with 0755 permissions for directory creation
- Implement proper cleanup with `defer` statements for resource management

Milestone Checkpoint:

After implementing the overlay filesystem component:

1. **Test layer mounting:** Run `go test -v ./internal/filesystem -run TestOverlayMount`
2. **Expected behavior:** Container should see merged view of all image layers
3. **Manual verification:**

```

# Check overlay mount exists

mount | grep overlay

# Verify upper directory captures changes

echo "test" > /path/to/merged/testfile

ls /path/to/upper/ # should show testfile

# Verify lower layers unchanged

ls /path/to/layer1/ # original contents preserved

```

BASH

4. Signs of trouble:

Mount failures usually indicate work directory issues or invalid layer paths

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
"Invalid argument" on mount	Work directory not empty or wrong filesystem	Check <code>ls -la workdir/</code> and <code>df workdir/</code>	Remove work directory contents, ensure same filesystem as upper
"Device busy" on unmount	Processes holding open files	Use <code>lsof +D /merged/path</code> to find processes	Kill processes or use MNT_DETACH for lazy unmount
Files missing in merged view	Wrong lowerdir order or missing layers	Check mount options in <code>/proc/mounts</code>	Verify layer paths exist and fix ordering
CoW not working	Upper directory permissions or filesystem issues	Test write to merged view, check if file appears in upper	Fix upper directory permissions (755)
Storage space not freed	Cleanup didn't remove upper directory	Check disk usage with <code>du -sh /var/lib/runtime/containers/</code>	Manual cleanup of container directories

Network Management Component

Milestone(s): Milestone 4 (Container Networking) - This section implements bridge networking for containers with port mapping and inter-container communication using veth pairs, bridge networks, and NAT rules.

Container networking transforms isolated processes into communicating services by creating virtual network infrastructure that mimics physical networking concepts. The network management component orchestrates multiple Linux kernel networking features to provide containers with network interfaces, IP addresses, routing capabilities, and connectivity to both other containers and the external world. This component must handle the complexity of bridging multiple network namespaces while maintaining proper isolation boundaries and enabling controlled communication pathways.

The fundamental challenge in container networking lies in the paradox of isolation versus connectivity. Containers need network isolation for security and resource management, but they also need selective connectivity for inter-service communication and external access. Our network management component resolves this paradox by creating controlled communication channels through virtual networking infrastructure, allowing containers to communicate as if they were separate physical machines connected through network switches and routers.

Mental Model: Virtual Cable System

Think of container networking like building a virtual office building with separate apartments (containers) that need telephone and internet connectivity. Each apartment starts completely isolated with no communication ability whatsoever. To provide connectivity, we run virtual cables and install virtual network equipment.

The **veth pair** acts like a special two-way cable with connectors on both ends. We plug one end into the apartment (container network namespace) and feed the other end through the wall into the building's main network room (host network namespace). This cable carries all network traffic bidirectionally between the apartment and the building's network infrastructure.

The **bridge** functions like a network switch in the building's main network room. All the cables coming from different apartments plug into this switch, allowing apartments to communicate with each other. The switch learns which apartment is connected to which port and intelligently forwards messages between apartments.

The **IP address assignment** works like assigning apartment numbers from a predetermined numbering scheme. Each apartment gets a unique address within the building's addressing plan (subnet), ensuring messages can be properly routed to the correct destination.

The **Port forwarding** acts like a receptionist service that forwards calls from the building's main phone line to specific apartment extensions. When someone calls the building from outside and asks for a specific service, the receptionist (NAT rules) knows which apartment provides that service and forwards the call appropriately.

This mental model helps understand that container networking creates a complete virtual networking infrastructure that mirrors physical networking concepts, allowing isolated containers to communicate in controlled and predictable ways.

Veth Pair and Bridge Setup

The foundation of container networking rests on creating **virtual ethernet pairs (veth pairs)** that connect isolated network namespaces to shared network infrastructure. A veth pair functions as a bidirectional pipe where packets written to one end immediately appear at the other end, enabling communication across namespace boundaries. The network management component must orchestrate the creation, configuration, and cleanup of these virtual network devices along with bridge networking that enables multi-container communication.

The veth pair creation process begins by generating unique interface names to avoid conflicts with existing network devices. The component creates the veth pair in the host network namespace, then moves one end into the target container's network namespace while keeping the other end in the host namespace. This asymmetric placement allows the host to manage connectivity while providing the container with an isolated network interface.

Decision: Veth Pair Naming Strategy

- **Context:** Veth interfaces need unique names to avoid conflicts, and names must be trackable for cleanup operations
- **Options Considered:** Random names, UUID-based names, container-ID-based names
- **Decision:** Use container ID prefix with interface type suffix (e.g., "veth-container123-host" and "veth-container123-container")
- **Rationale:** Container ID prefixes provide clear ownership mapping for cleanup operations while remaining human-readable for debugging. The host/container suffix clarifies which end of the pair each interface represents.
- **Consequences:** Enables efficient cleanup by interface name pattern matching, simplifies debugging by making interface ownership obvious, but limits container ID length due to Linux interface name restrictions.

After creating the veth pair, the component must configure each interface with appropriate settings. The container-side interface receives an IP address from the configured subnet pool, while the host-side interface remains addressless but gets attached to the container bridge. Both interfaces must be brought up (activated) to enable packet flow, and the container interface typically becomes the default gateway interface within the container's network namespace.

The bridge setup process creates or reuses a shared bridge device that functions as a virtual network switch for container communication. The bridge maintains a MAC address learning table that maps container MAC

addresses to specific ports, enabling efficient packet forwarding between containers. All host-side veth interfaces connect to this bridge, creating a shared layer-2 network segment for container communication.

Veth Pair Creation Algorithm:

1. Generate unique interface names using container ID and interface type suffixes to ensure no naming conflicts with existing network devices
2. Create the veth pair in the host network namespace using the netlink library or ip command with both interfaces initially in the host namespace
3. Move the container-side interface into the target container's network namespace using the setns system call or netlink operations
4. Configure the container-side interface with an assigned IP address and appropriate subnet mask from the container network's address pool
5. Set the container-side interface as the active default route within the container's network namespace for external connectivity
6. Bring up both interfaces (host-side and container-side) to enable packet forwarding and establish the communication channel
7. Attach the host-side interface to the container bridge, registering it as a bridge port for inter-container communication
8. Configure MAC address learning and forwarding rules on the bridge port to enable proper packet switching between containers
9. Store interface names and configuration details in the container state for cleanup operations when the container terminates

The bridge configuration requires careful attention to forwarding policies and security settings. The bridge must enable packet forwarding between attached ports while maintaining isolation from the host's main network interfaces unless explicitly configured otherwise. Bridge forwarding delays and learning timeouts affect container communication latency and convergence time for network topology changes.

Veth Configuration Parameter	Host Side Value	Container Side Value	Purpose
Interface Name	veth-{containerID}-host	veth-{containerID}-container	Unique identification and cleanup tracking
IP Address	None (bridge attached)	Assigned from subnet pool	Container network identity
Subnet Mask	N/A	/24 (or configured subnet)	Network boundary definition
MTU Size	1500 (or bridge MTU)	1500 (matches host side)	Maximum packet size consistency
MAC Address	Auto-generated	Auto-generated	Layer-2 addressing
Administrative State	UP	UP	Interface activation
Bridge Attachment	Yes (container bridge)	No (not applicable)	Inter-container communication
Default Route	No	Yes (via bridge IP)	External connectivity

The critical insight here is that veth pairs create a controlled breach in network namespace isolation. While containers remain isolated at the process and filesystem levels, the veth pair provides a precisely managed communication channel that enables networking while preserving other isolation boundaries.

Bridge Management Algorithm:

1. Check if the container bridge already exists, creating it if necessary with appropriate forwarding and learning settings
2. Configure bridge IP address and subnet if this is the first container using the bridge, establishing the gateway address for container routing
3. Enable IP forwarding on the bridge interface to allow packets to flow between bridge ports and external networks
4. Set bridge forwarding delay and aging time parameters to optimize convergence time and memory usage for container networking patterns
5. Attach the host-side veth interface as a bridge port, enabling the container to participate in the shared layer-2 network
6. Configure port-specific settings like learning, flooding, and STP participation based on security and performance requirements
7. Update bridge forwarding database with static entries if required for specific security or performance optimizations

8. Enable or disable bridge features like VLAN filtering, multicast snooping, and spanning tree protocol based on deployment requirements

The component must handle bridge lifecycle management, creating bridges on demand and cleaning them up when no containers remain attached. Bridge cleanup requires careful ordering to avoid leaving orphaned network devices or disrupting communication for remaining containers.

IP Address Assignment

Container IP address assignment requires maintaining a pool of available addresses within a configured subnet while ensuring uniqueness and preventing conflicts with host network addressing. The network management component implements an IP address manager that tracks allocations, handles assignment and release operations, and persists address assignments across container restarts to maintain network identity stability.

The IP address pool operates within a designated subnet range that must not conflict with the host's network configuration or other container runtime instances. The component typically reserves the first address in the subnet for the bridge gateway, the last address for broadcast, and manages the remaining addresses as an assignable pool. Address assignment follows a deterministic or pseudo-random strategy to balance predictability with conflict avoidance.

Decision: IP Address Assignment Strategy

- **Context:** Containers need unique IP addresses from a managed pool, with consideration for restart stability and conflict avoidance
- **Options Considered:** Sequential assignment, hash-based deterministic assignment, random assignment with conflict detection
- **Decision:** Implement hash-based deterministic assignment with fallback to sequential search for conflicts
- **Rationale:** Hash-based assignment using container ID provides stable addresses across container restarts, improving debugging and configuration consistency. Sequential fallback ensures assignment success even with hash collisions.
- **Consequences:** Enables predictable container addressing for development and testing scenarios, but requires careful hash distribution to avoid clustering in the address space.

The address assignment process begins by computing a candidate address using a hash function applied to the container ID, then checking for conflicts with existing assignments. If the candidate address is available, the component reserves it and associates it with the container in the persistent address allocation table. If conflicts occur, the component falls back to sequential search from the candidate address to find the next available address.

Address persistence ensures that containers receive the same IP address across restarts when possible, maintaining network identity for services that depend on stable addressing. The component stores address

assignments in a persistent data structure that survives runtime restarts and provides atomic allocation and release operations to prevent corruption during concurrent operations.

IP Assignment State	Container ID	Assigned IP	Allocation Time	Lease Expiry	Status
Active	container-web-01	172.16.0.10	2024-01-15T10:30:00Z	Never	Running
Active	container-db-01	172.16.0.11	2024-01-15T10:31:00Z	Never	Running
Reserved	container-web-02	172.16.0.12	2024-01-15T10:32:00Z	2024-01-15T11:32:00Z	Stopped
Released	N/A	172.16.0.13	N/A	N/A	Available

IP Address Assignment Algorithm:

1. Load the current address allocation table from persistent storage to understand which addresses are currently assigned or reserved
2. Compute candidate IP address by applying a hash function to the container ID, mapping to the configured subnet address range
3. Check if the candidate address is available in the allocation table, considering both active assignments and temporary reservations
4. If candidate is available, reserve the address for this container and update the allocation table with atomic write operations
5. If candidate is unavailable, perform sequential search starting from candidate address to find the next available address in the subnet
6. Validate that the selected address doesn't conflict with reserved addresses (gateway, broadcast, network) or host interface addresses
7. Create a lease record with container ID, assigned address, allocation timestamp, and lease expiration policy for cleanup operations
8. Configure the container's network interface with the assigned address, subnet mask, and default gateway pointing to the bridge
9. Update the allocation table with the final assignment and ensure changes are persisted to storage before returning success

The component implements address conflict detection by maintaining both active assignments (for running containers) and reserved assignments (for recently stopped containers). Reserved assignments have expiration times that allow address reuse while providing stability for containers that restart quickly. This two-tier approach balances address space efficiency with operational convenience.

Address release requires careful cleanup to avoid premature reuse that could cause network conflicts. When containers stop, their addresses transition to reserved status with configurable lease expiration times. After lease expiration, addresses return to the available pool. The component implements background cleanup processes that reclaim expired addresses and compact the allocation table.

Subnet Pool Configuration:

Parameter	Default Value	Description	Constraints
Subnet CIDR	172.16.0.0/24	Network range for container addresses	Must not conflict with host routes
Gateway Address	172.16.0.1	Bridge IP address for container default route	First usable address in subnet
Address Pool Start	172.16.0.10	First address available for container assignment	After gateway and reserved addresses
Address Pool End	172.16.0.250	Last address available for container assignment	Before broadcast and reserved addresses
Lease Duration	1 hour	Time to hold reserved addresses after container stop	Balance stability vs efficiency
Allocation Strategy	Hash + Sequential	Method for selecting candidate addresses	Deterministic with conflict resolution

The network management component must coordinate IP address assignments with DNS configuration when containers require hostname resolution. Address assignments can trigger DNS record creation in internal DNS servers, enabling containers to resolve each other by hostname in addition to IP address communication.

Port Forwarding with NAT

Port forwarding enables external clients to access services running inside containers by creating Network Address Translation (NAT) rules that redirect traffic from host ports to container ports. The network management component implements port forwarding using iptables rules that modify packet headers as traffic flows between the external network and container network namespaces. This capability transforms isolated container services into accessible network services while maintaining container isolation for non-forwarded ports.

The port forwarding mechanism operates by intercepting packets destined for specific host ports and redirecting them to corresponding container IP addresses and ports. The component creates iptables rules in the NAT table that perform Destination Network Address Translation (DNAT), changing the destination address from the host IP to the container IP while preserving the original source information for proper return routing.

Port forwarding configuration requires careful coordination between the host firewall rules, bridge forwarding policies, and container network configuration. The component must ensure that forwarded traffic can traverse all network boundaries while maintaining security restrictions for non-forwarded traffic. This involves creating rules in multiple iptables chains and tables to handle different phases of packet processing.

Decision: NAT Rule Management Strategy

- Context:** Container port forwarding requires iptables rules that must be created atomically and cleaned up reliably without interfering with other system rules
- Options Considered:** Direct iptables rule manipulation, custom chain isolation, netfilter hooks programmatic control
- Decision:** Use custom iptables chains for container rules with atomic rule set replacement
- Rationale:** Custom chains provide isolation from system rules, reducing the risk of conflicts. Atomic replacement ensures consistent rule states during updates. Chain-based organization simplifies bulk cleanup operations.
- Consequences:** Enables safe concurrent rule management and reliable cleanup, but requires understanding of iptables chain traversal and rule precedence for effective debugging.

The component creates dedicated iptables chains for container networking rules, avoiding conflicts with system-level firewall policies. Container-specific DNAT rules are added to these custom chains, with jump rules in the standard chains directing relevant traffic through the container rule processing. This architecture enables atomic updates and complete cleanup of container networking rules without affecting other system networking configuration.

Port forwarding rules must handle both TCP and UDP protocols with appropriate connection tracking and state management. The component creates separate rules for each protocol type, ensuring that connection state tracking works correctly for bi-directional communication. Return traffic must be properly handled through connection tracking to maintain the appearance of direct communication between external clients and container services.

Port Mapping Configuration	Host Port	Container IP	Container Port	Protocol	Rule Priority
Web Service	8080	172.16.0.10	80	TCP	100
Database Access	5432	172.16.0.11	5432	TCP	100
DNS Service	53	172.16.0.12	53	UDP	100
SSH Access	2222	172.16.0.10	22	TCP	100
Custom API	9000	172.16.0.13	3000	TCP	100

Port Forwarding Setup Algorithm:

1. Validate port mapping configuration to ensure host ports are not already in use by other services or containers
2. Create container-specific iptables chain names to isolate rules and enable atomic cleanup operations when containers are removed
3. Add jump rules to standard iptables chains (PREROUTING, FORWARD) that direct container traffic to the custom chains for processing
4. Create DNAT rules in the PREROUTING chain that change destination address from host IP to container IP for specified ports
5. Create FORWARD rules that allow traffic to flow from external interfaces to the container bridge for forwarded ports only
6. Enable connection tracking for return traffic by creating corresponding rules in the POSTROUTING chain for source NAT if required
7. Configure bridge forwarding rules to allow traffic destined for container IP addresses to traverse the bridge interface correctly
8. Test connectivity by verifying that packets can flow bidirectionally between external clients and container services through the NAT rules
9. Store port mapping configuration in container state for proper cleanup when the container is stopped or removed

The component must handle port conflict detection to prevent multiple containers from attempting to bind the same host port. Port allocation tracking maintains a registry of active port forwarding rules, preventing conflicts during container creation and enabling proper cleanup during container removal.

Connection tracking configuration affects the performance and reliability of port forwarding rules. The component configures netfilter connection tracking parameters to handle the expected connection volume and timeout characteristics of container services. Proper connection tracking ensures that established connections continue working even if iptables rules are temporarily modified during container updates.

IPTables Rule Structure:

The network management component creates a structured set of iptables rules organized into logical groups for maintainability and atomic updates:

Chain Name	Table	Purpose	Rule Examples
CONTAINER-DNAT	nat	Destination address translation for inbound traffic	<code>-p tcp --dport 8080 -j DNAT --to-destination 172.16.0.10:80</code>
CONTAINER-FORWARD	filter	Traffic forwarding authorization for container access	<code>-d 172.16.0.10 -p tcp --dport 80 -j ACCEPT</code>
CONTAINER-POSTROUTING	nat	Source address translation for outbound traffic if needed	<code>-s 172.16.0.0/24 ! -d 172.16.0.0/24 -j MASQUERADE</code>

Port forwarding cleanup requires removing all related iptables rules in the correct order to avoid temporary connectivity disruption or rule inconsistencies. The component implements a cleanup algorithm that removes rules from chains in dependency order, then removes custom chains once they are empty. This ensures that no orphaned rules remain that could interfere with future container networking operations.

⚠ Pitfall: Persistent IPTables Rules

A common mistake is failing to properly clean up iptables rules when containers are removed, leading to accumulated rule sets that can cause port conflicts and security vulnerabilities. The issue manifests as "port already in use" errors when creating new containers, or unexpected traffic forwarding to non-existent containers.

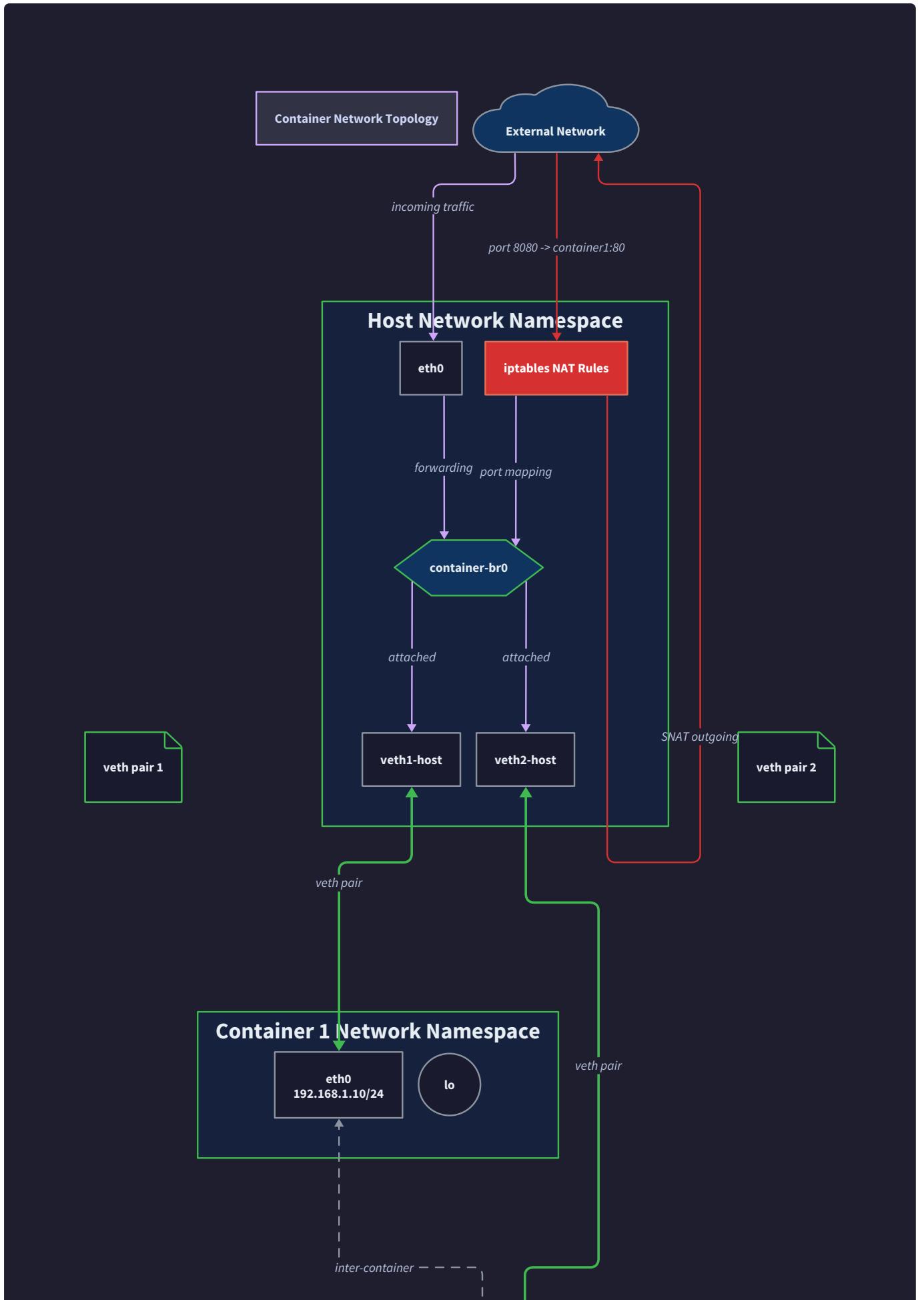
The problem occurs because iptables rules persist independently of the processes that created them. Even if a container stops or the container runtime crashes, the NAT and forwarding rules remain active in the kernel. This leads to several problematic scenarios: host ports appear occupied when they're not, traffic gets dropped because target containers no longer exist, and rule accumulation degrades networking performance.

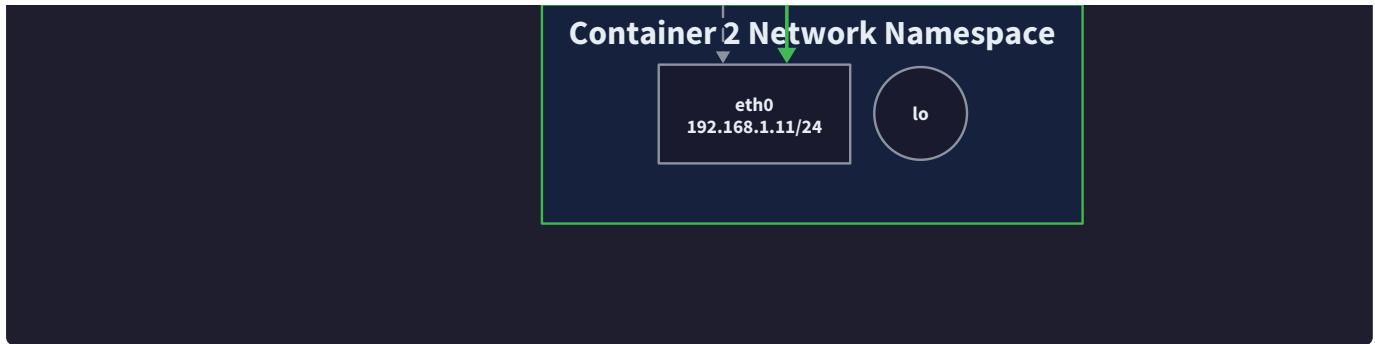
The solution requires implementing robust cleanup procedures that execute during both normal container shutdown and runtime recovery operations. The component must maintain accurate records of which iptables rules belong to which containers, and implement cleanup verification that ensures rules are actually removed. Additionally, the runtime should perform rule audit operations on startup to clean up orphaned rules from previous runtime instances.

Advanced Port Forwarding Features:

The network management component can extend basic port forwarding with additional features that improve usability and security:

Feature	Implementation Approach	Benefits	Complexity
Dynamic Port Allocation	Maintain pool of available host ports, assign automatically	Reduces port conflicts, simplifies container deployment	Medium
Port Range Forwarding	Create rules for contiguous port ranges rather than individual ports	Supports applications that use multiple ports	Medium
Source IP Filtering	Add source address restrictions to DNAT rules	Enhances security by limiting access origins	Low
Load Balancing	Distribute traffic across multiple container instances	Improves service availability and performance	High





The network topology diagram illustrates how these networking components work together to create a complete container networking solution. Containers receive isolated network namespaces connected through veth pairs to a shared bridge, with iptables NAT rules enabling external access to container services while maintaining isolation boundaries.

Common Pitfalls

⚠ Pitfall: Veth Interface Cleanup Order

A frequent mistake involves deleting the host-side veth interface before properly shutting down the container process, causing network connectivity loss and potential container hanging. When the host-side veth interface disappears while the container is still running, the container loses all network connectivity including the ability to shut down gracefully if it needs to communicate with external services during shutdown.

This problem manifests as containers that appear to stop responding during shutdown sequences, taking the full shutdown timeout before being forcefully terminated. The issue occurs because the container's network interface becomes unusable when its paired host interface is removed, but the container may not detect this failure immediately and continues attempting network operations.

The solution requires implementing proper shutdown sequencing where container processes are terminated first, allowing them to complete network-dependent shutdown operations before their network interfaces are removed. The component should wait for container process termination before cleaning up veth interfaces, and implement timeout handling for containers that fail to shut down gracefully within reasonable time limits.

⚠ Pitfall: Bridge MTU Misconfiguration

Incorrectly configured Maximum Transmission Unit (MTU) sizes between veth interfaces and the container bridge cause packet fragmentation and connectivity problems that are difficult to diagnose. When veth interfaces have different MTU sizes than the bridge they connect to, large packets get fragmented or dropped, leading to inconsistent connectivity where small requests work but large data transfers fail.

The problem typically appears as applications that work for small operations but fail when transferring larger amounts of data. Web applications might load basic pages but fail when uploading files or receiving large API responses. The issue is particularly confusing because basic connectivity tests succeed while real-world usage fails unpredictably.

The solution requires ensuring MTU consistency across all components in the network path: host interfaces, bridge interfaces, and container veth interfaces must all use compatible MTU sizes. The component should

detect the host network MTU and configure all container networking components to use the same or smaller MTU values, avoiding fragmentation issues.

Pitfall: IP Address Pool Exhaustion

Failing to implement proper address pool management leads to IP address exhaustion when containers are created and destroyed frequently, especially during development and testing scenarios. The pool exhaustion occurs when addresses remain marked as "in use" after containers are removed, eventually preventing new container creation even though no containers are actually using the addresses.

This problem manifests as container creation failures with "no available IP addresses" errors, even when few or no containers are currently running. The issue compounds over time as the pool becomes increasingly fragmented with unusable address reservations. Development workflows that create and destroy many containers quickly are particularly susceptible to this problem.

The solution requires implementing robust address lifecycle management with proper cleanup procedures and lease expiration handling. The component must track address assignments accurately, implement cleanup verification to ensure addresses are actually released when containers stop, and provide administrative tools to manually reclaim orphaned address allocations when automatic cleanup fails.

Pitfall: IPTables Rule Conflicts

Creating iptables rules without considering existing system firewall configuration can cause rule conflicts that break both container networking and host system connectivity. The conflicts arise when container networking rules interfere with system security policies, or when system changes modify the iptables configuration in ways that break container connectivity.

This problem appears as intermittent connectivity failures that depend on the specific system firewall configuration and the order of rule evaluation. Containers might lose network access after system updates that modify firewall rules, or system services might become inaccessible when container networking rules interfere with host networking policies.

The solution requires implementing defensive rule management that uses custom iptables chains to isolate container networking rules from system rules. The component should check for existing rule conflicts before creating new rules, use rule priorities that don't interfere with system security policies, and implement rule verification procedures that detect when external changes break container networking functionality.

Implementation Guidance

The network management component requires careful integration with Linux networking APIs and external tools to create and manage virtual network infrastructure. This component represents the most complex aspect of container runtime development due to its dependencies on kernel networking features and interaction with system-level network configuration.

Technology Recommendations:

Component	Simple Option	Advanced Option
Netlink Communication	<code>vishvananda/netlink</code> Go library	Direct netlink socket programming
IP Address Management	File-based JSON persistence	Embedded database (BoltDB/BadgerDB)
IPTables Management	<code>coreos/go-iptables</code> wrapper library	Direct iptables binary execution
Network Interface Creation	<code>ip</code> command execution via <code>os/exec</code>	Pure netlink API calls
Bridge Management	Linux bridge utilities + commands	Programmatic netlink bridge control

Recommended File Structure:

```

container-runtime/
  internal/network/
    manager.go           ← main network management orchestrator
    veth.go              ← veth pair creation and management
    bridge.go             ← bridge networking setup and control
    ipam.go               ← IP address allocation and management
    nat.go                ← port forwarding and iptables rules
    types.go              ← network-related data structures
    manager_test.go       ← comprehensive networking integration tests
  internal/netutil/
    netlink.go            ← netlink communication helpers
    iptables.go           ← iptables rule management utilities
    validation.go          ← network configuration validation
  configs/
    network-default.json   ← default network configuration

```

Infrastructure Starter Code (Complete Network Utilities):

GO

```
// Package netutil provides low-level networking utilities for container runtime

package netutil

import (
    "fmt"
    "net"
    "os/exec"
    "strconv"
    "strings"
    "github.com/vishvananda/netlink"
    "github.com/coreos/go-iptables/iptables"
)

// NetworkConfig defines network configuration for containers

type NetworkConfig struct {
    BridgeName     string   `json:"bridge_name"`
    SubnetCIDR    string   `json:"subnet_cidr"`
    GatewayIP     string   `json:"gateway_ip"`
    DNSServers    []string `json:"dns_servers"`
    IPTablesChain string   `json:"iptables_chain"`
}

// IPTablesManager wraps iptables operations with error handling

type IPTablesManager struct {
    ipt *iptables.IPTables
}

// NewIPTablesManager creates a new iptables manager instance

func NewIPTablesManager() (*IPTablesManager, error) {
```

```
ipt, err := iptables.New()

if err != nil {

    return nil, fmt.Errorf("failed to initialize iptables: %w", err)
}

return &IPTablesManager{ipt: ipt}, nil
}

// CreateChain creates a custom iptables chain for container rules

func (m *IPTablesManager) CreateChain(table, chain string) error {

exists, err := m.ipt.ChainExists(table, chain)

if err != nil {

    return fmt.Errorf("failed to check chain existence: %w", err)
}

if !exists {

    if err := m.ipt.NewChain(table, chain); err != nil {

        return fmt.Errorf("failed to create chain %s: %w", chain, err)
    }
}
return nil
}

// AddRule adds an iptables rule with validation

func (m *IPTablesManager) AddRule(table, chain string, rulespec ...string) error {

exists, err := m.ipt.Exists(table, chain, rulespec...)
if err != nil {

    return fmt.Errorf("failed to check rule existence: %w", err)
}

if !exists {
```

```
    if err := m.ipt.Insert(table, chain, 1, rulespec...); err != nil {
        return fmt.Errorf("failed to add rule: %w", err)
    }
}

return nil
}

// RemoveRule removes an iptables rule with validation

func (m *IPTablesManager) RemoveRule(table, chain string, rulespec ...string) error {
    exists, err := m.ipt.Exists(table, chain, rulespec...)
    if err != nil {
        return fmt.Errorf("failed to check rule existence: %w", err)
    }
    if exists {
        if err := m.ipt.Delete(table, chain, rulespec...); err != nil {
            return fmt.Errorf("failed to remove rule: %w", err)
        }
    }
    return nil
}

// NetlinkHelper provides higher-level netlink operations

type NetlinkHelper struct{}


// CreateVethPair creates a veth pair with specified names

func (n *NetlinkHelper) CreateVethPair(hostName, containerName string) error {
    hostVeth := &netlink.Veth{
        LinkAttrs: netlink.LinkAttrs{Name: hostName},

```

```
    PeerName: containerName,
}

if err := netlink.LinkAdd(hostVeth); err != nil {
    return fmt.Errorf("failed to create veth pair: %w", err)
}

return nil
}

// MoveInterfaceToNamespace moves network interface to target namespace

func (n *NetlinkHelper) MoveInterfaceToNamespace(interfaceName string, pid int) error {
    link, err := netlink.LinkByName(interfaceName)

    if err != nil {
        return fmt.Errorf("failed to find interface %s: %w", interfaceName, err)
    }

    if err := netlink.LinkSetNsPid(link, pid); err != nil {
        return fmt.Errorf("failed to move interface to namespace: %w", err)
    }

    return nil
}

// ConfigureInterface sets IP address and brings interface up

func (n *NetlinkHelper) ConfigureInterface(interfaceName, ipAddr string) error {
    link, err := netlink.LinkByName(interfaceName)

    if err != nil {
        return fmt.Errorf("failed to find interface %s: %w", interfaceName, err)
    }
```

```
addr, err := netlink.ParseAddr(ipAddr)

if err != nil {
    return fmt.Errorf("failed to parse IP address %s: %w", ipAddr, err)
}

if err := netlink.AddrAdd(link, addr); err != nil {
    return fmt.Errorf("failed to add IP address: %w", err)
}

if err := netlink.LinkSetUp(link); err != nil {
    return fmt.Errorf("failed to bring interface up: %w", err)
}

return nil
}

// CreateBridge creates and configures a network bridge

func (n *NetlinkHelper) CreateBridge(bridgeName, bridgeIP string) error {
    bridge := &netlink.Bridge{
        LinkAttrs: netlink.LinkAttrs{Name: bridgeName},
    }

    if err := netlink.LinkAdd(bridge); err != nil {
        if !strings.Contains(err.Error(), "file exists") {
            return fmt.Errorf("failed to create bridge: %w", err)
        }
    }
}
```

```
    }

    if bridgeIP != "" {
        if err := n.ConfigureInterface(bridgeName, bridgeIP); err != nil {
            return fmt.Errorf("failed to configure bridge IP: %w", err)
        }
    }

    return nil
}

// AttachInterfaceToBridge attaches interface to bridge

func (n *NetlinkHelper) AttachInterfaceToBridge(interfaceName, bridgeName string) error {
    iface, err := netlink.LinkByName(interfaceName)

    if err != nil {
        return fmt.Errorf("failed to find interface %s: %w", interfaceName, err)
    }

    bridge, err := netlink.LinkByName(bridgeName)

    if err != nil {
        return fmt.Errorf("failed to find bridge %s: %w", bridgeName, err)
    }

    if err := netlink.LinkSetMaster(iface, bridge); err != nil {
        return fmt.Errorf("failed to attach interface to bridge: %w", err)
    }
}
```

```
    return nil

}

// ValidateNetworkConfig validates network configuration parameters

func ValidateNetworkConfig(config NetworkConfig) error {

    if config.BridgeName == "" {

        return fmt.Errorf("bridge name cannot be empty")

    }

    _, _, err := net.ParseCIDR(config.SubnetCIDR)

    if err != nil {

        return fmt.Errorf("invalid subnet CIDR: %w", err)

    }

    if net.ParseIP(config.GatewayIP) == nil {

        return fmt.Errorf("invalid gateway IP address: %s", config.GatewayIP)

    }

    return nil

}
```

Core Logic Skeleton Code:

```
// Package network implements container networking with veth pairs, bridges, and NAT      GO
package network

import (
    "context"
    "fmt"
    "net"
    "sync"
    "time"
    "container-runtime/internal/netutil"
)

// Manager orchestrates container networking operations

type Manager struct {
    config          netutil.NetworkConfig
    ipam           *IPAddressManager
    iptables       *netutil.IPTablesManager
    netlink        *netutil.NetlinkHelper
    mu             sync.RWMutex
    containers     map[string]*ContainerNetwork
}

// ContainerNetwork tracks networking state for a container

type ContainerNetwork struct {
    ContainerID      string
    VethHost        string
    VethContainer   string
    IPAddress       string
}
```

GO

```
PortMappings []PortMapping

CreatedAt time.Time

}

// NewManager creates a new network manager instance

func NewManager(config netutil.NetworkConfig) (*Manager, error) {

    // TODO 1: Validate network configuration parameters

    // TODO 2: Initialize IP address manager with subnet pool

    // TODO 3: Create iptables manager for NAT rule management

    // TODO 4: Initialize netlink helper for interface operations

    // TODO 5: Create container bridge if it doesn't exist

    // TODO 6: Set up base iptables chains for container networking

    // TODO 7: Enable IP forwarding on the host system

    // Hint: Use netutil.ValidateNetworkConfig for validation

}

// SetupContainerNetwork creates complete network setup for container

func (m *Manager) SetupContainerNetwork(containerID string, portMappings []PortMapping, pid int) (*ContainerNetwork, error) {

    // TODO 1: Generate unique veth interface names for this container

    // TODO 2: Allocate IP address from the managed pool

    // TODO 3: Create veth pair in host network namespace

    // TODO 4: Move container-side veth to container network namespace

    // TODO 5: Configure container interface with allocated IP address

    // TODO 6: Attach host-side veth to container bridge

    // TODO 7: Create port forwarding rules for specified port mappings

    // TODO 8: Store container network state for cleanup operations

    // TODO 9: Verify connectivity by testing basic network operations
```

```
// Hint: Use m.createVethPair, m.allocateIPAddress, m.setupPortForwarding
}

// createVethPair creates and configures veth pair for container

func (m *Manager) createVethPair(containerID, hostVeth, containerVeth string, pid int)
error {

    // TODO 1: Create veth pair using netlink helper with generated names

    // TODO 2: Move container-side interface to container network namespace

    // TODO 3: Bring up host-side interface in host namespace

    // TODO 4: Attach host-side interface to container bridge

    // TODO 5: Verify both interfaces are properly configured and active

    // Hint: Use netlink helper methods for interface operations

}

// configureContainerInterface sets up networking inside container namespace

func (m *Manager) configureContainerInterface(containerVeth, ipAddress string, pid int)
error {

    // TODO 1: Enter container network namespace using setns syscall

    // TODO 2: Configure container interface with assigned IP address

    // TODO 3: Set up default route pointing to bridge gateway

    // TODO 4: Configure loopback interface for internal communication

    // TODO 5: Return to host network namespace after configuration

    // Hint: Use netlink operations within target namespace context

}

// setupPortForwarding creates NAT rules for container port access

func (m *Manager) setupPortForwarding(containerID, containerIP string, portMappings
[]PortMapping) error {

    // TODO 1: Create container-specific iptables chain for isolation

    // TODO 2: Add jump rule to direct traffic to container chain
```

```

// TODO 3: Create DNAT rules for each port mapping specification

// TODO 4: Create corresponding FORWARD rules for traffic authorization

// TODO 5: Add MASQUERADE rule for return traffic if needed

// TODO 6: Verify rules are active and processing traffic correctly

// Hint: Use iptables manager methods with proper rule ordering

}

// CleanupContainerNetwork removes all network resources for container

func (m *Manager) CleanupContainerNetwork(containerID string) error {

    // TODO 1: Load container network state from storage

    // TODO 2: Remove port forwarding iptables rules

    // TODO 3: Delete custom iptables chains if empty

    // TODO 4: Remove veth interfaces from bridge and host

    // TODO 5: Release IP address back to available pool

    // TODO 6: Clean up container network state records

    // TODO 7: Verify all resources are properly cleaned up

    // Hint: Implement cleanup in reverse order of setup operations

}

// IPAddressManager handles IP address allocation and tracking

type IPAddressManager struct {

    subnet      *net.IPNet

    gateway     net.IP

    allocated   map[string]string // containerID -> IP

    reserved   map[string]time.Time // IP -> expiry

    mu         sync.RWMutex

}

```

```

// AllocateIP assigns IP address to container with conflict detection

func (ipam *IPAddressManager) AllocateIP(containerID string) (string, error) {

    // TODO 1: Check if container already has IP assignment

    // TODO 2: Compute candidate IP using hash of container ID

    // TODO 3: Verify candidate IP is within subnet and not reserved

    // TODO 4: If candidate unavailable, search sequentially for free IP

    // TODO 5: Reserve selected IP address for this container

    // TODO 6: Update allocation records with persistent storage

    // TODO 7: Return assigned IP address in CIDR notation

    // Hint: Use hash function for deterministic assignment strategy

}

// ReleaseIP returns IP address to available pool

func (ipam *IPAddressManager) ReleaseIP(containerID string) error {

    // TODO 1: Look up current IP assignment for container

    // TODO 2: Remove active allocation record

    // TODO 3: Add IP to reserved pool with expiration time

    // TODO 4: Update persistent storage with allocation changes

    // TODO 5: Verify IP is properly marked as available

    // Hint: Implement grace period before returning IP to pool

}

```

Language-Specific Hints:

- Use `vishvananda/netlink` library for programmatic network interface management instead of executing shell commands
- The `coreos/go-iptables` library provides safe iptables rule management with existence checking and atomic operations
- Network namespace operations require `syscall.Setns()` to enter target namespace before configuring interfaces
- Use `sync.RWMutex` for concurrent access to IP allocation maps and container network state

- Implement proper cleanup with `defer` statements to ensure resources are released even if errors occur
- Use `context.Context` for timeout handling during network operations that might hang
- The `net` package provides utilities for IP address parsing, subnet calculations, and CIDR operations

Milestone Checkpoint:

After implementing the network management component, verify functionality with these tests:

```
# Test 1: Basic container networking                                                 BASH
go run cmd/container-runtime/main.go create --id test-net --image alpine --command "sleep 30" --port 8080:80

# Expected: Container starts with assigned IP address and veth pair created

# Test 2: Inter-container communication

go run cmd/container-runtime/main.go create --id container1 --image alpine --command "nc -l -p 8080"

go run cmd/container-runtime/main.go create --id container2 --image alpine --command "nc container1-ip 8080"

# Expected: Containers can communicate directly via bridge networking

# Test 3: Port forwarding functionality

go run cmd/container-runtime/main.go create --id web-server --image nginx --port 8080:80

curl http://localhost:8080

# Expected: External access to container service through port forwarding

# Test 4: Network cleanup verification

go run cmd/container-runtime/main.go remove web-server

iptables -t nat -L | grep "web-server"

ip link show | grep "veth-web-server"

# Expected: No iptables rules or veth interfaces remain after cleanup
```

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Container has no network connectivity	Veth pair not created or misconfigured	<code>ip link show</code> in both namespaces	Verify veth creation and namespace assignment
External clients can't reach container	Port forwarding rules missing or incorrect	<code>iptables -t nat -L -n</code> to check DNAT rules	Recreate iptables rules with correct addresses
Containers can't communicate with each other	Bridge not forwarding or IP assignment conflicts	<code>brctl show</code> and ping tests between containers	Check bridge configuration and IP allocations
"No available IP addresses" error	IP pool exhausted or addresses not released	Check IPAM allocation records and container states	Implement address cleanup or expand subnet
Network setup hangs during container creation	Netlink operations blocked or namespace issues	<code>strace</code> the runtime process for system call failures	Check for kernel version compatibility and permissions

The network management component represents the final major subsystem in the container runtime, providing the networking foundation that transforms isolated processes into communicating services. Proper implementation of veth pairs, bridge networking, and port forwarding creates a complete container networking solution that balances isolation with connectivity requirements.

Interactions and Data Flow

Milestone(s): All milestones (1-4) - This section describes how components coordinate during the complete container lifecycle, showing the orchestration of namespace creation (Milestone 1), resource control setup (Milestone 2), filesystem mounting (Milestone 3), and network configuration (Milestone 4).

Mental Model: Orchestra Conductor

Think of the container runtime as a symphony orchestra performing a complex musical piece. The **Runtime Manager** serves as the conductor, coordinating multiple specialized sections (components) to create a harmonious performance. Each section has specific expertise - the **Namespace Handler** is like the string section providing the foundational melody (isolation), the **Resource Control Component** is like the percussion section maintaining rhythm and timing (resource limits), the **Filesystem Manager** is like the brass section adding layers of richness (overlay filesystem), and the **Network Manager** is like the woodwinds connecting different musical phrases (container communication).

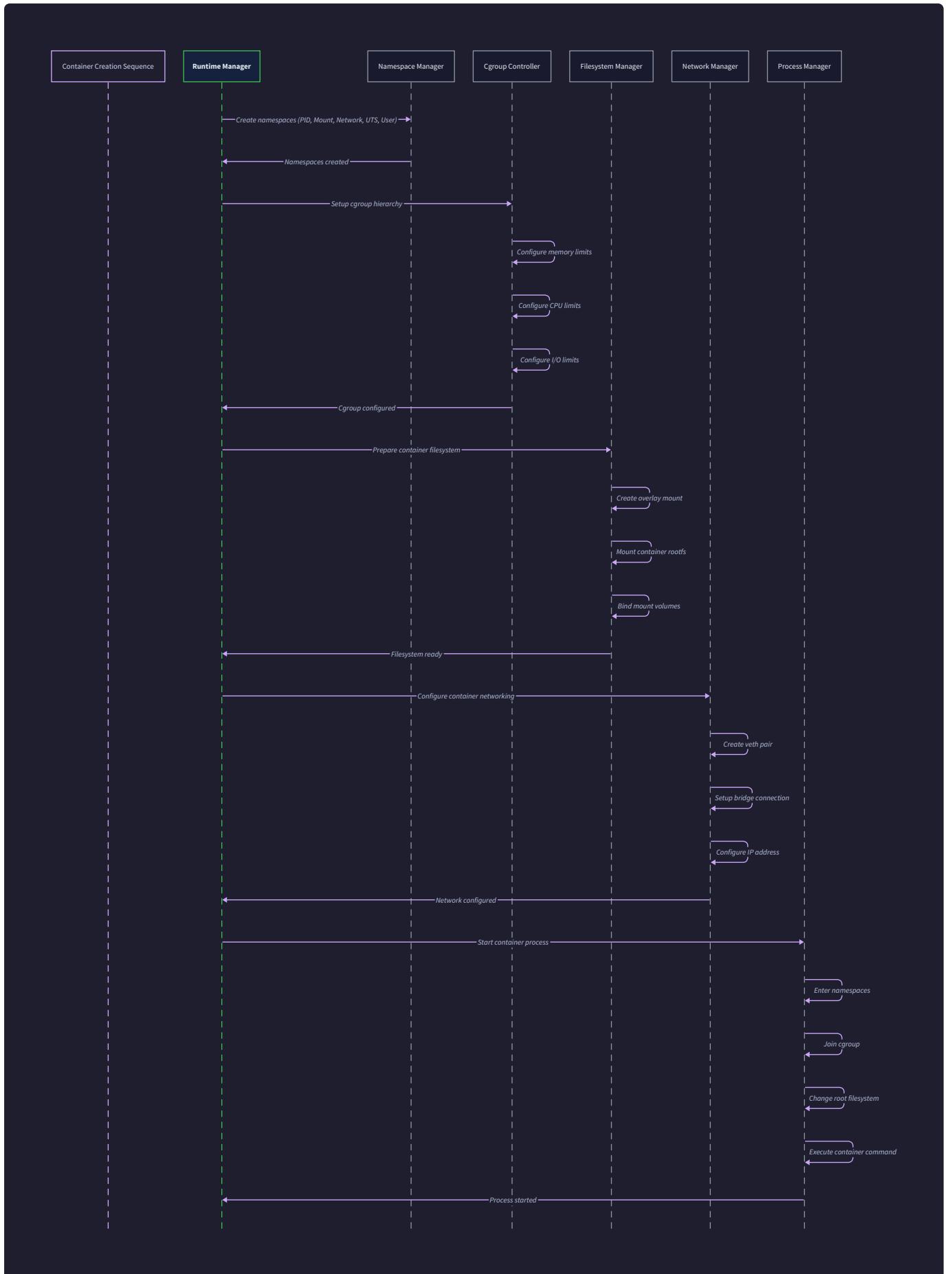
Just as a conductor must start each section at precisely the right moment and ensure they work together without interfering with each other, the Runtime Manager must orchestrate container creation by coordinating

namespace setup, cgroup creation, filesystem mounting, and network configuration in the correct sequence. If the conductor stops the music abruptly, each section must wind down gracefully - similarly, when stopping a container, each component must clean up its resources in the proper order to avoid leaving the system in an inconsistent state.

The magic happens in the coordination - each component performs its specialized function, but the real value emerges from how they work together to create an isolated, resource-controlled, networked container environment that appears seamless to the user.

Container Creation Sequence

The container creation sequence represents the most complex orchestration in our runtime, involving careful coordination between all components to establish a fully functional isolated environment. This process must handle partial failures gracefully, ensuring that if any step fails, previously created resources are properly cleaned up.



Pre-Creation Validation Phase

Before beginning the actual container creation, the Runtime Manager performs comprehensive validation to catch configuration errors early, before any system resources are allocated. This validation phase prevents scenarios where container creation fails halfway through due to invalid specifications, leaving partially created resources that require cleanup.

The validation process examines the `ContainerSpec` thoroughly, checking that the container ID follows the required format (alphanumeric characters, hyphens, and underscores only, 1-64 characters in length), the specified image layers exist in the local storage, resource limits are within acceptable ranges, and network configuration parameters are valid. Port mapping validation ensures that requested host ports are available and that container ports are within valid ranges (1-65535).

Validation Check	Purpose	Failure Consequence
Container ID format	Ensure filesystem-safe naming	Prevents mount point creation issues
Image layer existence	Verify overlay dependencies	Prevents overlayfs mount failures
Resource limit bounds	Validate cgroup parameters	Prevents cgroup controller errors
Port availability	Check host port conflicts	Prevents iptables rule conflicts
Network subnet capacity	Ensure IP addresses available	Prevents IP allocation failures

Design Insight: Front-loading validation prevents resource leakage. It's much cleaner to reject an invalid container specification before creating any namespaces or cgroups than to handle cleanup after partial creation failure.

Component Initialization Sequence

Once validation passes, the Runtime Manager begins the coordinated initialization sequence. This sequence is carefully ordered to handle dependencies between components - for example, namespaces must exist before processes can be assigned to cgroups, and the network namespace must be created before veth pairs can be moved into it.

Step 1: Container State Initialization

The Runtime Manager creates a new `ContainerState` record with status `ContainerCreated` and persists it to the state store. This persistent record serves as the authoritative source of truth about the container's existence and current state, enabling recovery operations if the runtime process crashes during creation.

Step 2: Namespace Creation

The Namespace Handler receives the container's `NamespaceConfig` and creates the requested Linux namespaces. The namespace creation follows a specific order: user namespace first (if enabled) to establish

privilege boundaries, followed by PID, mount, network, UTS, and IPC namespaces. Each namespace creation is verified before proceeding to the next.

The namespace creation process involves creating a new process with the appropriate `CLONE_NEW*` flags, configuring the namespace environment (such as UID/GID mappings for user namespaces), and establishing the communication channel for coordinating with the container process. The `NamespaceHandle` returned contains the process ID of the namespace holder and references to all created namespaces.

Step 3: Filesystem Preparation

With namespaces established, the Filesystem Manager prepares the container's root filesystem using overlayfs. This involves creating the directory structure for the overlay mount (upper directory for container changes, work directory for overlayfs operations), combining the image layers into the lower directories specification, and mounting the overlayfs with the proper options.

The overlay mount operation requires careful attention to mount options, ensuring that the work directory is on the same filesystem as the upper directory, that the lower directories are specified in the correct order (base layer first), and that the mount point has the appropriate permissions for the container's user namespace mappings.

Step 4: Cgroup Setup

The Resource Control Component creates the container's cgroup hierarchy and applies the specified resource limits. This involves creating a new cgroup under the runtime's cgroup subtree, enabling the required controllers (memory, CPU, I/O), configuring the resource limits according to the `ResourceLimits` specification, and preparing the cgroup for process assignment.

The cgroup creation must handle cgroups v2 delegation properly, ensuring that the runtime has sufficient permissions to create and manage the container cgroup, that the required controllers are available in the parent cgroup, and that resource limits are set before any processes are assigned to the cgroup.

Step 5: Network Configuration

The Network Manager sets up the container's networking environment by creating a veth pair, moving one end into the container's network namespace, configuring IP addresses and routing, and establishing port forwarding rules if required. This step requires coordination with the previously created network namespace.

The network setup process involves creating uniquely named veth interfaces, moving the container end of the veth pair into the container's network namespace, configuring the container interface with an allocated IP address, attaching the host end of the veth pair to the container bridge, and creating iptables rules for any requested port mappings.

Step 6: Essential Mount Setup

Within the container's mount namespace, essential filesystem mounts are established to provide the container with a functional environment. This includes mounting `/proc` with the appropriate options for the PID

namespace, mounting `/sys` with restricted access, creating `/dev` with essential device nodes, and establishing `/tmp` as a tmpfs mount.

These mounts are critical for container functionality - `/proc` provides process information and system interfaces, `/sys` exposes kernel information, and `/dev` provides access to essential devices like `/dev/null`, `/dev/zero`, and `/dev/random`.

State Transition and Process Launch

After all components have successfully prepared their resources, the Runtime Manager transitions the container state from `ContainerCreated` to `ContainerRunning` and launches the container's main process. This final step involves executing the container command within the prepared namespace environment, assigning the process to the container's cgroup, and establishing monitoring for the container process.

The process launch uses the Linux `execve` system call to replace the namespace holder process with the container's specified command, ensuring that the process inherits all the prepared namespace, filesystem, and network configuration. The process ID is recorded in the `ContainerState` for future management operations.

Failure Recovery During Creation

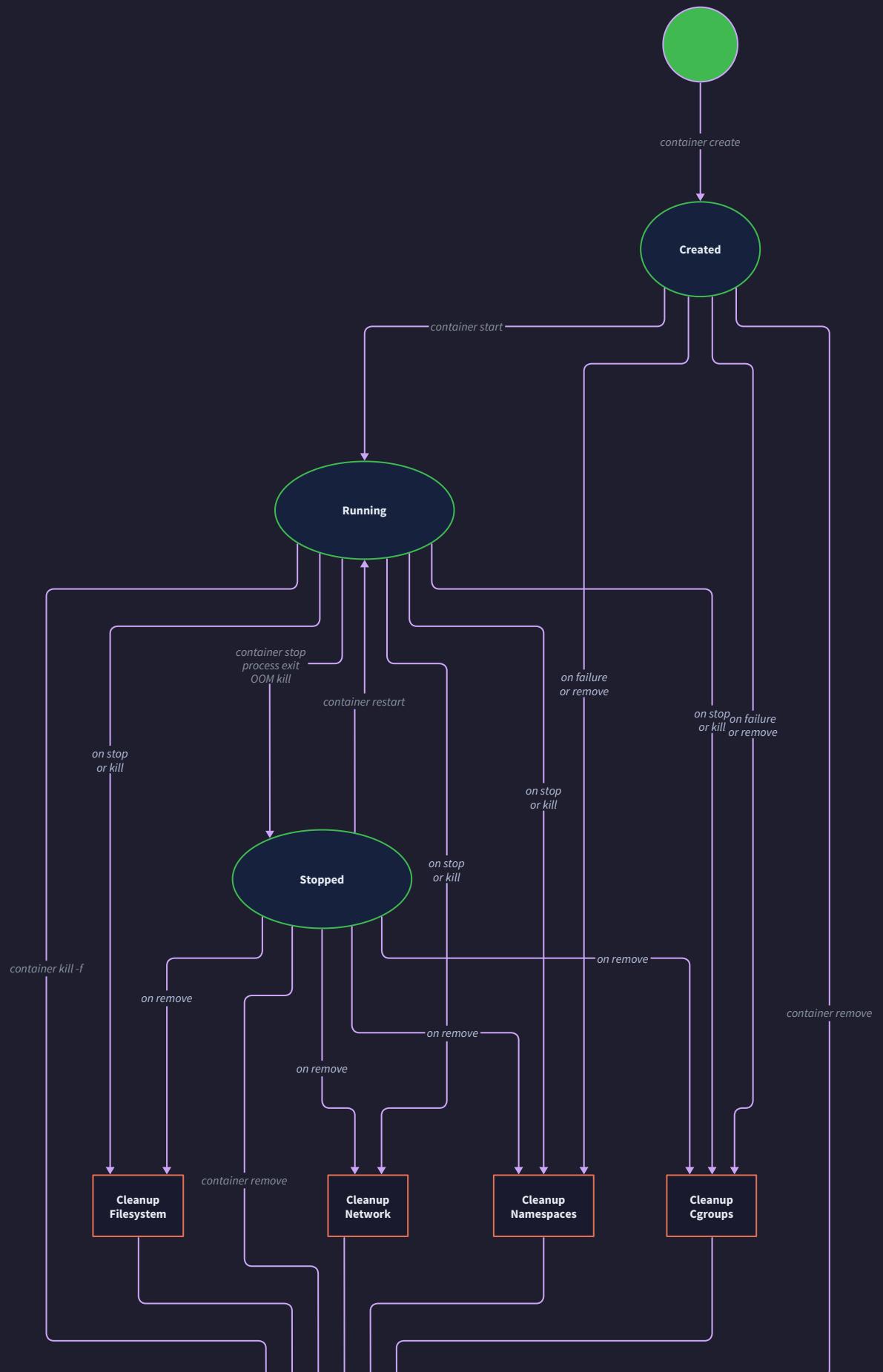
The container creation sequence must handle failures at any step, implementing a rollback mechanism that cleans up successfully created resources. The cleanup sequence runs in reverse order of creation, ensuring that dependencies are properly unwound.

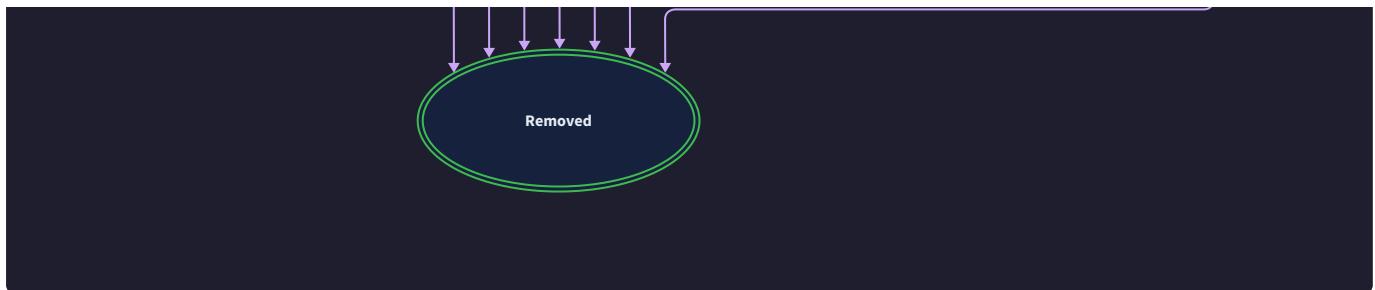
Failure Point	Resources to Clean Up	Cleanup Actions
Namespace creation	Container state	Delete state record
Filesystem preparation	Namespaces, state	Cleanup namespaces, delete state
Cgroup setup	Filesystem, namespaces, state	Unmount overlay, cleanup namespaces, delete state
Network configuration	Cgroups, filesystem, namespaces, state	Remove cgroup, unmount overlay, cleanup namespaces, delete state
Process launch	Network, cgroups, filesystem, namespaces, state	Full cleanup of all resources

Critical Design Decision: The creation sequence is designed to be atomic from the user's perspective - either a container is fully created and functional, or it doesn't exist at all. Partial containers are never left in the system.

Container Cleanup Sequence

The container cleanup sequence is equally critical to the creation sequence, ensuring that when containers are stopped and removed, all associated resources are properly released and the system is left in a clean state. This process must handle both graceful shutdowns and forced termination scenarios.





Graceful Termination Process

When a container stop request is received, the Runtime Manager begins a graceful termination process that allows the container application to shut down cleanly while systematically releasing system resources. This process respects the container's shutdown timeout while ensuring that system resources are not leaked.

Step 1: Process Termination Signal

The Runtime Manager sends a `SIGTERM` signal to the container's main process, allowing it to perform graceful shutdown operations such as closing database connections, saving state, and releasing application resources. The runtime waits for a configurable timeout period (typically 10 seconds) for the process to terminate voluntarily.

If the process doesn't terminate within the grace period, the Runtime Manager escalates to `SIGKILL`, which forcibly terminates the process. This two-phase termination approach balances application needs (graceful shutdown) with system reliability (guaranteed termination).

Step 2: Cgroup Process Migration and Cleanup

Once the container process has terminated, the Resource Control Component verifies that no other processes remain in the container's cgroup hierarchy. Any remaining processes are either migrated to a cleanup cgroup or terminated, depending on the runtime configuration.

The cgroup cleanup process then removes the container's cgroup hierarchy by first disabling all controllers, moving any remaining processes out of the cgroup, and finally removing the cgroup directory. This sequence is critical because cgroups with active processes or enabled controllers cannot be removed.

Step 3: Network Resource Cleanup

The Network Manager systematically removes all networking resources associated with the container. This involves removing iptables rules for port forwarding, releasing the allocated IP address back to the available pool, removing the veth pair (which automatically cleans up both host and container ends), and updating any bridge forwarding tables.

Network cleanup must be thorough because leaked networking resources can cause conflicts with future containers, prevent IP address reuse, and leave iptables rules that interfere with other container networking.

Step 4: Filesystem Unmounting and Cleanup

The Filesystem Manager unmounts the container's overlayfs and removes the associated directories. This process involves unmounting the overlay from the container's root mount point, removing the upper directory

(containing container changes), removing the work directory, and cleaning up any temporary mount points.

Special care is taken during unmount operations to handle busy filesystems - if processes are still accessing the container filesystem, the unmount is retried with increasingly aggressive approaches, culminating in lazy unmount (`MNT_DETACH`) if necessary.

Step 5: Namespace Cleanup

The Namespace Handler performs the final cleanup step by removing the container's namespaces. Since namespaces are automatically cleaned up by the kernel when the last process exits, this step primarily involves verifying that cleanup has occurred and releasing any runtime-held references to namespace file descriptors.

User namespace cleanup requires special attention to UID/GID mapping cleanup, ensuring that any temporary mapping files are removed and that the namespace's privilege boundaries are properly dissolved.

State Management During Cleanup

Throughout the cleanup sequence, the Runtime Manager updates the container's state record to reflect the cleanup progress. The container transitions from `ContainerRunning` to `ContainerStopped` once the process has terminated, and finally to `ContainerRemoved` once all resources have been cleaned up.

The state record serves as a checkpoint for recovery operations - if the runtime crashes during cleanup, it can resume the cleanup process from the last recorded state, preventing resource leaks and ensuring system consistency.

Cleanup Phase	Container State	Critical Actions	Failure Recovery
Process termination	<code>ContainerStopped</code>	Send SIGTERM/SIGKILL	Force kill on restart
Cgroup cleanup	<code>ContainerStopped</code>	Remove cgroup hierarchy	Retry cgroup removal
Network cleanup	<code>ContainerStopped</code>	Remove veth, iptables rules	Clean up by container ID scan
Filesystem cleanup	<code>ContainerStopped</code>	Unmount overlays	Force unmount on restart
Final cleanup	<code>ContainerRemoved</code>	Delete state record	Mark as removed

Forced Removal and Recovery

In some scenarios, containers may need to be forcibly removed due to unresponsive processes, corrupted state, or system recovery after crashes. The forced removal process bypasses graceful shutdown and aggressively cleans up resources.

Forced removal begins with `SIGKILL` to terminate all processes in the container's cgroup, proceeds with immediate resource cleanup without waiting for graceful termination, uses forced unmount operations for stuck filesystems, and removes networking resources even if some operations fail.

The recovery process after runtime crashes involves scanning the system for orphaned resources by examining cgroup hierarchies for container-specific paths, checking for overlay mounts with container-specific names, identifying veth interfaces and iptables rules with container tags, and cleaning up any discovered orphaned resources.

Operational Insight: Container cleanup is where many runtime bugs manifest. Thorough cleanup prevents resource exhaustion and ensures system stability under high container churn rates.

Cleanup Verification and Monitoring

After each cleanup operation, the Runtime Manager performs verification steps to ensure resources have been properly released. This includes verifying that the container's cgroup no longer exists in the filesystem, confirming that no overlay mounts remain for the container, checking that the container's IP address has been released, and ensuring that no iptables rules reference the container.

The verification process also updates system monitoring metrics, tracking successful cleanup operations, failed cleanup attempts requiring manual intervention, resource cleanup timing for performance optimization, and overall system resource utilization trends.

Component Communication Patterns

The interactions between components follow well-defined communication patterns that ensure consistency and enable proper error handling. These patterns establish clear boundaries and responsibilities while enabling the complex coordination required for container lifecycle management.

Synchronous Command-Response Pattern

The primary communication pattern used during container creation and removal is synchronous command-response, where the Runtime Manager issues commands to individual components and waits for completion before proceeding to the next step. This pattern ensures that each step completes successfully before dependent operations begin.

Each component operation returns both a success/failure indication and detailed information about the operation results. Success responses include resource handles, configuration details, and any information needed by subsequent steps. Failure responses include specific error codes, diagnostic information, and suggestions for recovery actions.

Component	Command Type	Response Data	Error Information
Namespace Handler	<code>CreateNamespaces</code>	<code>NamespaceHandle</code> with PIDs and paths	Namespace creation failures with system error codes
Resource Controller	<code>CreateContainerCgroup</code>	Cgroup path and applied limits	Controller availability and permission errors
Filesystem Manager	<code>PrepareContainerFilesystem</code>	Mount point and overlay paths	Mount failures and layer availability issues
Network Manager	<code>SetupContainerNetwork</code>	<code>ContainerNetwork</code> with interface details	IP allocation and bridge configuration errors

Event Notification Pattern

For monitoring and status updates, components use an event notification pattern to inform the Runtime Manager about state changes, resource usage alerts, and error conditions that occur outside of explicit command operations.

The event notification system operates asynchronously, allowing components to report important events without blocking their normal operations. Events include container process exit notifications, resource usage threshold breaches, network connectivity changes, and filesystem errors.

Resource Handle Management

Components return resource handles that encapsulate the resources they've created and provide controlled access for cleanup operations. These handles abstract the complexity of resource management while ensuring that resources can be properly released even if the component that created them is not available.

Resource handles include sufficient information for cleanup operations, maintain references to prevent premature resource release, provide verification methods for health checking, and enable diagnostic operations for troubleshooting.

Error Propagation and Recovery

The container runtime implements comprehensive error propagation and recovery mechanisms that ensure system stability even when individual operations fail. These mechanisms distinguish between recoverable errors (that can be retried) and permanent failures (that require different handling strategies).

Error Classification and Handling

Errors are classified into categories that determine the appropriate response strategy:

Transient Errors are temporary conditions that may resolve with retry, such as resource temporarily unavailable, network connectivity issues, and filesystem busy conditions. These errors trigger automatic retry.

with exponential backoff.

Configuration Errors indicate problems with the container specification or runtime configuration, such as invalid resource limits, missing image layers, and network configuration conflicts. These errors are reported immediately to the user without retry.

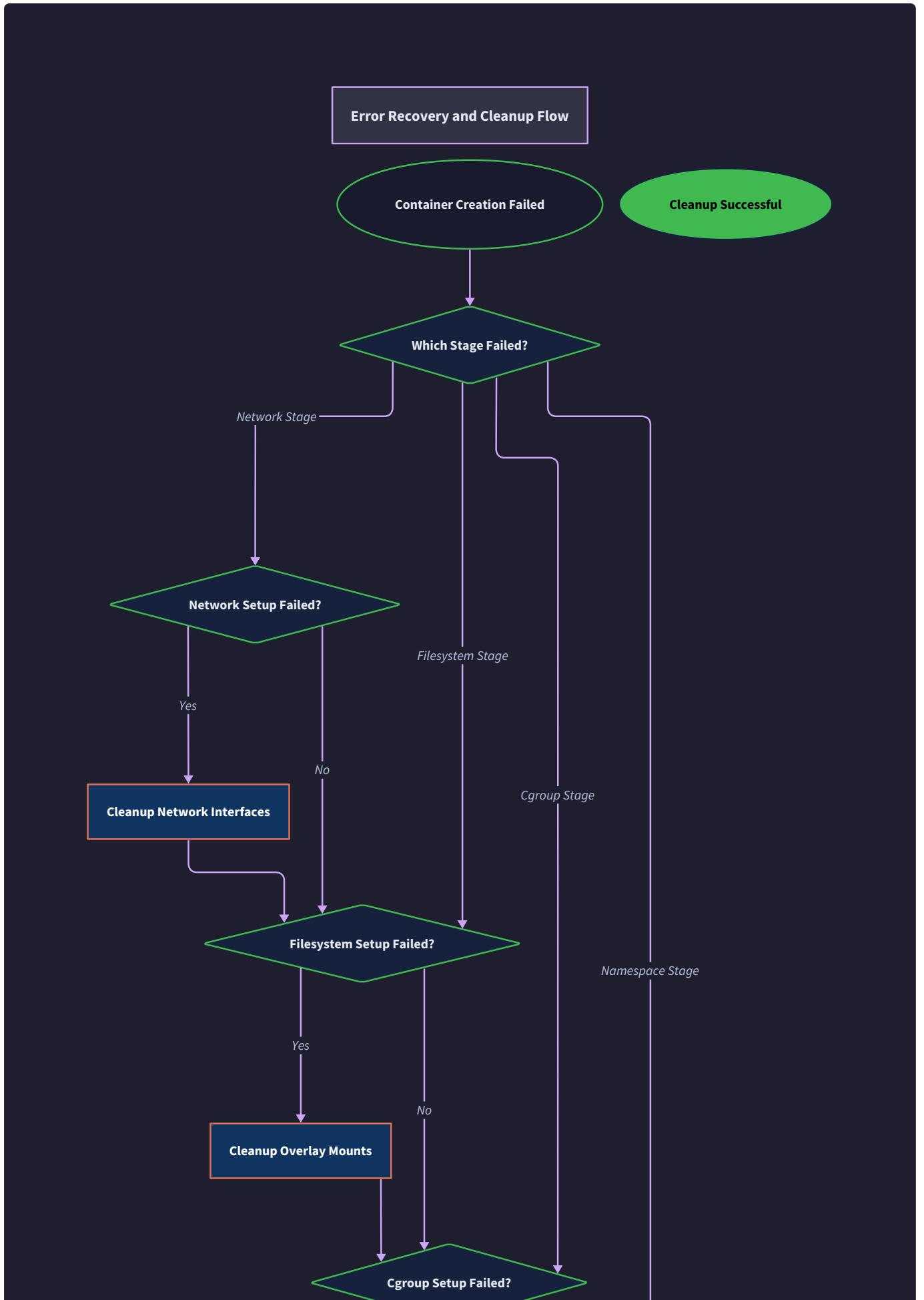
System Errors indicate serious problems with the underlying system, such as insufficient disk space, kernel feature unavailability, and permission denied errors. These errors may require administrative intervention.

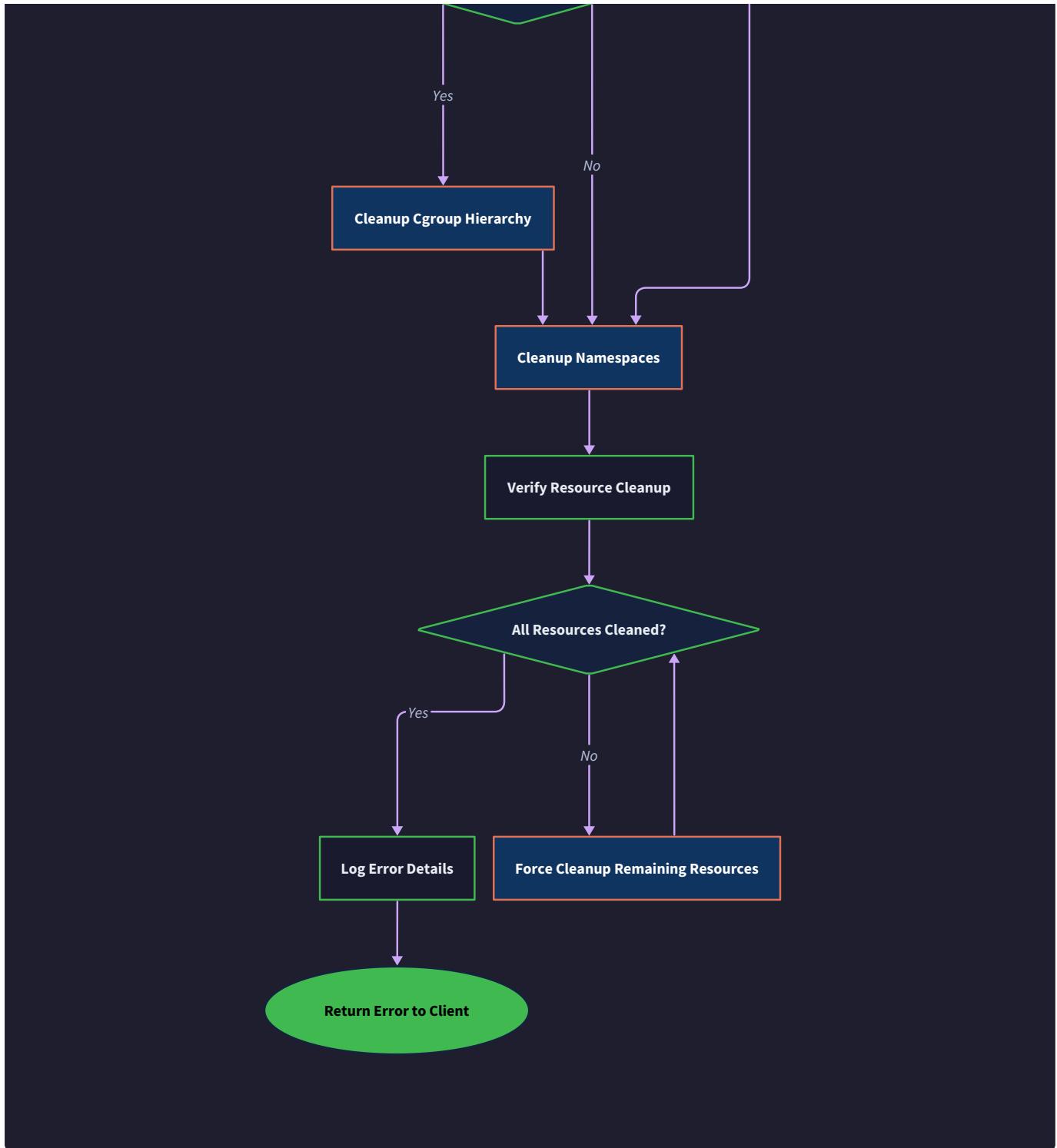
Resource Exhaustion occurs when system limits are reached, such as no available IP addresses, maximum containers reached, and insufficient memory for operations. These errors trigger cleanup of unused resources before retry.

Design Philosophy: Fail fast for configuration errors, retry intelligently for transient errors, and gracefully degrade for resource exhaustion scenarios.

Recovery State Machine

The runtime maintains a recovery state machine that tracks the progress of failed operations and determines the appropriate recovery actions:





The recovery process uses checkpointing to track which operations have completed successfully, enabling precise rollback without affecting unrelated system state. Recovery actions are idempotent, ensuring that repeated execution produces the same result without side effects.

Performance Considerations and Optimization

The container lifecycle operations are optimized for both individual container performance and system-wide scalability. These optimizations balance resource utilization, operation latency, and system throughput under various load patterns.

Parallel Component Operations

Where dependencies permit, component operations are executed in parallel to reduce container creation latency. For example, cgroup creation and network setup can proceed simultaneously after namespace creation, and filesystem layer verification can occur in parallel with network bridge preparation.

The parallel execution framework uses worker pools for each component type, coordinates dependencies through a directed acyclic graph, implements backpressure to prevent resource exhaustion, and maintains operation ordering where required for correctness.

Resource Pooling and Caching

Frequently used resources are pooled and cached to reduce allocation overhead:

Network Resource Pool: Pre-allocated IP addresses and veth interface names reduce network setup latency and prevent naming conflicts.

Filesystem Layer Cache: Image layers are cached with reference counting, enabling quick overlay setup for containers using the same base images.

Cgroup Template Cache: Common cgroup configurations are templated and cached, reducing the overhead of setting individual resource limits.

Batch Operations

When multiple containers are created simultaneously, the runtime batches certain operations to improve efficiency:

- iptables rules are batched and applied together to reduce netfilter reconfiguration overhead
- Filesystem operations are batched to take advantage of kernel I/O scheduling
- Cgroup operations are grouped to minimize cgroup hierarchy traversal

Implementation Guidance

The container lifecycle orchestration requires careful coordination of asynchronous operations while maintaining strong consistency guarantees. The following implementation approach provides a foundation for building reliable container lifecycle management.

Technology Recommendations

Component	Simple Option	Advanced Option
State Management	JSON files + file locking	Embedded database (bbolt/badger)
Process Management	os/exec + signal handling	Dedicated process manager with reaping
Error Handling	Structured errors + retry logic	Circuit breaker pattern with backoff
Logging	Standard log package	Structured logging (logrus/zap)
Monitoring	Basic metrics collection	Prometheus metrics + health checks

Recommended File Structure

```
internal/runtime/
    manager.go           ← Runtime Manager implementation
    manager_test.go      ← Lifecycle integration tests
    lifecycle.go         ← Container lifecycle state machine
    cleanup.go           ← Resource cleanup coordination
    validation.go        ← Container specification validation
    errors.go            ← Error types and handling
    state/
        manager.go       ← Container state persistence
        recovery.go      ← Crash recovery logic
    coordinator/
        coordinator.go   ← Component operation coordination
        parallel.go      ← Parallel operation execution
        rollback.go      ← Failure rollback logic
```

Core State Management Infrastructure

```
// StateManager provides persistent storage for container state with crash recovery          GO
type StateManager struct {
    stateDir    string
    containers map[string]*ContainerState
    mu          sync.RWMutex
}

// Recovery tracking

operations map[string]*OperationLog
recovery   *RecoveryManager
}

// OperationLog tracks the progress of multi-step operations for recovery

type OperationLog struct {
    ContainerID      string           `json:"container_id"`
    OperationType    string           `json:"operation_type"` // "create", "start",
    "stop", "remove"
    StartTime        time.Time        `json:"start_time"`
    CompletedSteps  []string         `json:"completed_steps"`
    CurrentStep     string           `json:"current_step"`
    RollbackNeeded  bool            `json:"rollback_needed"`
    ErrorDetails    map[string]interface{} `json:"error_details,omitempty"`
}

// SaveContainerState persists container state with atomic write operations

func (sm *StateManager) SaveContainerState(state *ContainerState) error {
    // TODO 1: Validate container state structure and required fields
    // TODO 2: Create temporary file for atomic write operation
}
```

```
// TODO 3: Serialize state to JSON with proper formatting

// TODO 4: Write to temporary file and fsync for durability

// TODO 5: Atomically rename temporary file to final location

// TODO 6: Update in-memory cache with new state

// Hint: Use os.Rename() for atomic file replacement

}

// LoadContainerState retrieves persisted state with error handling

func (sm *StateManager) LoadContainerState(containerID string) (*ContainerState, error) {

    // TODO 1: Validate container ID format and length

    // TODO 2: Construct state file path from container ID

    // TODO 3: Read state file with appropriate error handling

    // TODO 4: Parse JSON and validate required fields

    // TODO 5: Update in-memory cache with loaded state

    // TODO 6: Check for recovery operations in progress

    // Hint: Handle file not found vs. corruption differently

}
```

Runtime Manager Core Logic

```
// RuntimeManager orchestrates all container lifecycle operations
```

```
type RuntimeManager struct {

    stateManager      *StateManager
    nsHandler        *namespace.Handler
    cgroupController *cgroup.Controller
    fsManager         *filesystem.Manager
    networkManager   *network.Manager

    config           *RuntimeConfig
    eventChan        chan *RuntimeEvent
    shutdownCtx      context.Context
    shutdownCancel   context.CancelFunc
}
```

```
// CreateContainer implements the complete container creation sequence
```

```
func (rm *RuntimeManager) CreateContainer(spec *ContainerSpec) error {
    // TODO 1: Validate container specification thoroughly
    // TODO 2: Check for container ID conflicts in state store
    // TODO 3: Initialize container state record with Created status
    // TODO 4: Log operation start for crash recovery
    // TODO 5: Create namespaces according to namespace configuration
    // TODO 6: Prepare filesystem with overlay mount and layer setup
    // TODO 7: Create and configure cgroup with resource limits
    // TODO 8: Setup container networking with IP allocation
    // TODO 9: Update container state to Running status
    // TODO 10: Launch container process in prepared environment
}
```

GO

```
// TODO 11: Clean up resources if any step fails (rollback)

// Hint: Use defer functions for cleanup on error paths

}

// StopContainer implements graceful container termination

func (rm *RuntimeManager) StopContainer(containerID string, timeout time.Duration) error {

    // TODO 1: Load container state and verify it's running

    // TODO 2: Send SIGTERM to container process for graceful shutdown

    // TODO 3: Wait for process termination within timeout period

    // TODO 4: Send SIGKILL if process doesn't terminate gracefully

    // TODO 5: Update container state to Stopped status

    // TODO 6: Begin cleanup sequence for all component resources

    // TODO 7: Verify all resources have been properly released

    // TODO 8: Update monitoring metrics for container lifecycle

    // Hint: Use context with timeout for process termination wait

}
```

Component Coordination Framework

```
// ComponentCoordinator manages parallel and sequential operations across components
```

```
type ComponentCoordinator struct {
```

```
    components map[string]Component
```

```
    // Dependency graph for operation ordering
```

```
    dependencies map[string][]string
```

```
    // Operation tracking for rollback
```

```
    completedOps map[string][]string
```

```
    mu sync.Mutex
```

```
}
```

```
// Component interface that all runtime components must implement
```

```
type Component interface {
```

```
    // Setup prepares component resources for a container
```

```
    Setup(containerID string, spec *ContainerSpec) (*ComponentResult, error)
```

```
    // Cleanup releases component resources for a container
```

```
    Cleanup(containerID string) error
```

```
    // Verify checks that component resources are properly configured
```

```
    Verify(containerID string) error
```

```
    // GetDependencies returns components that must complete before this one
```

```
    GetDependencies() []string
```

```
}
```

GO

```
// ExecuteParallelOperations coordinates component operations with dependency handling

func (cc *ComponentCoordinator) ExecuteParallelOperations(containerID string, spec
*ContainerSpec) error {

    // TODO 1: Build dependency graph from component requirements

    // TODO 2: Identify operations that can execute in parallel

    // TODO 3: Create worker goroutines for independent operations

    // TODO 4: Use channels to coordinate dependency completion

    // TODO 5: Collect results and handle any operation failures

    // TODO 6: Implement rollback for failed parallel operations

    // TODO 7: Update operation log with completion status

    // Hint: Use sync.WaitGroup for parallel operation coordination

}
```

Error Handling and Recovery

```
// RecoveryManager handles crash recovery and partial operation cleanup
```

```
type RecoveryManager struct {

    stateManager    *StateManager

    operationLogs  map[string]*OperationLog

    components     map[string]Component

    // Recovery policies

    maxRetryAttempts int

    retryBackoff     time.Duration

    cleanupTimeout   time.Duration

}
```

```
// RecoverFromCrash identifies and completes interrupted operations
```

```
func (rm *RecoveryManager) RecoverFromCrash() error {

    // TODO 1: Scan operation log directory for incomplete operations

    // TODO 2: Load operation logs and determine recovery actions needed

    // TODO 3: Identify containers requiring rollback vs. completion

    // TODO 4: Execute cleanup operations for failed container creations

    // TODO 5: Restart monitoring for containers in running state

    // TODO 6: Update container states to reflect actual system state

    // TODO 7: Clean up orphaned system resources (cgroups, mounts, etc.)

    // Hint: Use system inspection to verify actual resource state

}

// RollbackOperation cleans up resources from a failed container operation
```

```
func (rm *RecoveryManager) RollbackOperation(opLog *OperationLog) error {

    // TODO 1: Determine which components completed successfully
```

GO

```

    // TODO 2: Execute cleanup in reverse order of creation

    // TODO 3: Handle cleanup failures with appropriate retry logic

    // TODO 4: Update container state to reflect rollback completion

    // TODO 5: Remove operation log once rollback completes

    // TODO 6: Log rollback results for debugging and monitoring

    // Hint: Components should implement idempotent cleanup operations

}

```

Milestone Checkpoint

After implementing the container lifecycle coordination:

- Creation Verification:** Run `go test ./internal/runtime/...` to verify creation sequence logic
- Manual Testing:** Create a container with `runtime create test-container config.json` and verify all namespaces, cgroups, and networking are properly configured
- Cleanup Verification:** Stop and remove the container, then inspect system state to ensure no leaked resources
- Recovery Testing:** Simulate runtime crash during container creation and verify recovery on restart
- Expected Behavior:** Containers should transition smoothly through Created → Running → Stopped → Removed states with proper resource management at each stage

Debugging Tips

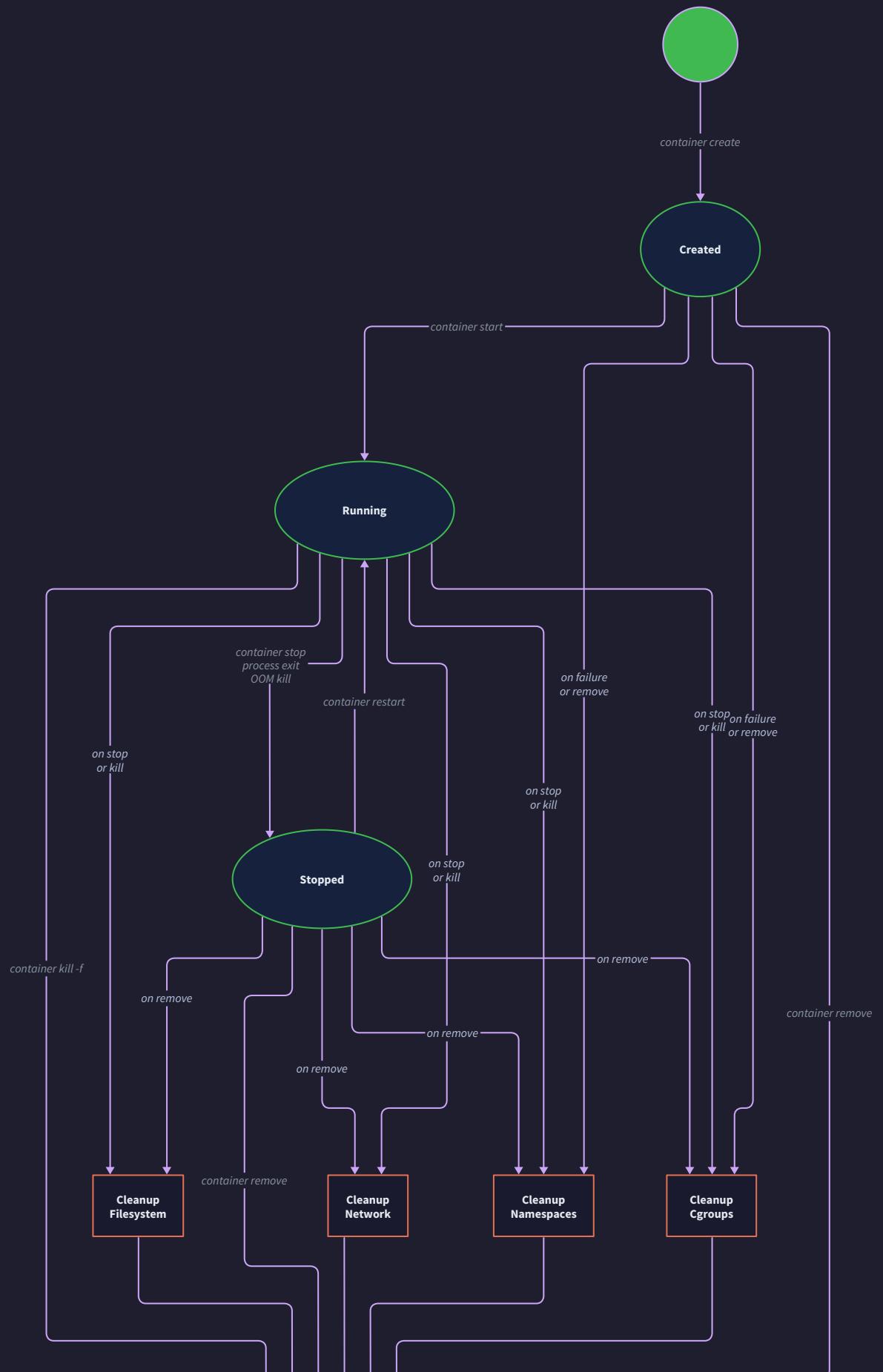
Symptom	Likely Cause	How to Diagnose	Fix
Container creation hangs	Component deadlock or dependency cycle	Check component dependency graph and operation logs	Implement operation timeouts and break dependency cycles
Partial container cleanup	Component cleanup failure	Examine container state and system resources	Implement idempotent cleanup and retry logic
Resource leaks after crashes	Missing recovery operations	Scan system for orphaned cgroups, mounts, network interfaces	Add comprehensive recovery scanning and cleanup
State corruption	Concurrent state modifications	Check for missing locks and atomic operations	Use file locking and atomic state updates

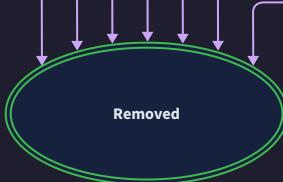
Error Handling and Edge Cases

Milestone(s): All milestones (1-4) - Error handling spans namespace creation (milestone 1), cgroup setup (milestone 2), filesystem mounting (milestone 3), and network configuration (milestone 4), requiring comprehensive failure recovery across all components.

Building a container runtime involves orchestrating multiple complex Linux kernel mechanisms, each with its own failure modes and edge cases. Think of error handling in a container runtime like managing a construction project with specialized contractors - when the electrician fails after the plumber has finished, you need a systematic plan to undo completed work while preserving what can be salvaged. Unlike simple applications where failures often mean "try again later," container runtime failures leave behind kernel resources (namespaces, cgroups, mount points, network interfaces) that must be meticulously cleaned up to avoid resource leaks and system instability.

The fundamental challenge in container runtime error handling is **partial failure recovery**. Container creation involves a precise sequence of operations across multiple kernel subsystems: creating namespaces, establishing cgroup hierarchies, mounting overlay filesystems, and configuring network interfaces. When operation 7 fails but operations 1-6 succeeded, the runtime must intelligently roll back the successful operations in the correct order while avoiding double-cleanup and resource conflicts.





Resource exhaustion scenarios present another class of critical edge cases. Unlike traditional applications that can gracefully degrade or queue requests, container runtimes must handle hard resource limits imposed by the kernel. When the system runs out of available PIDs, network interfaces, or memory, the runtime must detect these conditions early and provide meaningful feedback rather than cryptic kernel error messages.

Partial Failure Recovery

Partial failure recovery in container runtimes requires maintaining detailed operation logs and implementing idempotent cleanup procedures. Think of this like a database transaction - either all operations complete successfully, or the system returns to its original state with no side effects. However, unlike database transactions, kernel resource operations cannot be wrapped in a single atomic unit, requiring careful orchestration of cleanup procedures.

Decision: Operation Log with Rollback Mechanism

- **Context:** Container creation involves 10-15 distinct operations across multiple kernel subsystems, any of which can fail independently
- **Options Considered:**
 1. Best-effort cleanup without tracking
 2. Two-phase commit protocol for all operations
 3. Operation log with explicit rollback procedures
- **Decision:** Operation log with explicit rollback procedures
- **Rationale:** Two-phase commit is too heavy for kernel operations that cannot participate in distributed transactions. Best-effort cleanup leads to resource leaks. Operation logs provide precise tracking with efficient rollback.
- **Consequences:** Requires persistent storage for operation logs and careful ordering of cleanup procedures, but ensures no resource leaks and consistent system state.

The `OperationLog` structure tracks the progress of multi-step container operations and provides the foundation for intelligent rollback:

Field	Type	Description
ContainerID	string	Unique identifier linking log to container being created
OperationType	string	Type of operation (CREATE, START, STOP, REMOVE)
StartTime	time.Time	When the operation began for timeout detection
CompletedSteps	[]string	List of successfully completed operation steps
CurrentStep	string	Step currently being executed or that failed
RollbackNeeded	bool	Whether rollback procedures should be executed
ErrorDetails	map[string]interface{}	Structured error information for diagnosis

The rollback mechanism operates through **step-specific cleanup procedures** that can be safely executed multiple times (idempotent) and handle cases where the original operation partially succeeded. Each component provides cleanup methods that inspect the current system state and remove only resources that actually exist:

- 1. Check Operation Log:** The recovery manager scans persistent operation logs to identify incomplete operations from previous runtime executions
- 2. Determine Rollback Scope:** For each incomplete operation, examine the `CompletedSteps` list to identify which resources were successfully created
- 3. Execute Reverse Cleanup:** Call component-specific cleanup methods in reverse dependency order (network → filesystem → cgroup → namespaces)
- 4. Verify Resource Removal:** After each cleanup step, verify that kernel resources were actually removed and update the operation log
- 5. Handle Cleanup Failures:** If cleanup fails, mark specific resources as requiring manual intervention and continue with remaining cleanup
- 6. Log Completion:** Mark the operation log as fully rolled back or note any resources requiring administrator attention

Here's the detailed rollback sequence for each component:

Component	Cleanup Actions	Verification Steps	Failure Handling
Network Manager	Remove iptables rules, delete veth interfaces, release IP addresses	Check interface list, verify iptables rules removed	Log unreleased IPs for manual cleanup
Filesystem Manager	Unmount overlayfs, remove overlay directories, update layer references	Check mount table, verify directories removed	Force unmount with lazy flags if needed
Cgroup Controller	Remove processes from cgroups, delete cgroup directories	Check cgroup.procs file empty, verify directory removal	Kill remaining processes if needed
Namespace Handler	Close namespace file descriptors, clean up pivot_root artifacts	Check /proc/PID/ns/ entries, verify mount points	Mark namespaces for kernel cleanup on process exit

⚠ Pitfall: Cleanup Order Dependencies

A common mistake is cleaning up components in creation order rather than reverse dependency order. For example, attempting to remove a cgroup before stopping the network manager can fail because iptables rules might still reference the container's cgroup. The correct cleanup sequence respects dependency relationships: network configuration depends on namespaces existing, cgroups depend on processes being manageable, and filesystem mounts must be unmounted before namespace cleanup.

The `RecoveryManager` implements crash recovery by scanning operation logs on startup and completing interrupted operations:

Recovery Scenario	Detection Method	Recovery Action	Outcome
Runtime crashed during container creation	Operation log shows incomplete CREATE with recent timestamp	Execute rollback for all completed steps	Container fully removed, resources cleaned
Container process died unexpectedly	PID in container state no longer exists	Mark container as stopped, clean up kernel resources	Container marked STOPPED, resources cleaned
Manual container kill bypassed runtime	Container state shows RUNNING but process not found	Update state to STOPPED, leave resources for explicit cleanup	Consistent state, manual removal required
Kernel resource leaked from previous session	System resources exist but not in runtime state	Log leaked resources, optionally clean up obvious orphans	Clean system state, administrator notified

The key insight for partial failure recovery is that kernel resources have different cleanup requirements than application resources. A failed malloc() simply returns memory to the heap, but a failed namespace creation might leave mount points, cgroup entries, and network interfaces that must be explicitly cleaned up through separate system calls.

Resource Exhaustion Scenarios

Resource exhaustion scenarios in container runtimes require proactive detection and graceful degradation rather than reactive error handling. Think of resource exhaustion like a city's infrastructure during peak demand - water pressure drops before pipes burst, and smart systems detect the decline and take preventive action rather than waiting for catastrophic failure.

The container runtime must monitor multiple types of resource exhaustion:

Resource Type	Exhaustion Symptoms	Detection Method	Recovery Strategy
Process IDs	fork() returns EAGAIN, new containers fail to start	Check /proc/sys/kernel/pid_max and active PID count	Queue container requests, increase PID limit if possible
Memory	OOM killer activates, container creation fails	Monitor memory.current vs memory.max in cgroups	Reject new containers, suggest memory limit increases
Network Interfaces	veth creation fails, bridge attachment fails	Count interfaces in /sys/class/net/	Clean up unused interfaces, implement interface pooling
File Descriptors	open() returns EMFILE/ENFILE	Check /proc/PID/limits and /proc/sys/fs/file-nr	Close unused FDs, increase process limits
Disk Space	Filesystem operations fail with ENOSPC	Monitor filesystem usage for container storage	Clean up stopped containers, rotate logs
Cgroup Hierarchy Depth	cgroup creation fails with nested limits	Check cgroup depth in /sys/fs/cgroup hierarchy	Flatten cgroup structure, remove unused cgroups

PID Exhaustion Detection and Mitigation:

PID exhaustion is particularly problematic for container runtimes because each container typically creates multiple processes (init process, potential child processes, namespace management processes). The runtime implements PID exhaustion protection through monitoring and admission control:

- 1. Current PID Usage Monitoring:** Periodically scan /proc to count active PIDs and calculate utilization against the system limit (/proc/sys/kernel/pid_max)
- 2. Per-Container PID Tracking:** Maintain accurate counts of PIDs allocated to each container through cgroup process accounting
- 3. Admission Control:** Reject new container creation requests when PID utilization exceeds a configured threshold (e.g., 85%)
- 4. PID Cleanup Detection:** Identify containers with higher PID usage than expected and investigate potential PID leaks
- 5. Emergency PID Recovery:** When PID exhaustion is imminent, forcefully terminate containers marked as non-critical to free PIDs for essential operations

Memory Pressure Handling:

Memory exhaustion affects both the container runtime itself and the containers it manages. The runtime implements multi-level memory pressure detection:

Pressure Level	Detection Threshold	Runtime Action	Container Action
Normal	Memory usage < 70% of system	Normal operation	No restrictions
Warning	Memory usage 70-85% of system	Log warnings, reduce caching	Monitor container memory trends
Critical	Memory usage 85-95% of system	Reject new containers	Apply stricter memory limits to new containers
Emergency	Memory usage > 95% or swap thrashing	Stop non-essential containers	Trigger graceful shutdown of containers marked non-critical

The runtime tracks memory pressure through multiple indicators:

- 1. System Memory Utilization:** Monitor /proc/meminfo for available memory vs total memory
- 2. Container Memory Pressure:** Check memory.pressure files in container cgroups to detect containers approaching their limits
- 3. Swap Activity:** Monitor swap usage trends to detect memory thrashing before system becomes unresponsive
- 4. OOM Event Detection:** Scan kernel logs and cgroup memory.events for OOM kill events affecting containers
- 5. Memory Allocation Failure Rate:** Track the rate of memory allocation failures in container creation operations

⚠ Pitfall: Resource Exhaustion Cascades

A subtle failure mode occurs when resource exhaustion in one area triggers exhaustion in another. For example, when the runtime cannot create new network interfaces due to kernel limits, it might retry the operation multiple times, consuming PIDs for helper processes and eventually hitting PID limits as well. The runtime must implement circuit breaker patterns that prevent retry loops from amplifying resource exhaustion across multiple subsystems.

Network Resource Exhaustion:

Network interface exhaustion is less common but can occur in environments running many containers with complex networking requirements. The runtime implements network resource management through:

Network Resource	Limit Type	Detection Method	Mitigation Strategy
veth Interfaces	Kernel compile-time limit	Count interfaces in /sys/class/net/	Implement interface reuse pool
IP Addresses	Configured subnet size	Track allocated IPs in IPAM	Expand subnet or implement IP recycling
iptables Rules	Memory and rule count limits	Monitor iptables rule count	Consolidate rules, implement rule cleanup
Bridge Interfaces	Administrative limit	Count bridge interfaces	Reuse bridges across containers
netns File Descriptors	Process file descriptor limit	Track open namespace file descriptors	Close unused namespace references

The `ResourceMonitor` component implements comprehensive resource exhaustion detection:

Monitoring Function	Check Interval	Alert Threshold	Action Threshold
System PID usage	30 seconds	70% utilization	85% utilization
Memory pressure	15 seconds	Moderate pressure	High pressure
Filesystem space	60 seconds	80% full	90% full
Network interface count	120 seconds	80% of observed maximum	95% of observed maximum
Cgroup hierarchy depth	300 seconds	Depth > 10 levels	Depth > 15 levels

Resource Exhaustion Recovery Procedures:

When resource exhaustion is detected, the runtime follows escalating recovery procedures:

- 1. Level 1 - Preventive Measures:** Increase monitoring frequency, log detailed resource usage, reject non-critical operations

2. **Level 2 - Resource Reclamation:** Clean up stopped containers, release unused resources, compact resource allocations
3. **Level 3 - Load Shedding:** Reject new container creation requests, terminate containers marked as non-essential
4. **Level 4 - Emergency Cleanup:** Force cleanup of resources that appear leaked, restart runtime components to reset resource tracking
5. **Level 5 - System Protection:** Invoke emergency shutdown procedures to prevent system-wide resource exhaustion

The critical principle in resource exhaustion handling is failing fast and failing clearly. When the runtime detects that it cannot reliably create new containers due to resource constraints, it should immediately return a clear error message rather than attempting the operation and leaving behind partially created resources that worsen the exhaustion.

Implementation Guidance

The error handling and recovery system requires careful coordination of persistent state management and idempotent cleanup procedures. This implementation focuses on building robust error handling that prevents resource leaks and provides clear diagnostic information.

Technology Recommendations

Component	Simple Option	Advanced Option
Operation Logging	JSON files with file locking	SQLite database with transactions
Resource Monitoring	Periodic polling with goroutines	Event-driven monitoring with inotify
Process Coordination	Channel-based synchronization	Context-based cancellation with sync.WaitGroup
Error Classification	Custom error types with wrapping	Structured errors with error codes and metadata

Recommended File Structure

```
internal/
  runtime/
    manager.go          ← RuntimeManager with error handling
    recovery.go         ← RecoveryManager implementation
    operation_log.go    ← OperationLog persistence
  monitoring/
    resource_monitor.go ← ResourceMonitor implementation
    alerts.go           ← Resource exhaustion alerting
  errors/
    types.go            ← Error type definitions
    recovery.go         ← Error recovery utilities
  components/
    namespace/
      cleanup.go        ← Namespace cleanup procedures
    cgroup/
      cleanup.go        ← Cgroup cleanup procedures
    filesystem/
      cleanup.go        ← Filesystem cleanup procedures
    network/
      cleanup.go        ← Network cleanup procedures
```

Core Error Handling Infrastructure

```
// OperationLog tracks multi-step operations for failure recovery          GO

type OperationLog struct {

    ContainerID     string           `json:"container_id"`
    OperationType   string           `json:"operation_type"`
    StartTime       time.Time        `json:"start_time"`
    CompletedSteps []string          `json:"completed_steps"`
    CurrentStep     string           `json:"current_step"`
    RollbackNeeded bool             `json:"rollback_needed"`
    ErrorDetails    map[string]interface{} `json:"error_details"`

}

// RecoveryManager handles crash recovery and resource cleanup

type RecoveryManager struct {

    logDir      string
    components  map[string]Component
    mu          sync.RWMutex

}

// PersistOperationStep records successful completion of an operation step

func (rm *RecoveryManager) PersistOperationStep(containerID, step string) error {

    // TODO 1: Load existing operation log from disk using container ID

    // TODO 2: Add step to CompletedSteps list if not already present

    // TODO 3: Update CurrentStep to reflect progress

    // TODO 4: Write updated log back to disk with atomic file operations

    // TODO 5: Ensure file permissions allow cleanup by other processes

    // Hint: Use a temporary file + rename for atomic updates

}
```

```
// ExecuteRollback cleans up resources from a failed operation

func (rm *RecoveryManager) ExecuteRollback(opLog *OperationLog) error {

    // TODO 1: Reverse the order of completed steps for cleanup

    // TODO 2: For each step, identify the responsible component

    // TODO 3: Call component-specific cleanup method with error handling

    // TODO 4: Update operation log to track cleanup progress

    // TODO 5: Handle partial cleanup failures gracefully

    // TODO 6: Mark operation log as fully cleaned up

    // Hint: Continue cleanup even if individual steps fail

}
```

Resource Monitoring Implementation

```
// ResourceMonitor tracks system resource usage and exhaustion          GO

type ResourceMonitor struct {

    alertThresholds map[string]float64

    checkInterval   time.Duration

    alertHandlers   []func(ResourceAlert)

    mu              sync.RWMutex

}

// StartMonitoring begins periodic resource usage checking

func (rm *ResourceMonitor) StartMonitoring(ctx context.Context) {

    ticker := time.NewTicker(rm.checkInterval)

    defer ticker.Stop()

    for {

        select {

        case <-ctx.Done():

            return

        case <-ticker.C:

            // TODO 1: Check PID usage by scanning /proc filesystem

            // TODO 2: Check memory pressure from /proc/meminfo

            // TODO 3: Check filesystem usage for container storage

            // TODO 4: Count network interfaces in /sys/class/net/

            // TODO 5: Compare usage against configured thresholds

            // TODO 6: Generate alerts for resources approaching limits

            // Hint: Use /proc/sys/kernel/pid_max for PID limit

        }

    }

}
```

```
    }

}

// CheckPIDExhaustion determines if system is approaching PID limit

func (rm *ResourceMonitor) CheckPIDExhaustion() (float64, error) {

    // TODO 1: Read PID limit from /proc/sys/kernel/pid_max

    // TODO 2: Count active PIDs by scanning /proc directory

    // TODO 3: Calculate utilization percentage

    // TODO 4: Return usage ratio for threshold comparison

    // Hint: Handle race conditions where PIDs change during scan

}
```

Component Cleanup Interfaces

```
// Component defines cleanup interface for runtime components  
GO  
  
type Component interface {  
  
    // CleanupResources removes all resources associated with container  
  
    CleanupResources(containerID string) error  
  
  
    // VerifyCleanup checks that all resources were actually removed  
  
    VerifyCleanup(containerID string) error  
  
  
    // ListResources returns resources currently allocated to container  
  
    ListResources(containerID string) ([]string, error)  
  
}  
  
// Implement cleanup for each component following this pattern:  
  
// CleanupNamespaces removes namespace resources for container  
  
func (nh *NamespaceHandler) CleanupResources(containerID string) error {  
  
    // TODO 1: Load namespace handle from persistent state  
  
    // TODO 2: Close all namespace file descriptors  
  
    // TODO 3: Unmount any remaining filesystem mounts  
  
    // TODO 4: Remove namespace-specific directories  
  
    // TODO 5: Verify namespace cleanup completed successfully  
  
    // Hint: Use lazy unmount (MNT_DETACH) for stuck mounts  
  
}  
  
// CleanupCgroups removes cgroup hierarchy for container  
  
func (cc *CgroupController) CleanupResources(containerID string) error {  
  
    // TODO 1: Find cgroup path from container ID
```

```
// TODO 2: Kill any remaining processes in cgroup  
  
// TODO 3: Remove container cgroup directory  
  
// TODO 4: Update cgroup reference counts  
  
// TODO 5: Clean up empty parent cgroups if needed  
  
// Hint: Check cgroup.procs is empty before removing directory  
  
}
```

Milestone Checkpoints

After implementing operation logging:

- Run `go test ./internal/runtime/recovery_test.go` to verify log persistence
- Create a container and verify operation log is written to disk
- Kill the runtime process during container creation and verify rollback on restart

After implementing resource monitoring:

- Run `go test ./internal/monitoring/resource_test.go` to verify threshold detection
- Create containers until PID usage approaches threshold
- Verify monitoring detects resource pressure and generates appropriate alerts

After implementing component cleanup:

- Run integration test that creates container and injects failure at each step
- Verify all kernel resources are cleaned up after each failure scenario
- Check that no mount points, cgroup directories, or network interfaces leak

Debugging Common Error Handling Issues

Symptom	Likely Cause	How to Diagnose	Fix
"Resource busy" during cleanup	Another process still using resource	<code>lsof</code> for files, <code>lsns</code> for namespaces	Kill processes holding resource
Operation log corruption	Concurrent writes or crash during write	Check file timestamps and JSON validity	Implement file locking or atomic writes
Cleanup hangs indefinitely	Deadlock in kernel resource cleanup	<code>strace</code> the cleanup process	Use timeouts and force cleanup options
Resource leak after container removal	Component cleanup not called or failed	Check system resources before/after	Verify cleanup methods are idempotent
False resource exhaustion alerts	Monitoring logic counting incorrectly	Compare manual resource counts	Fix counting logic and add bounds checking

Testing Strategy

Milestone(s): All milestones (1-4) - This section provides comprehensive testing approaches for namespace isolation (milestone 1), resource control (milestone 2), filesystem layering (milestone 3), and container networking (milestone 4), along with end-to-end integration scenarios.

Mental Model: Multi-Layer Quality Gates

Think of testing a container runtime like quality control in a multi-story manufacturing plant. Each floor represents a milestone - the first floor handles basic isolation (like separate workspaces), the second floor adds resource management (like power and material limits), the third floor implements storage systems (like shared warehouses with private modifications), and the fourth floor connects communication networks (like phone systems between departments). Quality inspectors work at three levels: **unit inspectors** check each machine in isolation, **floor supervisors** verify that all machines on a floor work together properly, and **plant managers** run full production scenarios to ensure the entire facility operates as designed. Just as a defect on the first floor can cascade upward and ruin the final product, a namespace isolation bug can break resource limits, filesystem mounting, and networking. Our testing strategy mirrors this layered approach - we verify each component works correctly in isolation, then test milestone-by-milestone integration, and finally run realistic end-to-end scenarios that exercise the complete container lifecycle.

The key insight is that container runtime testing requires **progressive validation** - you cannot meaningfully test resource limits if namespace isolation is broken, and networking tests are pointless if the filesystem layer cannot mount properly. Each milestone builds upon the previous one, creating a testing dependency chain

that must be validated in sequence. This approach helps developers understand not just whether their implementation works, but where failures originate and how they propagate through the system.

Milestone Checkpoints

Each milestone represents a significant capability increment in our container runtime. The testing approach for each milestone focuses on verifying the specific isolation mechanisms and their integration with previously implemented components. These checkpoints provide concrete validation criteria and specific test scenarios that demonstrate proper functionality.

Milestone 1: Process Isolation with Namespaces

The first milestone establishes the foundation of container isolation using Linux namespaces. Testing focuses on verifying that each namespace type correctly isolates the container's view of system resources from the host and other containers.

PID Namespace Validation tests ensure that the container process sees itself as PID 1 and cannot see host processes. The test creates a container, executes a process inside it, and verifies that `/proc/1/` inside the container refers to the container's initial process rather than the host's init system. Additionally, the test should confirm that `ps aux` inside the container shows only container processes, while `ps aux` on the host shows both host and container processes with different PID mappings.

Mount Namespace Validation verifies filesystem isolation by creating different mount points inside the container that should not appear on the host filesystem. The test mounts a temporary filesystem at `/tmp/container-test` inside the container, writes a file to that location, then confirms that the mount point and file are not visible from the host namespace. This test also validates that changes to existing mount points inside the container (such as remounting `/proc` with different options) do not affect the host's view of those filesystems.

Network Namespace Validation confirms network stack isolation by checking that the container has its own network interfaces, routing tables, and firewall rules. The test should verify that `ip link list` inside the container shows different interfaces than on the host, that the container cannot see network connections established on the host using `netstat`, and that network configuration changes inside the container do not affect host networking.

UTS Namespace Validation tests hostname and domain name isolation by setting different values inside the container and verifying they do not affect the host system. The test sets the container hostname to `test-container`, confirms that `hostname` inside the container returns this value, while `hostname` on the host returns the original host name.

User Namespace Validation (when implemented) verifies UID and GID mapping by creating a container process that appears to run as root (UID 0) inside the container but actually runs as an unprivileged user on the host. The test should confirm that `id` inside the container shows UID 0, while the same process appears with a mapped UID when viewed from the host.

Test Scenario	Validation Method	Expected Behavior	Failure Indicators
PID isolation	Check <code>/proc/1/comm</code> inside container	Container process name, not host init	Shows host init process or "No such file"
Process visibility	Run <code>ps aux</code> inside container	Only container processes visible	Host processes appear in container
Mount isolation	Create mount inside container, check host	Mount not visible on host filesystem	Host shows container mounts
Network interface isolation	Compare <code>ip link</code> output	Different interfaces in container vs host	Same interfaces visible in both
Hostname isolation	Set hostname in container, check host	Host hostname unchanged	Container hostname affects host
File ownership mapping	Create file as root in user namespace	File owned by mapped UID on host	File shows UID 0 on host filesystem

The milestone 1 checkpoint should demonstrate that a simple container can be created and started, with the container process executing in isolated namespaces. The test creates a container that runs `sleep 60`, then uses various inspection commands to verify isolation properties while the container is running.

Milestone 2: Resource Limits with Cgroups

The second milestone adds resource control capabilities on top of namespace isolation. Testing focuses on verifying that cgroups v2 controllers correctly enforce CPU, memory, and I/O limits, and that resource monitoring provides accurate usage statistics.

Memory Limit Enforcement tests create containers with specific memory limits and verify that processes exceeding those limits are terminated by the OOM killer. The test starts a container with a 50MB memory limit, runs a process that attempts to allocate 100MB of memory, and confirms that the process is killed before it can consume host memory beyond the limit. Additionally, the test should verify that `GetMemoryUsage` returns accurate current consumption and that memory pressure indicators increase as the limit is approached.

CPU Limit Enforcement validates that CPU quota and period settings correctly throttle container processes. The test creates a container with a 50% CPU limit (`cpu.max` set to "50000 100000"), runs a CPU-intensive process, and measures actual CPU consumption over a period to confirm it does not exceed the configured limit. The test should also verify that multiple containers with CPU limits can coexist without interfering with each other's quotas.

Process Limit Enforcement tests the PIDs controller by creating a container with a maximum of 10 processes, then attempting to fork more processes and verifying that process creation fails once the limit is reached. This test ensures that containers cannot exhaust the host's PID space through excessive process creation.

Resource Usage Monitoring validates the accuracy of resource consumption reporting. The test creates containers with known workloads (specific memory allocation patterns, CPU-intensive loops, file I/O operations) and compares the reported usage statistics from `GetResourceUsage` against expected values. This test also verifies that usage statistics are updated within reasonable time intervals.

OOM Event Detection tests the container runtime's ability to detect and report out-of-memory kills. The test creates a container with a low memory limit, runs a process that gradually increases memory consumption until it triggers an OOM kill, then verifies that `CheckOOMEvents` correctly reports the kill event with appropriate metadata.

Test Scenario	Resource Limit	Test Workload	Expected Outcome	Validation Method
Memory enforcement	50MB	Allocate 100MB array	Process killed by OOM	Check exit code and OOM counter
CPU throttling	0.5 CPU cores	Infinite loop	~50% CPU usage	Measure CPU time over 10 seconds
PID limit	10 processes	Fork bomb	Fork fails after 10	Count processes in cgroup
Memory monitoring	100MB	Allocate 75MB	75MB usage reported	Compare <code>GetMemoryUsage()</code> result
OOM detection	25MB	Gradual allocation	OOM event detected	<code>CheckOOMEvents()</code> returns true

The milestone 2 checkpoint demonstrates that containers respect resource limits and that the runtime can accurately monitor and report resource consumption. The test creates multiple containers with different resource limits, runs various workloads simultaneously, and verifies that each container's resource usage stays within its configured bounds while resource monitoring provides accurate statistics.

Milestone 3: Overlay Filesystem

The third milestone implements layered filesystem support using overlayfs, enabling efficient copy-on-write semantics for container images. Testing focuses on verifying proper layer stacking, copy-on-write behavior, and cleanup operations.

Layer Stacking Validation tests that multiple read-only layers combine correctly with a writable upper layer to create the expected merged filesystem view. The test creates a container with three layers: a base layer containing `/bin/sh`, a second layer adding `/etc/hostname`, and a third layer adding `/usr/local/bin/custom-tool`. The container should see all files from all layers in their correct locations, with files from higher layers overriding those from lower layers when paths conflict.

Copy-on-Write Behavior validates that file modifications inside the container create copies in the upper layer without affecting the original read-only layers. The test modifies an existing file from a lower layer (such as

`/etc/hosts`), then verifies that the original file remains unchanged in the lower layer while the modified version appears in the upper layer. The test also creates new files and confirms they appear only in the upper layer.

Whiteout File Handling tests overlayfs support for file deletion across layers. The test deletes a file that exists in a lower layer and verifies that a whiteout file is created in the upper layer, causing the file to disappear from the merged view while preserving the original file in the lower layer.

Layer Cleanup Verification ensures that overlay mounts and directories are properly removed when containers are destroyed. The test creates a container, allows it to make filesystem changes, then removes the container and verifies that all overlay-related mount points are unmounted and temporary directories are cleaned up. This test also checks that shared lower layers are not removed when they are still referenced by other containers.

Overlay Mount Integrity validates that the overlayfs mount is correctly configured and accessible. The test verifies that the merged directory shows the expected combined view of all layers, that the work directory is properly configured for overlayfs internal operations, and that mount options are correctly applied.

Test Scenario	Setup	Container Action	Verification	Expected Result
Layer combination	3 layers with overlapping files	Read files from all layers	Check file contents	Higher layer files visible
Copy-on-write	Base layer with <code>/etc/passwd</code>	Modify <code>/etc/passwd</code>	Check original layer	Original file unchanged
File creation	Empty upper layer	Create <code>/tmp/newfile</code>	Check layer location	File only in upper layer
File deletion	File in lower layer	Delete file	Check whiteout creation	File hidden, whiteout present
Mount cleanup	Running container	Remove container	Check mount points	All overlay mounts removed

The milestone 3 checkpoint demonstrates that containers can be created with layered filesystems, that copy-on-write semantics work correctly, and that filesystem resources are properly cleaned up. The test creates a container from multiple layers, performs various filesystem operations (read, write, delete), and verifies both the visible behavior and the underlying layer management.

Milestone 4: Container Networking

The fourth milestone implements container networking using veth pairs and bridge networking. Testing focuses on verifying network isolation, container-to-container communication, and port forwarding functionality.

Veth Pair Creation tests that virtual ethernet pairs are correctly created and configured. The test creates a container and verifies that a veth pair connects the container's network namespace to the host, with one end

visible in the container (`eth0`) and the other end visible on the host (with a generated name like `vethXXXXXX`). The test should confirm that both ends of the pair are properly configured with appropriate MAC addresses and are in the UP state.

Bridge Network Integration validates that container veth interfaces are correctly attached to the container bridge and can communicate with other containers. The test creates two containers on the same bridge network, assigns IP addresses from the configured subnet, and verifies that the containers can ping each other using their IP addresses. This test also confirms that containers cannot communicate with external networks unless explicitly configured.

IP Address Management tests that containers receive unique IP addresses from the configured subnet pool and that address conflicts are prevented. The test creates multiple containers and verifies that each receives a different IP address, that addresses are properly allocated and released when containers start and stop, and that reused addresses are not assigned to active containers.

Port Forwarding Configuration validates that NAT rules correctly forward traffic from host ports to container ports. The test creates a container running a simple HTTP server on port 8080, configures port forwarding from host port 9000 to container port 8080, then verifies that HTTP requests to `localhost:9000` on the host are successfully forwarded to the container service.

Network Isolation Verification tests that containers in different network namespaces cannot directly access each other's network resources unless explicitly connected. The test creates containers in separate network namespaces and confirms that they cannot see each other's network interfaces or established connections.

DNS Resolution validates that containers can resolve DNS names using configured nameservers. The test creates a container with DNS servers configured, attempts to resolve a known hostname from inside the container, and verifies that DNS queries are properly forwarded and resolved.

Test Scenario	Network Setup	Test Action	Expected Outcome	Validation Method
Veth pair creation	Single container	Check interfaces	Veth pair exists and configured	<code>ip link</code> in both namespaces
Bridge communication	Two containers on bridge	Ping between containers	Successful ping	<code>ping</code> command returns 0
IP allocation	Multiple containers	Start/stop containers	Unique IPs assigned	Check IP uniqueness
Port forwarding	Container with HTTP server	HTTP request to host port	Request reaches container	HTTP response received
Network isolation	Containers in separate namespaces	Attempt inter-container access	Access blocked	Connection timeout/refused
DNS resolution	Container with nameservers	Resolve external hostname	DNS query succeeds	<code>nslookup</code> returns IP

The milestone 4 checkpoint demonstrates complete container networking functionality by creating multiple containers that can communicate with each other and with external services through port forwarding. The test creates a multi-container application (such as a web server and database) connected through bridge networking, with the web server accessible from the host via port forwarding.

Integration Test Scenarios

Integration testing validates that all milestone components work together correctly in realistic container use cases. These scenarios exercise the complete container lifecycle and test complex interactions between namespace isolation, resource control, filesystem layering, and networking components.

Scenario 1: Multi-Container Web Application

This scenario tests a realistic web application deployment with multiple containers that must coordinate through networking while maintaining proper isolation and resource limits.

Application Architecture: The test deploys a three-tier web application consisting of a web server container (nginx serving static content), an application server container (running a simple HTTP API), and a database container (running a lightweight database like SQLite with HTTP interface). Each container has different resource requirements and network connectivity needs.

Container Configuration: The web server container receives 100MB memory limit and 0.5 CPU cores, with port 80 forwarded to host port 8080. The application server gets 200MB memory and 1.0 CPU core, with port 3000 accessible only from other containers on the bridge network. The database container receives 300MB memory and 0.3 CPU cores, with port 5432 accessible only from the application server.

Test Execution Flow: The test creates all three containers with their respective configurations, starts them in dependency order (database first, then application server, then web server), and verifies that the complete application stack functions correctly. External HTTP requests to the host's port 8080 should flow through the web server to the application server, which queries the database and returns results.

Validation Points: The test validates that each container operates within its resource limits throughout the test duration, that network communication works correctly between containers while external access is properly restricted, that filesystem changes in each container remain isolated, and that all containers can be stopped and cleaned up without leaving orphaned resources.

Failure Scenario Testing: The test intentionally introduces failures such as killing the database container and verifying that the application server handles the connection failure gracefully, exceeding memory limits in one container and confirming that other containers continue operating normally, and network partitioning between containers to test error handling.

Test Phase	Actions	Expected Behavior	Validation Method
Container startup	Start containers in dependency order	All containers reach running state	Check container status
Application connectivity	HTTP request to web server	Request processed through full stack	HTTP response with database data
Resource isolation	Load test individual containers	Each container stays within limits	Monitor cgroup statistics
Network isolation	Direct database access from host	Connection refused	Network connection timeout
Failure recovery	Kill database container	Application server reports database error	Check application server logs
Cleanup verification	Remove all containers	All resources cleaned up	Check for orphaned mounts/processes

Scenario 2: Batch Processing with Resource Contention

This scenario tests container runtime behavior under resource pressure, with multiple containers competing for CPU and memory resources while processing data through shared filesystem layers.

Workload Design: The test creates five containers that perform different types of batch processing: CPU-intensive mathematical calculations, memory-intensive data sorting, I/O-intensive file processing, network-intensive data transfer, and mixed-workload data analysis. Each container has different resource limits that intentionally create contention scenarios.

Resource Configuration: Container limits are set to create controlled resource pressure: total CPU limits exceed available cores by 150%, total memory limits approach system memory capacity, and I/O intensive

containers share the same underlying storage. This configuration tests the runtime's ability to enforce limits and maintain isolation under resource pressure.

Data Flow Testing: Containers process data through a pipeline where the output of one container becomes input for another, implemented through shared filesystem layers and network communication. This tests the integration of filesystem layering with networking while maintaining resource isolation.

Monitoring and Validation: Throughout the test execution, resource usage monitoring validates that containers stay within their configured limits even under pressure, that resource contention in one container does not cause others to exceed their limits, and that the container runtime maintains stable operation during peak resource utilization.

Container	Workload Type	CPU Limit	Memory Limit	Test Validation
calc-1	Mathematical computation	1.5 cores	100MB	CPU usage \leq 1.5 cores
sort-1	Large dataset sorting	0.5 cores	800MB	Memory usage \leq 800MB
io-1	File processing	0.3 cores	50MB	I/O operations complete
net-1	Data transfer	0.2 cores	100MB	Network throughput stable
mixed-1	Combined workload	1.0 cores	300MB	All limits respected

Scenario 3: Container Lifecycle Stress Testing

This scenario exercises the complete container lifecycle under high frequency operations, testing the runtime's stability and resource cleanup capabilities.

High-Frequency Operations: The test rapidly creates, starts, stops, and removes containers in various patterns: sequential operations (create-start-stop-remove cycles), parallel operations (multiple containers created simultaneously), and mixed operations (some containers running long-term while others cycle rapidly).

Resource Exhaustion Testing: The test pushes the runtime toward various resource limits: creating containers until approaching the maximum number of namespaces, allocating IP addresses until the subnet pool is nearly exhausted, creating filesystem layers until storage space becomes constrained, and generating network traffic until bandwidth limits are reached.

Error Recovery Validation: The test intentionally introduces various failure conditions during container lifecycle operations: killing container processes during startup, interrupting filesystem mount operations, simulating network interface failures, and forcing unclean container shutdowns to test cleanup and recovery mechanisms.

Concurrency and Race Condition Testing: Multiple test threads perform container operations simultaneously to expose potential race conditions in resource allocation, namespace creation, cgroup management, and network configuration.

Test Pattern	Operation Count	Concurrency Level	Success Criteria	Failure Detection
Rapid cycling	100 containers	10 parallel threads	All operations succeed	Check for resource leaks
Long-running mixed	50 containers	5 threads, mixed lifecycle	System remains stable	Monitor system resources
Resource exhaustion	Until limits reached	Single thread	Graceful limit handling	Operations fail cleanly
Failure injection	20 containers	Random failures	Proper error recovery	No orphaned resources
Race condition testing	30 containers	20 parallel threads	Consistent resource state	Check for conflicts

Scenario 4: Security Isolation Validation

This scenario tests the security boundaries created by namespace isolation, ensuring that containers cannot escape their isolation or interfere with host system security.

Privilege Escalation Prevention: The test attempts various privilege escalation techniques from within containers, including attempting to access host filesystem paths outside the container's view, trying to modify host network configuration through namespace boundaries, attempting to access host processes or system resources, and testing user namespace mapping to ensure proper privilege isolation.

Resource Access Boundaries: Containers attempt to access resources beyond their configured limits, including trying to consume more memory than allocated, attempting to access network interfaces outside their namespace, trying to modify filesystem layers that should be read-only, and attempting to access other containers' resources directly.

Information Disclosure Prevention: The test verifies that containers cannot access information they should not see, including host system process information, other containers' filesystem contents, network traffic from other containers, and host system configuration details.

Container-to-Container Isolation: Multiple containers attempt to interfere with each other through various vectors, including filesystem access attempts, network communication outside configured channels, resource exhaustion attacks against other containers, and process interference attempts.

Security Test Category	Attack Vector	Expected Behavior	Validation Method
Filesystem isolation	Access <code>/host/etc/passwd</code>	Access denied or file not found	Check file access results
Process isolation	Kill host process from container	Operation fails	Check process list
Network isolation	Access other container ports	Connection refused	Network connection test
Resource isolation	Exhaust shared resources	Other containers unaffected	Monitor other container performance
User namespace isolation	Access files as different UID	Permission denied	Check file access permissions

Common Pitfalls in Container Runtime Testing

Testing container runtimes presents unique challenges because failures often involve complex interactions between kernel features, filesystem operations, and network configuration. Understanding common pitfalls helps developers create more robust test scenarios and debug issues more effectively.

⚠ Pitfall: Race Conditions in Container Startup Container startup involves multiple asynchronous operations (namespace creation, filesystem mounting, network configuration) that can complete in different orders. Tests that immediately check container state after calling `StartContainer` may fail intermittently because the container hasn't fully initialized. The test should poll container status with a reasonable timeout, or better yet, wait for a specific readiness signal from the container process (such as a service listening on a port or a status file being created).

⚠ Pitfall: Incomplete Resource Cleanup Detection Tests that create and destroy containers rapidly may not detect resource leaks because the effects accumulate slowly. Running a single create-destroy cycle and checking for cleanup appears to work, but running hundreds of cycles reveals leaked mount points, network interfaces, or cgroup directories. Effective testing requires baseline resource measurements before the test, multiple iterations to amplify leak effects, and comprehensive resource scanning after test completion.

⚠ Pitfall: Host System Dependencies Container runtime tests often assume specific host system configurations (kernel version, available namespaces, cgroups v2 enabled, bridge networking support) that may not be available in all test environments. Tests should verify prerequisites before execution and either skip gracefully or provide clear error messages when requirements are not met. Additionally, tests should not assume specific network interface names, IP address ranges, or filesystem paths that may vary between systems.

⚠ Pitfall: Timing-Dependent Resource Limit Validation Resource usage measurements (CPU, memory, I/O) require time to stabilize and may fluctuate based on system load and kernel scheduling decisions. Tests that immediately check resource usage after starting a workload may see inconsistent results. Effective

resource limit testing requires sustained workloads running for sufficient time periods, multiple measurements averaged together, and appropriate tolerance ranges that account for measurement variance while still detecting limit violations.

⚠ Pitfall: Network Configuration Interference Container networking tests can interfere with each other and with host networking configuration, especially when creating bridges, assigning IP addresses, or configuring iptables rules. Tests running in parallel may create conflicting network configurations or exhaust IP address pools. Robust network testing requires isolated network namespaces for test execution, unique IP address ranges for each test, and thorough cleanup of iptables rules and network interfaces.

⚠ Pitfall: Filesystem Layer Corruption Overlayfs operations can leave the filesystem in inconsistent states if interrupted during critical operations like mounting or unmounting. Tests that kill container processes or force unclean shutdowns may corrupt overlay layers, affecting subsequent tests. Proper testing requires careful cleanup of overlay mounts even after failures, validation that overlay directories are in expected states, and potentially rebuilding overlay layers if corruption is detected.

Implementation Guidance

The container runtime testing implementation requires a systematic approach that validates each milestone's functionality in isolation and then tests integrated scenarios that exercise multiple components simultaneously.

Technology Recommendations

Component	Simple Option	Advanced Option
Test Framework	Go testing package with table-driven tests	Ginkgo BDD framework with custom matchers
Test Isolation	Temporary directories with UUID names	Linux user namespaces for complete test isolation
Resource Monitoring	Parse /proc and /sys files directly	Prometheus client library with custom metrics
Network Testing	Simple ping and HTTP requests	Packet capture analysis with gopacket library
Filesystem Validation	Directory listing and file content checks	inotify-based filesystem change monitoring
Container Orchestration	Sequential test execution	Parallel test execution with resource pools

Recommended File Structure

```
project-root/  
  GO  
  
  cmd/container-runtime/  
    main.go          ← CLI entry point  
  
    internal/runtime/  
      manager.go  
      container.go  
  
    internal/namespace/  
      namespace.go  
      namespace_test.go   ← Unit tests  
  
    internal/cgroup/  
      cgroup.go  
      cgroup_test.go    ← Unit tests  
  
    internal/filesystem/  
      overlay.go  
      overlay_test.go   ← Unit tests  
  
    internal/network/  
      network.go  
      network_test.go   ← Unit tests  
  
  test/  
    integration/  
      milestone1_test.go    ← Namespace isolation tests  
      milestone2_test.go    ← Resource control tests  
      milestone3_test.go    ← Filesystem layering tests  
      milestone4_test.go    ← Container networking tests  
  
    e2e/  
      webapp_test.go      ← Multi-container web app
```

```
batch_test.go           ← Batch processing scenario  
lifecycle_test.go      ← Lifecycle stress testing  
security_test.go        ← Security isolation validation  
fixtures/               ← Test data and images  
test-image/              ← Simple container image layers  
workloads/               ← Test programs and scripts  
helpers/                ← Testing utilities  
test_utils.go            ← Common testing functions  
resource_monitor.go      ← Resource validation helpers  
network_helpers.go       ← Network testing utilities
```

Test Infrastructure Starter Code

```
// test/helpers/test_utils.go                                     GO

package helpers

import (
    "fmt"
    "os"
    "path/filepath"
    "testing"
    "time"
    "github.com/google/uuid"
)

// TestEnvironment provides isolated testing environment for container runtime tests

type TestEnvironment struct {
    TempDir      string
    ContainerID string
    Cleanup      []func() error
}

// NewTestEnvironment creates isolated test environment with unique IDs and cleanup

func NewTestEnvironment(t *testing.T) *TestEnvironment {
    tempDir := filepath.Join("/tmp", "container-test-"+uuid.New().String())
    if err := os.MkdirAll(tempDir, 0755); err != nil {
        t.Fatalf("Failed to create temp directory: %v", err)
    }

    env := &TestEnvironment{

```

```
    TempDir:      tempDir,
    ContainerID: "test-" + uuid.New().String()[:8],
    Cleanup:      make([]func() error, 0),
}

// Register cleanup for test completion

t.Cleanup(func() {
    env.CleanupAll()
})

return env
}

// AddCleanup registers cleanup function to be called when test completes

func (te *TestEnvironment) AddCleanup(cleanup func() error) {
    te.Cleanup = append(te.Cleanup, cleanup)
}

// CleanupAll executes all registered cleanup functions

func (te *TestEnvironment) CleanupAll() {
    for i := len(te.Cleanup) - 1; i >= 0; i-- {
        if err := te.Cleanup[i](); err != nil {
            fmt.Printf("Cleanup error: %v\n", err)
        }
    }
    os.RemoveAll(te.TempDir)
}
```

```
// WaitForCondition polls condition function until it returns true or timeout

func WaitForCondition(condition func() bool, timeout time.Duration, interval time.Duration) bool {
    deadline := time.Now().Add(timeout)

    for time.Now().Before(deadline) {
        if condition() {
            return true
        }

        time.Sleep(interval)
    }

    return false
}
```

GO

```
// test/helpers/resource_monitor.go

package helpers

import (
    "bufio"
    "fmt"
    "os"
    "path/filepath"
    "strconv"
    "strings"
    "time"
)

// ResourceSnapshot captures resource usage at a point in time

type ResourceSnapshot struct {
    Timestamp    time.Time
    MemoryUsage int64 // bytes
    CPUUsage    int64 // nanoseconds
    PIDCount    int64 // number of processes
}

// ResourceMonitor tracks container resource usage over time

type ResourceMonitor struct {
    cgroupPath string
    snapshots []ResourceSnapshot
}

// NewResourceMonitor creates monitor for specified cgroup path

func NewResourceMonitor(cgroupPath string) *ResourceMonitor {
```

```
    return &ResourceMonitor{  
  
        cgroupPath: cgroupPath,  
  
        snapshots: make([]ResourceSnapshot, 0),  
    }  
}  
  
// TakeSnapshot captures current resource usage values  
  
func (rm *ResourceMonitor) TakeSnapshot() error {  
  
    snapshot := ResourceSnapshot{  
  
        Timestamp: time.Now(),  
    }  
  
    // Read memory usage from cgroup  
  
    memoryFile := filepath.Join(rm.cgroupPath, "memory.current")  
  
    if data, err := os.ReadFile(memoryFile); err == nil {  
  
        if usage, err := strconv.ParseInt(strings.TrimSpace(string(data)), 10, 64); err ==  
nil {  
  
            snapshot.MemoryUsage = usage  
        }  
    }  
  
    // Read CPU usage from cgroup  
  
    cpuFile := filepath.Join(rm.cgroupPath, "cpu.stat")  
  
    if file, err := os.Open(cpuFile); err == nil {  
  
        defer file.Close()  
  
        scanner := bufio.NewScanner(file)  
  
        for scanner.Scan() {  
  
            line := scanner.Text()  
        }  
    }  
}
```

```
        if strings.HasPrefix(line, "usage_usec ") {

            if usec, err := strconv.ParseInt(strings.Fields(line)[1], 10, 64); err ==
nil {

                snapshot.CPUUsage = usec * 1000 // convert to nanoseconds

            }

            break

        }

    }

}

// Read process count from cgroup

pidsFile := filepath.Join(rm.cgroupPath, "pids.current")

if data, err := os.ReadFile(pidsFile); err == nil {

    if count, err := strconv.ParseInt(strings.TrimSpace(string(data)), 10, 64); err ==
nil {

        snapshot.PIDCount = count

    }

}

rm.snapshots = append(rm.snapshots, snapshot)

return nil

}

// GetAverageUsage returns average resource usage over monitoring period

func (rm *ResourceMonitor) GetAverageUsage(duration time.Duration) ResourceSnapshot {

    cutoff := time.Now().Add(-duration)

    var memorySum, cpuSum, pidSum int64

    var count int64
```

```
for _, snapshot := range rm.snapshots {

    if snapshot.Timestamp.After(cutoff) {

        memorySum += snapshot.MemoryUsage

        cpuSum += snapshot.CPUUsage

        pidSum += snapshot.PIDCount

        count++

    }

}

if count == 0 {

    return ResourceSnapshot{}

}

return ResourceSnapshot{

    Timestamp:    time.Now(),

    MemoryUsage:  memorySum / count,

    CPUUsage:    cpuSum / count,

    PIDCount:   pidSum / count,

}

}
```

Milestone Test Implementation Skeletons

```
// test/integration/milestone1_test.go
```

GO

```
package integration
```

```
import (
```

```
    "testing"
```

```
    "time"
```

```
    "your-project/internal/runtime"
```

```
    "your-project/test/helpers"
```

```
)
```

```
func TestNamespaceIsolation(t *testing.T) {
```

```
    env := helpers.NewTestEnvironment(t)
```

```
    runtimeManager := runtime.NewManager()
```

```
    // TODO 1: Create container spec with namespace configuration
```

```
    // - Enable PID, mount, network, UTS namespaces
```

```
    // - Set container hostname to differentiate from host
```

```
    // - Configure basic command like ["sleep", "30"]
```

```
    // TODO 2: Create and start container
```

```
    // - Call CreateContainer with spec
```

```
    // - Call StartContainer to begin process execution
```

```
    // - Verify container reaches ContainerRunning state
```

```
    // TODO 3: Validate PID namespace isolation
```

```
    // - Execute command in container to check PID 1
```

```
    // - Compare process list inside container vs host
```

```
// - Verify container process appears with different PID on host

// TODO 4: Validate mount namespace isolation

// - Create temporary mount inside container

// - Verify mount is not visible on host

// - Check that /proc filesystem is properly mounted in container


// TODO 5: Validate network namespace isolation

// - Compare network interface list inside container vs host

// - Verify container cannot see host network connections

// - Check that container has its own loopback interface


// TODO 6: Validate UTS namespace isolation

// - Check hostname inside container matches configured value

// - Verify host hostname is unchanged

// - Test that hostname change in container doesn't affect host


// TODO 7: Cleanup and verify

// - Stop and remove container

// - Verify all namespace resources are cleaned up

// - Check for any orphaned processes or mounts

}

func TestUserNamespaceMapping(t *testing.T) {

    // TODO: Implement user namespace UID/GID mapping tests

    // - Create container with user namespace enabled

    // - Configure UID mapping (container root -> host unprivileged user)
```

```
// - Verify file ownership appears correctly from both perspectives  
// - Test privilege boundaries are enforced  
}
```

GO

```
// test/integration/milestone2_test.go

package integration

import (
    "testing"
    "time"
    "your-project/internal/runtime"
    "your-project/test/helpers"
)

func TestResourceLimits(t *testing.T) {
    env := helpers.NewTestEnvironment(t)

    runtimeManager := runtime.NewManager()

    monitor := helpers.NewResourceMonitor("")

    testCases := []struct {
        name      string
        memoryLimit int64
        cpuLimit   int64
        workload   string
        expectOOM  bool
    }{
        {
            name:           "memory_limit_enforced",
            memoryLimit: 50 * 1024 * 1024, // 50MB
            cpuLimit:     1000000000,       // 1 CPU core
            workload:     "allocate_100mb",
            expectOOM:    true,
        },
    }
}
```

```
        },

        {

            name:          "cpu_limit_enforced",

            memoryLimit:  200 * 1024 * 1024, // 200MB

            cpuLimit:     5000000000,           // 0.5 CPU cores

            workload:     "cpu_intensive",

            expectOOM:    false,

        },

    }

}

for _, tc := range testCases {

    t.Run(tc.name, func(t *testing.T) {

        // TODO 1: Create container spec with resource limits

        // - Set memory limit to tc.memoryLimit

        // - Set CPU limit to tc.cpuLimit

        // - Configure workload command based on tc.workload

        // TODO 2: Create and start container with monitoring

        // - Create container with resource-limited spec

        // - Start resource monitoring before container start

        // - Begin container execution

        // TODO 3: Monitor resource usage during workload execution

        // - Take resource snapshots every 100ms for test duration

        // - Track memory usage, CPU usage, and process count

        // - Calculate average usage over test period

    })

}
```

```
// TODO 4: Validate resource limit enforcement

// - Check that memory usage stays within limits (unless OOM expected)

// - Verify CPU usage respects configured quota

// - Confirm process count stays within PID limits


// TODO 5: Handle OOM scenarios if expected

// - Check for OOM killer activation if tc.expectOOM is true

// - Verify container exit code indicates OOM termination

// - Confirm other system processes are not affected


// TODO 6: Cleanup and verify resource release

// - Stop container and verify graceful resource cleanup

// - Check that cgroup directories are removed

// - Confirm system resource usage returns to baseline

})

}

}

func TestResourceMonitoring(t *testing.T) {

// TODO: Implement resource usage monitoring accuracy tests

// - Create containers with known resource consumption patterns

// - Compare reported usage statistics against expected values

// - Test monitoring update frequency and accuracy

// - Validate alert thresholds and notifications

}
```

Milestone Checkpoint Commands

```
# Milestone 1: Namespace Isolation                                BASH

go test ./test/integration/milestone1_test.go -v

# Expected output should show:

# - Container created with isolated namespaces

# - PID 1 inside container is container process

# - Network interfaces differ between container and host

# - Hostname isolation working correctly

# - All namespace resources cleaned up after test

# Manual verification commands:

sudo ./container-runtime create test-container --image=test-image --command="sleep 60"

sudo ./container-runtime start test-container

# In separate terminal, check namespace isolation:

sudo ./container-runtime exec test-container ps aux          # Should show only container
processes

sudo ./container-runtime exec test-container hostname        # Should show container
hostname

sudo ./container-runtime exec test-container ip link list    # Should show container
interfaces

# Milestone 2: Resource Control

go test ./test/integration/milestone2_test.go -v

# Expected behavior:

# - Containers respect memory limits (OOM kill when exceeded)

# - CPU usage stays within configured quotas

# - Resource monitoring reports accurate usage statistics

# - Cgroup cleanup occurs properly after container removal
```

```
# Manual verification:

sudo ./container-runtime create limited-container --memory=50m --cpu=0.5 --command="stress-test"

sudo ./container-runtime start limited-container

# Monitor resource usage:

cat /sys/fs/cgroup/container-runtime/limited-container/memory.current

cat /sys/fs/cgroup/container-runtime/limited-container/cpu.stat

# Milestone 3: Filesystem Layering

go test ./test/integration/milestone3_test.go -v

# Expected results:

# - Overlay filesystem properly combines multiple layers

# - Copy-on-write behavior creates files in upper layer

# - File modifications don't affect read-only base layers

# - Layer cleanup removes overlay mounts and directories

# Manual verification:

sudo ./container-runtime create layered-container --image=multi-layer-image

findmnt | grep overlay                                # Should show overlay mount

sudo ./container-runtime exec layered-container touch /test-file

ls /var/lib/container-runtime/containers/*/upper/    # Should contain test-file

# Milestone 4: Container Networking

go test ./test/integration/milestone4_test.go -v

# Expected functionality:

# - Veth pairs connect containers to bridge network

# - Containers can communicate using IP addresses
```

```
# - Port forwarding enables external access to services

# - Network isolation prevents unauthorized access

# Manual verification:

sudo ./container-runtime create web-server --port-mapping=8080:80 --command="simple-http-server"

sudo ./container-runtime create client --command="sleep 60"

sudo ./container-runtime start web-server

sudo ./container-runtime start client

# Test connectivity:

curl http://localhost:8080                                # Should reach container web server

sudo ./container-runtime exec client ping <web-server-ip>  # Should succeed
```

The testing implementation provides comprehensive validation of each milestone's functionality while building toward realistic integration scenarios. The progressive testing approach ensures that developers can verify each component works correctly before integrating with dependent components, making debugging more manageable and building confidence in the overall system design.

Debugging Guide

Milestone(s): All milestones (1-4) - This section provides debugging techniques for namespace isolation (milestone 1), resource control (milestone 2), filesystem layering (milestone 3), and network management (milestone 4). Effective debugging requires understanding both the kernel-level mechanisms and the container runtime's abstraction layers.

Building a container runtime involves orchestrating multiple complex Linux kernel mechanisms that can fail in subtle ways. When a container fails to start, exhibits unexpected behavior, or consumes incorrect resources, the root cause could lie in namespace creation, cgroup configuration, filesystem mounting, or network setup. This debugging guide provides systematic approaches for diagnosing problems in each component, using kernel inspection tools and container runtime state examination.

Mental Model: Detective Investigation

Think of debugging a container runtime like investigating a crime scene. Each kernel mechanism (namespaces, cgroups, overlayfs, network interfaces) leaves traces in the system that tell a story about what

happened. Just as a detective follows evidence from multiple sources to reconstruct events, we use kernel state inspection tools, log analysis, and systematic testing to understand why containers behave unexpectedly. The key is knowing where to look for evidence and how to interpret what we find.

The debugging process follows a systematic approach: first observe symptoms from the container's perspective, then examine the underlying kernel state, identify discrepancies between expected and actual configuration, and finally trace the root cause back through the container runtime's component interactions. Each component leaves different types of evidence in different locations within the kernel's virtual filesystems.

Namespace Debugging Techniques

Namespace issues manifest as containers seeing unexpected processes, having incorrect filesystem views, or lacking proper isolation from the host system. These problems typically stem from incorrect namespace creation, failed filesystem operations, or improper process assignment to namespaces.

Understanding Namespace State Inspection

The Linux kernel exposes namespace information through the `/proc` filesystem, allowing us to inspect which namespaces exist and which processes belong to them. Each process has namespace information in `/proc/[pid]/ns/` that shows symbolic links to namespace identifiers. When containers share namespaces unexpectedly or fail to create new namespaces, this directory provides the ground truth about actual kernel state.

The `lsns` command provides a high-level view of all namespaces on the system, showing which processes belong to each namespace type. This tool helps identify whether container processes are actually isolated or accidentally sharing host namespaces. For example, if a container process appears in the same PID namespace as the host, the namespace creation failed silently.

Common Namespace Debugging Scenarios

Symptom	Likely Cause	Diagnostic Commands	Fix Approach
Container sees all host processes	PID namespace creation failed	<code>lsns -t pid</code> , <code>ls -la /proc/[pid]/ns/</code>	Check namespace creation flags, verify CLONE_NEWPID
Container filesystem shows host files	Mount namespace creation failed or pivot_root failed	<code>findmnt</code> , <code>cat /proc/[pid]/mountinfo</code>	Verify mount namespace, check pivot_root sequence
Container has host hostname	UTS namespace not created or not configured	<code>unshare -u hostname</code> , check <code>/proc/[pid]/ns/uts</code>	Ensure UTS namespace creation and hostname setting
Container networking uses host interfaces	Network namespace creation failed	<code>ip netns list</code> , <code>ip link show</code> in container	Verify network namespace creation and veth setup
Permission denied in container	User namespace mapping incorrect	<code>cat /proc/[pid]/uid_map</code> , <code>cat /proc/[pid]/gid_map</code>	Fix UID/GID mapping configuration

Process and Mount Inspection Techniques

When containers exhibit incorrect process visibility, the first step is determining which PID namespace the container process actually belongs to. The `/proc/[pid]/ns/pid` symlink shows the namespace identifier, which should be unique for each container. If multiple containers or the host share the same PID namespace identifier, namespace isolation has failed.

Mount namespace problems require examining the actual mount table that the container process sees. The `/proc/[pid]/mountinfo` file shows all mount points visible to a specific process, including overlay filesystem mounts, bind mounts, and special filesystems like `/proc` and `/sys`. Comparing the container's mount table with the expected configuration reveals whether filesystem isolation is working correctly.

The `pivot_root` operation is particularly error-prone and leaves specific evidence when it fails. If `pivot_root` fails, the container process will still see the host root filesystem instead of the container's overlay filesystem. The mount table will show the original host mounts instead of the expected container filesystem hierarchy.

Network Namespace Inspection

Network namespace isolation problems manifest as containers having unexpected network interfaces or connectivity. The `ip netns` command lists all named network namespaces, but container network namespaces are typically unnamed and must be accessed through process namespace references.

To inspect a container's network configuration, use `nsenter` to execute network commands within the container's namespace: `nsenter -t [pid] -n ip addr show` reveals which interfaces exist inside the

container. A properly isolated container should only see its container interface and the loopback interface, not host interfaces.

Network namespace debugging also involves verifying veth pair creation and bridge attachment. Each veth pair creates two linked interfaces, with one end in the host namespace and one in the container namespace. The `ip link show` command in both namespaces should show the paired interfaces with matching indices.

File Permission and User Namespace Issues

User namespace problems typically manifest as permission denied errors when the container tries to access files or perform operations that should be allowed. These issues stem from incorrect UID/GID mapping between the container and host user namespaces.

The `/proc/[pid]/uid_map` and `/proc/[pid]/gid_map` files show the current mapping configuration. Each line specifies a range mapping: container ID range, host ID range, and count. Incorrect mappings cause the container process to have unexpected privileges or lack necessary permissions.

User namespace debugging requires understanding the interaction between namespace isolation and filesystem permissions. Files created by the container appear on the host filesystem with the mapped host UID/GID, which must align with the host user running the container runtime.

⚠ Pitfall: Empty /proc Directory in Container

One of the most common namespace issues occurs when the container's `/proc` directory is empty or shows host processes instead of container processes. This happens when the mount namespace is created correctly, but `/proc` is not remounted within the container's PID namespace.

The fix requires mounting a new `/proc` filesystem after creating the PID namespace but before executing the container process:

```
# Inside container, after PID namespace creation:  
mount -t proc proc /proc
```

BASH

This issue is particularly confusing because the container has a separate PID namespace (process isolation works), but tools like `ps` fail because they can't read process information from `/proc`.

⚠ Pitfall: pivot_root Requires Mount Point

The `pivot_root` system call fails with "Invalid argument" if the new root directory is not a mount point. This is a common error when trying to pivot to a regular directory instead of a mounted filesystem.

The solution is ensuring the new root is a mount point before calling `pivot_root`. For overlay filesystems, this means the overlay mount operation must complete successfully before attempting the filesystem pivot. Always verify mount success before proceeding to `pivot_root`.

Namespace Debugging Command Reference

Command	Purpose	Example Usage
<code>lsns</code>	List all namespaces	<code>lsns -t pid -p [container-pid]</code>
<code>nsenter</code>	Execute command in namespace	<code>nsenter -t [pid] -a ps aux</code>
<code>unshare</code>	Create namespace for testing	<code>unshare -p -f --mount-proc bash</code>
<code>findmnt</code>	Show mount table	<code>findmnt -N [pid]</code>
<code>cat /proc/[pid]/ns/*</code>	Show namespace IDs	Compare container vs host namespace IDs

Cgroup Debugging Techniques

Cgroup-related problems manifest as containers exceeding resource limits, failing to start due to permission issues, or resource monitoring returning incorrect values. These issues typically stem from incorrect cgroup hierarchy setup, improper controller configuration, or permission problems with cgroup filesystem access.

Understanding Cgroup State Inspection

The cgroups v2 unified hierarchy exposes all configuration and statistics through the `/sys/fs/cgroup` filesystem. Each container gets its own cgroup directory containing controller configuration files and runtime statistics. Understanding the relationship between cgroup hierarchy, controller enablement, and process assignment is crucial for effective debugging.

The `systemd-cgls` command provides a tree view of the entire cgroup hierarchy, showing which processes belong to each cgroup. This tool helps identify whether container processes are assigned to the correct cgroups and whether cgroup nesting is configured properly.

Cgroup Hierarchy and Controller Verification

Component	File Location	Purpose	Diagnostic Command
Cgroup membership	/sys/fs/cgroup/[path]/cgroup.procs	Lists PIDs in cgroup	cat cgroup.procs
Enabled controllers	/sys/fs/cgroup/cgroup.controllers	Shows available controllers	cat cgroup.controllers
Memory limit	/sys/fs/cgroup/[path]/memory.max	Current memory limit	cat memory.max
Memory usage	/sys/fs/cgroup/[path]/memory.current	Current memory consumption	cat memory.current
CPU quota	/sys/fs/cgroup/[path]/cpu.max	CPU quota and period	cat cpu.max
OOM events	/sys/fs/cgroup/[path]/memory.events	OOM kill statistics	grep oom_kill memory.events

The cgroup hierarchy must be properly structured for controllers to function correctly. Each container should have its own subdirectory under a parent cgroup where the necessary controllers are enabled. The `cgroup.subtree_control` file in parent directories controls which controllers are available to child cgroups.

Memory Controller Debugging

Memory limit problems often manifest as containers being killed unexpectedly or consuming more memory than expected. The memory controller provides detailed statistics about memory usage patterns, pressure indicators, and out-of-memory events.

The `memory.current` file shows instantaneous memory usage, while `memory.max` shows the configured limit. If usage consistently approaches the limit, the container may be experiencing memory pressure. The `memory.pressure` file provides pressure stall information indicating when memory allocation is being delayed.

Out-of-memory events are recorded in `memory.events`, which tracks both OOM kills and memory pressure events. When a container is killed due to memory exhaustion, this file shows the `oom_kill` count incrementing. This helps distinguish between application crashes and resource limit enforcement.

CPU Controller Debugging

CPU limit issues manifest as containers running slower than expected or consuming more CPU than allocated. The CPU controller uses quota and period values to implement CPU time limits, which can be confusing to interpret and debug.

The `cpu.max` file contains two values: quota and period (e.g., "100000 100000" means 100ms quota per 100ms period, allowing 100% CPU). The `cpu.stat` file shows actual CPU usage statistics, including throttling events when the quota is exceeded.

CPU throttling debugging requires comparing the configured quota with actual usage patterns. High values in `cpu.stat` for `throttled_usec` indicate the container is being actively throttled. This may be expected behavior if the limit is working correctly, or indicate misconfigured limits if the container should have more CPU access.

Common Cgroup Debugging Scenarios

Symptom	Likely Cause	Investigation Steps	Resolution
Container immediately killed	Memory limit too low or OOM kill	Check <code>memory.events</code> for oom_kill, compare <code>memory.max</code> vs <code>memory.current</code>	Increase memory limit or optimize container memory usage
Container runs slowly	CPU throttling or insufficient quota	Check <code>cpu.stat</code> for throttling, verify <code>cpu.max</code> configuration	Adjust CPU quota or identify CPU-intensive operations
Resource monitoring shows zero usage	Cgroup not created or process not assigned	Verify cgroup directory exists, check <code>cgroup.procs</code> contains container PID	Fix cgroup creation or process assignment
Permission denied writing cgroup files	Insufficient privileges or controller not enabled	Check file ownership, verify controller in <code>cgroup.subtree_control</code>	Run with appropriate privileges or enable controllers
Cgroup cannot be removed	Processes still in cgroup or child cgroups exist	Check <code>cgroup.procs</code> is empty, verify no subdirectories	Kill remaining processes or remove child cgroups first

⚠ Pitfall: Cgroups v1 vs v2 Interface Differences

Many debugging approaches fail because they assume cgroups v1 interfaces while the system uses cgroups v2. The file locations, interface syntax, and controller behaviors differ significantly between versions.

To determine which version is active, check `/proc/mounts` for cgroup filesystem types. Cgroups v2 shows `cgroup2` filesystem type mounted at `/sys/fs/cgroup`, while v1 shows separate `cgroup` mounts for each controller.

The debugging commands and file paths in this guide assume cgroups v2. For systems still using v1, controller files are located under separate mount points like `/sys/fs/cgroup/memory/` and `/sys/fs/cgroup/cpu/`.

⚠ Pitfall: Controller Delegation Requirements

Cgroups v2 requires explicit controller delegation from parent to child cgroups. If a controller is not enabled in the parent's `cgroup.subtree_control`, child cgroups cannot use that controller even if it's available at the system level.

To fix delegation issues, enable controllers at each level of the hierarchy: `echo "+memory +cpu" > /sys/fs/cgroup/[parent]/cgroup.subtree_control`. This must be done recursively for all parent cgroups up to the container's cgroup.

Resource Monitoring and Statistics Interpretation

Effective cgroup debugging requires understanding how to interpret the statistics and pressure indicators provided by each controller. The statistics files update continuously and provide both instantaneous values and cumulative counters.

Memory statistics include detailed breakdowns of different memory types: anonymous memory, file cache, shared memory, and kernel memory. Understanding these categories helps identify whether memory usage is due to application data, file system cache, or kernel overhead.

CPU statistics track not just usage time but also scheduling fairness, voluntary context switches, and throttling events. High involuntary context switch counts may indicate CPU contention, while excessive throttling suggests quota adjustments are needed.

Cgroup Debugging Tool Reference

Tool	Purpose	Key Usage
<code>systemd-cgls</code>	Display cgroup hierarchy	<code>systemd-cgls -u [container-service]</code>
<code>systemd-cgtop</code>	Live cgroup resource usage	<code>systemd-cgtop</code>
<code>cat /sys/fs/cgroup/*/cgroup.*</code>	Inspect cgroup configuration	Check membership, controllers, limits
<code>echo [pid] > cgroup.procs</code>	Manually assign process	For testing process assignment
<code>cgexec</code>	Execute command in cgroup	<code>cgexec -g memory:/test/container command</code>

Network Debugging Techniques

Container networking problems manifest as connectivity failures, incorrect IP addresses, unreachable services, or port mapping not working. These issues typically involve veth pair creation, bridge configuration, IP address assignment, or iptables rule management.

Understanding Container Network Architecture

Container networking creates a complex topology involving multiple network namespaces, virtual interfaces, bridges, and NAT rules. Each component can fail independently, making systematic debugging essential. The key is understanding the packet path from external clients through the host networking stack to the container application.

A typical packet flow involves: external client → host interface → iptables DNAT → bridge → veth pair → container interface → container application. Each step in this path can be verified independently using standard Linux networking tools.

Network Interface and Bridge Inspection

Component	Host Command	Container Command	What to Verify
Veth pair existence	<code>ip link show</code>	<code>nsenter -t [pid] -n ip link show</code>	Both ends exist and are UP
Bridge membership	<code>bridge link show</code>	N/A	Container veth attached to bridge
IP address assignment	<code>bridge fdb show</code>	<code>nsenter -t [pid] -n ip addr show</code>	Container has correct IP
Routing configuration	<code>ip route show</code>	<code>nsenter -t [pid] -n ip route show</code>	Default route points to bridge gateway
Interface statistics	<code>ip -s link show [interface]</code>	Same inside container	Check for packet drops or errors

Network interface debugging starts with verifying that veth pairs are created correctly and both ends are operational. The `ip link show` command should show the host end of the veth pair with state UP, and the corresponding interface inside the container namespace should also be UP with the expected name.

Bridge configuration requires verifying that the container's veth interface is properly attached to the container bridge. The `bridge link show` command displays all interfaces attached to bridges, and the container interface should appear in this list with the correct bridge association.

IP Address and Routing Verification

IP address assignment problems cause containers to have incorrect addresses or duplicate addresses that conflict with other containers. The container's IP address should be unique within the bridge subnet and properly configured with the correct netmask and gateway.

Within the container namespace, `ip addr show` should display the assigned IP address on the container interface. The default route should point to the bridge gateway IP address, enabling communication with other containers and external networks.

Address resolution (ARP) problems can cause connectivity issues even when IP configuration appears correct. The `ip neigh show` command displays the ARP table, which should contain entries for the gateway and other containers that have been contacted.

Port Forwarding and NAT Rule Debugging

Port mapping relies on iptables NAT rules to forward traffic from host ports to container ports. These rules span multiple iptables tables and chains, making debugging complex but systematic.

Table	Chain	Rule Type	Purpose	Verification Command
nat	PREROUTING	DNAT	Forward host port to container	<code>iptables -t nat -L PREROUTING -n -v</code>
filter	FORWARD	ACCEPT	Allow forwarded traffic	<code>iptables -L FORWARD -n -v</code>
nat	POSTROUTING	MASQUERADE	Source NAT for outbound	<code>iptables -t nat -L POSTROUTING -n -v</code>

NAT rule debugging requires checking that destination NAT (DNAT) rules exist for each port mapping and that the rules correctly translate host IP:port to container IP:port. The rule should appear in the PREROUTING chain with the correct target specification.

Forward rules in the filter table must allow traffic to pass between the host and container interfaces. Missing ACCEPT rules in the FORWARD chain cause connections to be dropped even when NAT rules are correctly configured.

Common Network Debugging Scenarios

Symptom	Likely Cause	Diagnostic Steps	Resolution
Container has no network interface	Network namespace creation failed or veth not moved	Check <code>ip link</code> in container, verify network namespace	Recreate veth pair and move to namespace
Container cannot reach other containers	Bridge not configured or IP conflict	Check bridge membership, verify unique IPs	Fix bridge setup or resolve IP conflict
External clients cannot reach container service	Port forwarding rules missing or incorrect	Check iptables NAT rules, test with telnet	Add or fix DNAT rules in iptables
Container cannot reach external services	No default route or DNS misconfiguration	Check routing table in container, test DNS	Configure default route and DNS servers
Intermittent connectivity issues	MTU mismatch or packet fragmentation	Check MTU on all interfaces, capture packets	Set consistent MTU values across path

⚠ Pitfall: Veth Interface Names and Namespace Movement

Veth interfaces have confusing naming behavior when moved between namespaces. Creating a veth pair with specific names and then moving one end to a container namespace can result in unexpected interface names inside the container.

The solution is to rename the container interface after moving it to the target namespace: `ip link set [old-name] name [new-name]`. This ensures consistent interface naming regardless of the kernel's automatic name generation.

Additionally, attempting to delete a veth pair from the wrong namespace can fail silently, leaving interfaces in an inconsistent state. Always delete veth pairs from the namespace containing the interface you're targeting.

⚠ Pitfall: iptables Rule Persistence and Cleanup

Container runtime crashes or improper shutdown can leave iptables rules in place, causing port conflicts when restarting containers with the same port mappings. These orphaned rules can prevent new containers from binding to the same ports.

The fix requires explicitly cleaning up iptables rules during container removal: `iptables -t nat -D PREROUTING -p tcp --dport [host-port] -j DNAT --to [container-ip]:[container-port]`. Implementing proper cleanup ensures that container removal completely reverses the network configuration changes.

Network Performance and Packet Analysis

Network debugging often requires analyzing actual packet flow to identify where communication fails. The `tcpdump` tool can capture packets on any interface, including veth interfaces and bridges, showing exactly what traffic is flowing and where it stops.

Capturing packets on both ends of a veth pair reveals whether traffic is reaching the container interface and whether responses are being sent back. The command `tcpdump -i [interface] -n` shows packet headers without hostname resolution, making it easier to verify IP addresses and ports.

Bridge packet analysis helps identify forwarding issues: `tcpdump -i [bridge-name] -n` captures all traffic crossing the bridge, including inter-container communication and traffic to/from the host namespace.

Network Debugging Tool Reference

Tool	Purpose	Key Usage Examples
<code>ip</code>	Interface and routing management	<code>ip link</code> , <code>ip addr</code> , <code>ip route</code> , <code>ip netns</code>
<code>bridge</code>	Bridge configuration inspection	<code>bridge link show</code> , <code>bridge fdb show</code>
<code>iptables</code>	Firewall and NAT rule management	<code>iptables -L -n -v</code> , <code>iptables -t nat -L</code>
<code>nsenter</code>	Execute commands in network namespace	<code>nsenter -t [pid] -n [command]</code>
<code>tcpdump</code>	Packet capture and analysis	<code>tcpdump -i [interface] -n</code>
<code>ss</code>	Socket statistics and connection state	<code>ss -t lnp</code> (listening sockets), <code>ss -an</code> (all connections)

Implementation Guidance

The debugging techniques described above require systematic application during container runtime development. Effective debugging combines automated testing with manual inspection tools to identify problems early and understand their root causes.

Technology Recommendations for Debugging Infrastructure

Component	Simple Option	Advanced Option
Log Analysis	Text logs with grep/awk	Structured logging with JSON + log aggregation
State Inspection	Manual shell commands	Automated scripts with health checks
Network Testing	ping/curl commands	Packet capture with automated analysis
Resource Monitoring	Manual /sys/fs/cgroup inspection	Prometheus metrics collection
Error Tracking	Function return values	Distributed tracing with OpenTelemetry

Recommended File Structure for Debugging Support

```
container-runtime/
  cmd/
    runtime/main.go           ← main runtime binary
    debug/
      inspect.go             ← debug inspection commands
      health-check.go        ← automated health checking
  internal/
    debug/
      namespace.go           ← namespace debugging utilities
      cgroup.go               ← cgroup inspection helpers
      network.go              ← network debugging tools
    logging/
      structured.go          ← structured logging setup
  scripts/
    debug-container.sh       ← shell script for manual debugging
    cleanup-orphaned.sh     ← cleanup script for orphaned resources
```

Debug Inspection Infrastructure

GO

```
// DebugInspector provides comprehensive container runtime debugging capabilities.

// It coordinates inspection across namespaces, cgroups, filesystem, and network
components.

type DebugInspector struct {

    runtime *RuntimeManager

    logger *StructuredLogger


    // Component-specific inspectors

    namespaceInspector *NamespaceInspector

    cgroupInspector    *CgroupInspector

    networkInspector   *NetworkInspector

    filesystemInspector *FilesystemInspector

}

// InspectContainer performs comprehensive debugging inspection of a container.

// Returns detailed state information for all container components.

func (d *DebugInspector) InspectContainer(containerID string) (*ContainerDebugInfo, error)
{
    // TODO 1: Load container state from StateManager

    // TODO 2: Inspect namespace state using NamespaceInspector

    // TODO 3: Collect cgroup statistics and configuration

    // TODO 4: Verify network configuration and connectivity

    // TODO 5: Check filesystem mount status and overlay state

    // TODO 6: Aggregate all inspection results into ContainerDebugInfo

    // TODO 7: Identify any configuration inconsistencies or errors

}

// ContainerDebugInfo aggregates debugging information from all container components.

type ContainerDebugInfo struct {
```

```
ContainerID    string           `json:"container_id"`

State          ContainerState   `json:"state"`

Namespace      NamespaceDebugInfo `json:"namespace"`

Cgroup         CgroupDebugInfo  `json:"cgroup"`

Network        NetworkDebugInfo `json:"network"`

Filesystem     FilesystemDebugInfo `json:"filesystem"`

Issues         []DebugIssue     `json:"issues"`

Recommendations []string        `json:"recommendations"`

}
```

Namespace Debugging Implementation

```
// NamespaceInspector provides tools for inspecting namespace state and configuration.      GO

type NamespaceInspector struct {

    procFS string // typically "/proc"
}

// InspectNamespaceState examines namespace configuration for a container process.

// Returns detailed information about namespace types, IDs, and isolation status.

func (ni *NamespaceInspector) InspectNamespaceState(pid int) (*NamespaceDebugInfo, error) {

    // TODO 1: Read namespace information from /proc/[pid]/ns/
    // TODO 2: Compare container namespace IDs with host namespace IDs
    // TODO 3: Verify expected namespace types are created and isolated
    // TODO 4: Check mount table from /proc/[pid]/mountinfo
    // TODO 5: Verify /proc filesystem is properly mounted in container
    // TODO 6: Inspect user namespace UID/GID mappings
    // TODO 7: Check UTS namespace hostname configuration
}

// VerifyProcessIsolation checks if container process sees only expected processes.

// Tests PID namespace isolation by comparing visible processes.

func (ni *NamespaceInspector) VerifyProcessIsolation(containerPID int)
(*ProcessIsolationResult, error) {

    // TODO 1: Execute 'ps aux' inside container namespace using nsenter
    // TODO 2: Parse process list and verify container process has PID 1
    // TODO 3: Ensure no host processes are visible in container
    // TODO 4: Check that only container and child processes are visible
    // TODO 5: Return isolation status with any violations detected
}
```

Cgroup Debugging Implementation

```
// CgroupInspector provides tools for inspecting cgroup configuration and resource usage. GO
type CgroupInspector struct {

    cgroupRoot string // typically "/sys/fs/cgroup"
}

// InspectCgroupConfig examines cgroup setup and resource limits for a container.

// Returns comprehensive information about controllers, limits, and current usage.

func (ci *CgroupInspector) InspectCgroupConfig(cgroupPath string) (*CgroupDebugInfo, error)
{
    // TODO 1: Verify cgroup directory exists at expected path

    // TODO 2: Check enabled controllers in cgroup.controllers

    // TODO 3: Read memory limits from memory.max and memory.swap.max

    // TODO 4: Read CPU configuration from cpu.max and cpu.weight

    // TODO 5: Collect current usage from memory.current and cpu.stat

    // TODO 6: Check for OOM events in memory.events

    // TODO 7: Verify process assignment in cgroup.procs

}

// MonitorResourcePressure tracks resource pressure indicators for debugging.

// Returns pressure information that can indicate resource bottlenecks.

func (ci *CgroupInspector) MonitorResourcePressure(cgroupPath string) (*PressureInfo, error) {
    // TODO 1: Read memory pressure from memory.pressure file

    // TODO 2: Read CPU pressure from cpu.pressure file

    // TODO 3: Read IO pressure from io.pressure file

    // TODO 4: Parse pressure stall information (some, full averages)

    // TODO 5: Calculate pressure trends over monitoring period

    // TODO 6: Return structured pressure information with analysis
}
```

}

Network Debugging Implementation

GO

```
// NetworkInspector provides tools for debugging container networking configuration.

type NetworkInspector struct {
    netlink NetlinkInterface
    iptables IPTablesInterface
}

// InspectNetworkConfig examines network setup for container connectivity debugging.

// Returns detailed information about interfaces, routing, and NAT configuration.

func (ni *NetworkInspector) InspectNetworkConfig(containerID string, pid int)
(*NetworkDebugInfo, error) {

    // TODO 1: List network interfaces in host and container namespaces

    // TODO 2: Verify veth pair creation and proper link state

    // TODO 3: Check bridge membership for container interface

    // TODO 4: Inspect IP address assignment and routing configuration

    // TODO 5: Verify iptables NAT rules for port forwarding

    // TODO 6: Test connectivity between container and bridge gateway

    // TODO 7: Check DNS configuration in container namespace

}

// TestConnectivity performs connectivity tests for network debugging.

// Tests various network paths to identify where communication fails.

func (ni *NetworkInspector) TestConnectivity(containerPID int, targetHost string,
targetPort int) (*ConnectivityTestResult, error) {

    // TODO 1: Test loopback connectivity within container

    // TODO 2: Test connectivity to bridge gateway from container

    // TODO 3: Test connectivity to other containers on same bridge

    // TODO 4: Test connectivity to external hosts from container

    // TODO 5: Test port forwarding from host to container service

    // TODO 6: Return detailed results showing which tests passed/failed
```

```
}
```

Automated Health Checking

```
// HealthChecker provides automated validation of container runtime state. GO

// Performs systematic checks to identify common configuration problems.

type HealthChecker struct {

    inspector *DebugInspector

    logger    *StructuredLogger

}

// ValidateContainerHealth performs comprehensive health checking for a container.

// Returns structured results indicating any detected issues or inconsistencies.

func (hc *HealthChecker) ValidateContainerHealth(containerID string) (*HealthCheckResult, error) {

    // TODO 1: Verify container state is consistent across all components

    // TODO 2: Check that all expected namespaces are created and isolated

    // TODO 3: Validate cgroup configuration matches container specification

    // TODO 4: Verify network configuration and connectivity

    // TODO 5: Check filesystem mount status and overlay configuration

    // TODO 6: Test basic container operations (exec, signal delivery)

    // TODO 7: Return comprehensive health status with recommendations

}
```

Milestone Checkpoints for Debugging Implementation

After implementing debugging capabilities for each milestone:

Milestone 1 Checkpoint - Namespace Debugging:

```
# Test namespace inspection                                                 BASH

go run ./cmd/debug inspect --container test-container --component namespaces

# Expected output shows:

# - PID namespace: isolated (container PID 1 ≠ host PID)

# - Mount namespace: isolated (/proc shows only container processes)

# - Network namespace: isolated (only container interfaces visible)

# - UTS namespace: isolated (container hostname differs from host)

# Test process isolation verification

sudo nsenter -t $(pgrep test-container) -p ps aux

# Should show only container processes with container process as PID 1
```

Milestone 2 Checkpoint - Cgroup Debugging:

```
# Test cgroup inspection                                                 BASH

go run ./cmd/debug inspect --container test-container --component cgroups

# Expected output shows:

# - Memory limit: configured and enforced

# - CPU quota: configured with expected values

# - Process assignment: container PID in correct cgroup

# - Resource usage: current consumption within limits

# Test resource monitoring

go run ./cmd/debug monitor --container test-container --duration 30s

# Should show resource usage trends and pressure indicators
```

Milestone 3 Checkpoint - Filesystem Debugging:

```
# Test overlay filesystem inspection  
go run ./cmd/debug inspect --container test-container --component filesystem  
  
# Expected output shows:  
  
# - Overlay mount: correctly configured with upper/lower/work directories  
# - Layer stacking: proper order and accessibility  
# - Copy-on-write: file modifications isolated to upper layer  
  
# Verify overlay mount  
findmnt | grep test-container  
  
# Should show overlayfs mount with correct layer directories
```

BASH

Milestone 4 Checkpoint - Network Debugging:

```
# Test network configuration inspection  
go run ./cmd/debug inspect --container test-container --component network  
  
# Expected output shows:  
  
# - Veth pair: created and properly linked  
# - Bridge membership: container attached to bridge  
# - IP assignment: unique IP from configured pool  
# - Port forwarding: iptables rules for mapped ports  
  
# Test connectivity  
go run ./cmd/debug test-connectivity --container test-container --target google.com:80  
  
# Should show successful connectivity test results
```

Debugging Script Examples

BASH

```
#!/bin/bash

# debug-container.sh - Manual debugging script for container issues

CONTAINER_ID="$1"

if [ -z "$CONTAINER_ID" ]; then

    echo "Usage: $0 <container-id>"

    exit 1

fi

echo "==== Container Debug Information ===="

echo "Container ID: $CONTAINER_ID"

# Get container PID

PID=$(pgrep -f "$CONTAINER_ID")

if [ -z "$PID" ]; then

    echo "ERROR: Container process not found"

    exit 1

fi

echo "Container PID: $PID"

# Check namespace isolation

echo -e "\n==== Namespace Information ===="

ls -la "/proc/$PID/ns/"

# Check cgroup assignment

echo -e "\n==== Cgroup Information ===="

cat "/proc/$PID/cgroup"

# Check network configuration
```

```

echo -e "\n==== Network Configuration ==="

nsenter -t "$PID" -n ip addr show

nsenter -t "$PID" -n ip route show

# Check resource usage

echo -e "\n==== Resource Usage ==="

CGROUP_PATH=$(grep memory "/proc/$PID/cgroup" | cut -d: -f3)

if [ -n "$CGROUP_PATH" ]; then

    cat "/sys/fs/cgroup$CGROUP_PATH/memory.current"

    cat "/sys/fs/cgroup$CGROUP_PATH/memory.max"

fi

```

This debugging implementation provides comprehensive tools for identifying and resolving container runtime issues across all components. The structured approach ensures systematic problem diagnosis while the automated health checking helps prevent issues from reaching production.

Future Extensions

Milestone(s): Beyond milestones 1-4 - This section outlines advanced features that could be added to enhance our container runtime with production-grade security, advanced networking capabilities, and enterprise features.

Mental Model: Construction Site Evolution

Think of our container runtime like a construction site that starts with basic safety measures and essential tools. We began with fundamental isolation (hard hats and safety barriers), basic resource allocation (power and water connections), simple shelters (basic structures), and basic communication (walkie-talkies). Now we're ready to add advanced security systems (keycard access, surveillance cameras), sophisticated infrastructure (fiber optic networks, smart building systems), and professional-grade management tools (project management software, quality control systems).

Just as a construction site evolves from meeting basic safety requirements to implementing enterprise-grade security and management systems, our container runtime can grow from providing fundamental isolation to offering production-ready security policies, advanced networking topologies, and sophisticated operational

capabilities. Each enhancement builds upon the solid foundation we've established while adding specialized capabilities for different use cases.

The key insight is that these extensions don't replace our core components—they enhance and extend them. Our namespace isolation becomes the foundation for advanced security policies. Our basic networking becomes the substrate for complex multi-host topologies. Our simple resource management becomes the basis for sophisticated orchestration integration.

Current Foundation Assessment

Before exploring future extensions, it's important to understand what our current implementation provides and where natural extension points exist. Our container runtime has established four core capabilities that serve as extension foundations.

Our namespace isolation component provides the fundamental security boundary that advanced security policies can build upon. The `NamespaceHandle` abstraction already supports user namespace UID/GID mapping, which becomes the foundation for capability management and privilege escalation prevention. The mount namespace isolation provides the filesystem access control foundation that security policies like AppArmor and SELinux can leverage.

The resource control component offers basic memory, CPU, and device limits through cgroups v2, but production environments often need more sophisticated resource policies. Our current `ResourceLimits` structure and `Controller` interfaces provide natural extension points for advanced resource management features like priority classes, quality of service guarantees, and dynamic resource allocation.

Our network management component implements basic bridge networking with port forwarding, providing the foundation for advanced networking features. The `Manager` component's abstraction of network operations and the `ContainerNetwork` state tracking create natural integration points for Container Network Interface (CNI) plugins and multi-host networking solutions.

The container lifecycle management in our `RuntimeManager` provides orchestration integration points. The `ContainerSpec` and `ContainerState` data model can be extended to support additional metadata, labels, and configuration options that orchestration systems require.

Design Insight: Extension points should be designed into the original architecture, not bolted on afterwards. Our component-based architecture with clear interfaces makes it easier to add new capabilities without disrupting existing functionality.

Security Enhancements

Production container runtimes require multiple layers of security beyond basic namespace isolation. Advanced security features provide defense in depth, ensuring that even if one security boundary is compromised, additional protections remain in place.

Mental Model: Multi-Layer Security Castle

Think of container security like a medieval castle with multiple defensive layers. Our current namespace isolation is like the outer wall—it keeps most threats out, but sophisticated attackers might find ways over or through it. Advanced security features add additional defensive layers: the moat (seccomp filters blocking dangerous system calls), the gatehouse (capability management controlling what actions are allowed), and the inner keep (AppArmor/SELinux policies providing fine-grained access control).

Each security layer operates independently but works together to provide comprehensive protection. If an attacker bypasses the namespace isolation (outer wall), they still face seccomp filters (moat) that prevent dangerous system calls. If they somehow execute restricted system calls, capability restrictions (gatehouse) limit what damage they can cause. Even if they gain elevated privileges, mandatory access control policies (inner keep) restrict what resources they can access.

Seccomp Filter Integration

Secure Computing Mode (seccomp) filters provide system call level security by restricting which kernel interfaces containerized processes can access. This addresses a fundamental limitation of namespace isolation—while namespaces control what resources a process can see, they don't restrict which system calls the process can make.

Our current `NamespaceConfig` structure provides a natural extension point for seccomp integration. We can add seccomp profile configuration alongside existing namespace settings:

Extension Field	Type	Purpose
<code>SeccompProfile</code>	<code>string</code>	Path to seccomp BPF profile or predefined profile name
<code>SeccompMode</code>	<code>SeccompMode</code>	Filter mode: disabled, strict, filter, or custom
<code>AllowedSyscalls</code>	<code>[]string</code>	Explicit whitelist of permitted system calls
<code>BlockedSyscalls</code>	<code>[]string</code>	Explicit blacklist of dangerous system calls
<code>SeccompNotification</code>	<code>bool</code>	Enable seccomp user notification for advanced filtering

The implementation would extend our namespace creation process to apply seccomp filters before executing the container process. The filter installation must happen after namespace creation but before the container process starts, ensuring the restrictions apply to the containerized process and its children.

A sophisticated seccomp implementation would support multiple profile types. Default profiles provide reasonable security for common use cases—web servers, databases, batch jobs—while custom profiles allow fine-tuned control for specialized applications. Profile composition enables combining multiple partial profiles, such as a base container profile plus application-specific additions.

Decision: Seccomp Profile Management Strategy

- **Context:** Seccomp filters require careful balance between security and application compatibility
- **Options Considered:**
 - Hard-coded restrictive filters for maximum security
 - Fully permissive mode for maximum compatibility
 - Configurable profiles with sensible defaults
- **Decision:** Configurable profiles with curated defaults for common application types
- **Rationale:** Provides security benefits while maintaining usability; defaults protect against common attack vectors while custom profiles enable specialized applications
- **Consequences:** Requires maintaining profile library and documentation; adds configuration complexity but provides flexible security model

Capability Management System

Linux capabilities provide fine-grained privilege control by breaking down root privileges into discrete capabilities that can be individually granted or restricted. Our container runtime can integrate capability management to implement the principle of least privilege.

The capability system would extend our `NamespaceConfig` with capability configuration:

Capability Field	Type	Purpose
<code>AllowedCapabilities</code>	<code>[]string</code>	Capabilities container processes can use
<code>DroppedCapabilities</code>	<code>[]string</code>	Capabilities explicitly removed from container
<code>AmbientCapabilities</code>	<code>[]string</code>	Capabilities inherited by child processes
<code>BoundingSet</code>	<code>[]string</code>	Maximum capabilities container can ever acquire
<code>NoNewPrivileges</code>	<code>bool</code>	Prevent privilege escalation via execve

The implementation would manage capability sets during process creation, using the `cap_set_proc()` interface to configure process capabilities before executing the container command. This requires careful handling of capability inheritance and the interaction between user namespaces and capabilities.

Advanced capability management includes capability-aware process execution, ensuring that capabilities are properly dropped when switching to non-root users within the container. The system should also support capability audit logging, tracking when processes attempt to use specific capabilities for security monitoring.

AppArmor and SELinux Integration

Mandatory Access Control (MAC) systems like AppArmor and SELinux provide policy-based security that restricts resource access regardless of traditional Unix permissions. Integration with these systems adds

another security layer beyond namespace isolation and capability restrictions.

MAC integration extends container configuration with policy specification:

MAC Field	Type	Purpose
AppArmorProfile	string	AppArmor profile name or path for container processes
SELinuxLabel	string	SELinux security context for container processes
SELinuxType	string	SELinux type enforcement domain
SELinuxUser	string	SELinux user identity
SELinuxRole	string	SELinux role for process context
MACEnforcement	MACMode	MAC enforcement level: disabled, permissive, enforcing

The implementation requires platform-specific integration with MAC systems. On AppArmor systems, this involves loading profiles and applying them to container processes using `aa_change_profile()`. On SELinux systems, it requires setting security contexts using `setexeccon()` before process execution.

MAC policy management becomes a significant operational concern. The runtime should support policy loading, validation, and fallback mechanisms when specified policies are unavailable. Integration with policy development tools helps administrators create and maintain custom policies for containerized applications.

Architecture Decision: MAC System Integration Strategy

- **Context:** Different Linux distributions use different MAC systems (AppArmor on Ubuntu/SUSE, SELinux on RHEL/CentOS)
- **Options Considered:**
 - Support only one MAC system to reduce complexity
 - Runtime detection and automatic MAC system selection
 - Plugin-based architecture supporting multiple MAC systems
- **Decision:** Plugin-based MAC integration with runtime detection
- **Rationale:** Provides maximum compatibility across distributions; plugin architecture allows adding new MAC systems; runtime detection reduces configuration burden
- **Consequences:** Adds architectural complexity but enables broad deployment; requires maintaining multiple MAC integrations

Security Policy Composition

Advanced security requires combining multiple security mechanisms into coherent policies. A security policy composition system would allow administrators to define security profiles that automatically configure seccomp filters, capabilities, and MAC policies together.

Security profiles provide named collections of security settings:

Profile Component	Configuration Scope	Integration Point
Seccomp Rules	System call restrictions	Applied during process creation
Capability Sets	Privilege limitations	Configured via capability management
MAC Policies	Resource access control	Integrated with AppArmor/SELinux
Namespace Options	Isolation boundaries	Extended namespace configuration
Resource Quotas	Security-related limits	Enhanced resource control

Profile composition enables security policy inheritance and customization. Base profiles provide security foundations for broad categories (web applications, databases, batch processing), while override mechanisms allow application-specific customizations without duplicating common security settings.

The implementation would extend our `ContainerSpec` to reference security profiles:

Security Extension	Type	Purpose
<code>SecurityProfile</code>	<code>string</code>	Named security profile to apply
<code>SecurityOverrides</code>	<code>SecurityConfig</code>	Specific overrides to profile defaults
<code>SecurityAudit</code>	<code>bool</code>	Enable security event logging and monitoring

Advanced Networking Features

Production container deployments require sophisticated networking capabilities beyond basic bridge networking. Advanced networking features enable complex topologies, multi-host communication, and integration with orchestration systems.

Mental Model: Metropolitan Transportation Network

Think of advanced container networking like evolving from a simple company shuttle system to a metropolitan transportation network. Our current bridge networking is like shuttle buses connecting different buildings on a campus—it works well for local communication but doesn't scale to citywide transportation needs.

Advanced networking features are like adding subway systems (overlay networks for multi-host communication), express highways (high-performance networking), traffic management systems (network policies and quality of service), and integrated payment systems (Container Network Interface compatibility for orchestration integration).

Each networking enhancement serves different transportation needs while integrating with the overall network infrastructure. Multi-host networking provides the backbone for distributed communication, network policies manage traffic flow and security, and CNI integration enables interoperability with orchestration systems.

Multi-Host Networking Implementation

Multi-host networking enables containers running on different physical hosts to communicate as if they were on the same local network. This requires overlay networking technologies that encapsulate container traffic for transmission across the underlying network infrastructure.

The multi-host networking extension would add overlay network management to our `NetworkConfig` :

Multi-Host Field	Type	Purpose
<code>OverlayDriver</code>	<code>string</code>	Overlay network implementation: VXLAN, Geneve, IPsec
<code>ClusterSubnet</code>	<code>string</code>	CIDR block for cross-host container networking
<code>OverlayInterface</code>	<code>string</code>	Host interface for overlay network traffic
<code>EncryptionEnabled</code>	<code>bool</code>	Enable overlay traffic encryption
<code>MTUSize</code>	<code>int</code>	Maximum transmission unit for overlay packets
<code>MulticastEnabled</code>	<code>bool</code>	Enable multicast for service discovery

VXLAN (Virtual Extensible LAN) provides the most common overlay implementation. It encapsulates layer-2 Ethernet frames within UDP packets, allowing container traffic to traverse layer-3 networks. The implementation requires creating VXLAN interfaces, configuring bridge forwarding tables, and managing overlay network membership.

Advanced overlay networking includes distributed route management, where each host maintains routing information for containers on other hosts. This can be implemented through static configuration, dynamic protocols like BGP, or integration with service discovery systems like etcd or Consul.

Network address translation becomes more complex in multi-host environments. East-west traffic (container-to-container across hosts) uses overlay routing, while north-south traffic (external-to-container) requires careful NAT rule coordination to avoid conflicts between hosts.

Container Network Interface (CNI) Integration

Container Network Interface (CNI) provides a standardized plugin architecture for container networking, enabling integration with Kubernetes and other orchestration systems. CNI integration transforms our container runtime from a standalone system into an orchestration-compatible component.

CNI integration extends our network management with plugin support:

CNI Extension	Type	Purpose
CNICConfigPath	string	Directory containing CNI plugin configurations
CNIPluginPath	string	Directory containing CNI plugin binaries
CNINetworkName	string	Default network configuration to use
CNICacheDir	string	Directory for CNI plugin state and caching
CNIArgs	map[string]string	Additional arguments passed to CNI plugins

The implementation replaces our direct network management with CNI plugin execution. Instead of creating veth pairs and bridge configurations directly, the runtime executes CNI plugins that handle network setup according to standardized interfaces.

CNI plugin lifecycle follows the ADD/DEL pattern. During container creation, the runtime calls the CNI plugin ADD command with container information and network configuration. The plugin creates network interfaces, assigns IP addresses, and configures routing according to its implementation. During container removal, the DEL command cleans up network resources.

Advanced CNI integration supports plugin chaining, where multiple CNI plugins collaborate to provide complete networking functionality. For example, a bridge plugin creates basic connectivity while a firewall plugin adds security rules and a monitoring plugin adds traffic statistics.

Decision: CNI Integration Strategy

- **Context:** CNI provides standardized networking but adds complexity compared to direct implementation
- **Options Considered:**
 - Replace all networking with CNI plugins
 - Support CNI as optional alternative to built-in networking
 - Implement CNI-compatible plugins wrapping our existing networking
- **Decision:** Support CNI as configurable alternative with built-in networking as fallback
- **Rationale:** Provides orchestration compatibility while maintaining simplicity for standalone use; CNI plugins handle complex networking scenarios our built-in implementation can't support
- **Consequences:** Increases testing complexity but enables Kubernetes integration; requires maintaining both networking paths

Network Policy Enforcement

Network policies provide declarative security and traffic management for container networking. They define which containers can communicate with each other and what external resources containers can access.

Network policy implementation extends our networking with policy specification:

Policy Component	Type	Purpose
NetworkPolicies	[]NetworkPolicy	List of policies applied to container
DefaultPolicy	PolicyAction	Default action for non-matching traffic
PolicyEnforcement	EnforcementMode	Policy enforcement level

Network policy enforcement requires integration with traffic filtering mechanisms. On Linux systems, this typically means generating iptables rules that implement policy decisions. Advanced implementations use eBPF programs for high-performance packet filtering.

Policy evaluation becomes complex with multiple policies applying to the same container. The implementation needs conflict resolution mechanisms, policy precedence rules, and efficient policy lookup algorithms for high-throughput environments.

Quality of Service and Traffic Shaping

Advanced networking includes Quality of Service (QoS) mechanisms that prioritize traffic and manage bandwidth allocation. This ensures critical container communications receive necessary network resources even under load.

QoS implementation adds traffic management to container networking:

QoS Component	Type	Purpose
BandwidthLimits	BandwidthConfig	Maximum bandwidth for container traffic
TrafficPriority	PriorityClass	Traffic prioritization level
QueueDiscipline	string	Linux traffic control queuing discipline
RateShaping	RateConfig	Traffic shaping parameters

Implementation uses Linux traffic control (tc) subsystem to configure queuing disciplines, traffic classes, and rate limiting. This requires understanding of network queuing theory and careful configuration to avoid unintended traffic delays.

Advanced QoS features include adaptive bandwidth allocation, where containers can burst above their guaranteed bandwidth when network capacity is available, and priority-based preemption, where high-priority traffic can temporarily reduce bandwidth available to lower-priority flows.

Image Management and Distribution

Production container runtimes require sophisticated image management capabilities beyond the basic layer handling we implemented. Image management encompasses image distribution, storage optimization,

security scanning, and metadata management.

Mental Model: Digital Library System

Think of advanced image management like evolving from a personal book collection to managing a digital library system. Our current layer management is like having books on shelves—you can find and use them, but there's no catalog system, no way to share with other libraries, and no way to verify book authenticity.

Advanced image management adds library infrastructure: catalog systems (image registries with metadata), interlibrary loans (image distribution and sharing), authentication systems (image signing and verification), and storage optimization (deduplication and compression).

Container Image Registry Integration

Image registry integration enables pulling container images from remote repositories, supporting image distribution workflows common in production environments. This extends our runtime beyond locally available layers to support dynamic image retrieval.

Registry integration extends our `ContainerSpec` with remote image support:

Registry Field	Type	Purpose
<code>ImageRegistry</code>	<code>string</code>	Registry URL for image retrieval
<code>ImageCredentials</code>	<code>RegistryCredentials</code>	Authentication for private registries
<code>ImageTag</code>	<code>string</code>	Specific image version or tag
<code>ImageDigest</code>	<code>string</code>	Content-addressable image identifier
<code>PullPolicy</code>	<code>PullPolicy</code>	When to pull images: Always, IfNotPresent, Never

Implementation requires supporting Docker Registry HTTP API V2 for image manifest retrieval and layer downloading. This includes handling authentication, layer deduplication, and resumable downloads for large images.

Advanced registry integration includes image mirroring and caching, where frequently used images are cached locally to reduce registry load and improve startup times. Multi-registry support enables fallback between primary and mirror registries for high availability.

Image Security and Verification

Image security addresses supply chain security by verifying image authenticity and scanning for vulnerabilities. This becomes critical in production environments where compromised images pose significant security risks.

Image security extends our image handling with verification:

Security Component	Type	Purpose
ImageSigning	SigningConfig	Image signature verification settings
VulnerabilityScanning	ScanConfig	Security vulnerability detection
PolicyEnforcement	SecurityPolicy	Image acceptance policies
TrustedRegistries	[]string	Whitelist of trusted image sources

Implementation integrates with image signing systems like Notary or cosign for cryptographic verification of image authenticity. Vulnerability scanning requires integration with security databases and scanning tools to identify known vulnerabilities in image contents.

Policy enforcement enables organizations to define acceptance criteria for container images, automatically rejecting images that don't meet security requirements. This includes vulnerability thresholds, signature requirements, and source registry restrictions.

Orchestration Integration

Production container runtimes increasingly operate as components within larger orchestration systems. Orchestration integration enables our runtime to participate in cluster management, service discovery, and automated deployment workflows.

Mental Model: Orchestra Conductor System

Think of orchestration integration like enabling a talented musician to play as part of a symphony orchestra. Our container runtime is like a skilled instrumentalist who can perform solo pieces beautifully, but orchestration integration adds the ability to follow a conductor, coordinate with other musicians, and contribute to complex symphonic performances.

The runtime learns to respond to orchestration signals (start this container when that service is ready), participate in ensemble coordination (register service endpoints for load balancing), and adapt to the conductor's tempo (scale up during high demand, scale down during quiet periods).

Kubernetes Container Runtime Interface (CRI)

Container Runtime Interface provides the standard interface between Kubernetes and container runtimes. CRI integration transforms our runtime into a Kubernetes-compatible component that can manage pods and containers under kubelet orchestration.

CRI implementation requires implementing the RuntimeService and ImageService gRPC interfaces:

CRI Service	Key Methods	Purpose
RuntimeService	RunPodSandbox , CreateContainer , StartContainer	Pod and container lifecycle
ImageService	PullImage , ListImages , RemoveImage	Image management operations
StreamingService	Exec , Attach , PortForward	Interactive container operations

Implementation challenges include adapting our single-container model to Kubernetes' pod abstraction, where multiple containers share network and storage namespaces. This requires extending our namespace management to support shared namespaces between related containers.

CRI integration also requires implementing Kubernetes-specific features like pod security contexts, resource quotas, and liveness/readiness probes. These extend our basic container capabilities with orchestration-aware functionality.

Service Mesh Integration

Service mesh architectures require container runtimes to support sidecar injection, traffic interception, and service identity management. Integration enables our runtime to participate in sophisticated microservices architectures.

Service mesh support extends container networking with mesh capabilities:

Mesh Component	Type	Purpose
SidecarInjection	bool	Automatic sidecar proxy injection
TrafficInterception	InterceptConfig	Traffic redirection configuration
ServiceIdentity	IdentityConfig	Service authentication and authorization
MeshConfiguration	MeshConfig	Service mesh integration settings

Implementation requires coordinating with service mesh control planes for configuration distribution and telemetry collection. This includes supporting init containers for traffic redirection setup and managing certificate distribution for mutual TLS authentication.

Storage and Volume Management

Advanced storage features extend our basic filesystem layering with persistent volumes, dynamic provisioning, and storage optimization capabilities.

Persistent Volume Integration

Persistent volumes enable containers to access storage that survives container lifecycle events. This requires extending our filesystem management beyond container-scoped overlay filesystems.

Volume integration extends `ContainerSpec` with storage specification:

Volume Field	Type	Purpose
<code>Volumes</code>	<code>[]VolumeMount</code>	List of volumes to mount in container
<code>VolumeDrivers</code>	<code>map[string]VolumeDriver</code>	Storage driver implementations
<code>StorageClasses</code>	<code>[]StorageClass</code>	Available storage types and policies

Implementation requires supporting multiple volume types: host path mounts for simple local storage, network file systems for shared storage, and cloud provider storage APIs for dynamic provisioning.

Storage Optimization

Storage optimization includes features like image layer deduplication, compression, and content-addressable storage that reduce storage overhead and improve performance.

Advanced storage features include snapshot support for volume backup and restoration, encryption for sensitive data protection, and storage quotas for multi-tenant environments.

Common Pitfalls in Advanced Extensions

⚠ Pitfall: Feature Creep Without Architecture Planning Adding advanced features without considering their architectural impact leads to tightly coupled, difficult-to-maintain systems. Each extension should have clear interfaces and minimal dependencies on other components. Plan extension points during initial design rather than retrofitting them later.

⚠ Pitfall: Security Feature Interference Multiple security mechanisms can interfere with each other if not carefully coordinated. Seccomp filters might block system calls that AppArmor policies expect to audit. User namespace UID mapping might conflict with capability management. Design security feature integration holistically rather than adding each mechanism independently.

⚠ Pitfall: CNI Plugin Compatibility Assumptions Assuming all CNI plugins behave consistently leads to runtime errors in diverse environments. Different CNI plugins have different requirements for network namespace setup, interface naming, and cleanup behavior. Test with multiple CNI plugins and handle plugin-specific quirks gracefully.

⚠ Pitfall: Performance Impact of Advanced Features Advanced features often add performance overhead that may not be acceptable in all environments. Image scanning delays container startup. Network policies add packet processing overhead. QoS traffic shaping adds latency. Make advanced features configurable and provide performance monitoring to quantify their impact.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Seccomp Integration	Static JSON profiles	eBPF-based dynamic filtering
MAC System Support	Single system (AppArmor or SELinux)	Runtime detection with plugin architecture
CNI Implementation	Execute CNI binaries directly	CNI library integration with caching
Image Registry	HTTP client with basic auth	Full OCI distribution spec support
Orchestration	Simple webhook API	Full CRI gRPC implementation

Recommended File Structure

```
project-root/
  internal/extensions/
    security/
      seccomp/
        profiles.go      ← seccomp profile management
        filters.go       ← BPF filter installation
      capabilities/
        manager.go      ← capability set management
      mac/
        apparmor.go     ← AppArmor integration
        selinux.go      ← SELinux integration
        interface.go    ← MAC system abstraction
    networking/
      cni/
        plugin_manager.go ← CNI plugin execution
        config_parser.go  ← CNI configuration handling
      overlay/
        vxlan.go         ← VXLAN overlay implementation
        routing.go       ← Overlay route management
      policy/
        enforcer.go     ← Network policy enforcement
        iptables.go      ← iptables rule management
    storage/
      registry/
        client.go        ← Registry API client
        auth.go          ← Registry authentication
      volumes/
        manager.go       ← Volume lifecycle management
      drivers/
        hostpath.go      ← Host path volume driver
        nfs.go           ← NFS volume driver
  orchestration/
    cri/
      runtime_service.go ← CRI RuntimeService implementation
      image_service.go   ← CRI ImageService implementation
    mesh/
      sidecar.go        ← Service mesh sidecar management
```

Security Extension Starter Code

```
// Package security provides advanced security features for container runtime      GO
package security

import (
    "context"
    "fmt"
    "syscall"
    "unsafe"
)

// SecurityManager coordinates multiple security mechanisms

type SecurityManager struct {
    seccompEnabled    bool
    capabilityEnabled bool
    macEnabled        bool
    profiles          map[string]*SecurityProfile
}

// SecurityProfile defines a complete security configuration

type SecurityProfile struct {
    Name            string
    SeccompProfile  string
    AllowedCapabilities []string
    DroppedCapabilities []string
    AppArmorProfile  string
    SELinuxLabel     string
}
```

```

// NewSecurityManager creates a security manager with system detection

func NewSecurityManager() (*SecurityManager, error) {
    return &SecurityManager{
        seccompEnabled:      detectSeccompSupport(),
        capabilityEnabled:  detectCapabilitySupport(),
        macEnabled:         detectMACSupport(),
        profiles:          make(map[string]*SecurityProfile),
    }, nil
}

// ApplySecurityProfile applies all security mechanisms from profile

func (sm *SecurityManager) ApplySecurityProfile(ctx context.Context, containerID string,
profile *SecurityProfile) error {
    // TODO 1: Validate security profile completeness and compatibility

    // TODO 2: Apply seccomp filters using prctl(PR_SET_SECCOMP) if enabled

    // TODO 3: Configure process capabilities using cap_set_proc if enabled

    // TODO 4: Set MAC security context (AppArmor or SELinux) if enabled

    // TODO 5: Enable no_new_privs to prevent privilege escalation

    // TODO 6: Record applied security settings for audit and debugging

    return fmt.Errorf("not implemented")
}

// SeccompFilterManager handles seccomp BPF filter installation

type SeccompFilterManager struct {
    profileDir string
}

// InstallSeccompFilter loads and installs BPF seccomp filter

func (sfm *SeccompFilterManager) InstallSeccompFilter(profilePath string) error {

```

```
// TODO 1: Load seccomp profile from JSON file

// TODO 2: Compile syscall names to architecture-specific numbers

// TODO 3: Generate BPF program from syscall allow/deny rules

// TODO 4: Install BPF filter using prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER)

// TODO 5: Verify filter installation and test with sample syscalls

return fmt.Errorf("not implemented")

}
```

CNI Integration Starter Code

```
// Package cni provides Container Network Interface integration          GO

package cni

import (
    "context"
    "encoding/json"
    "fmt"
    "os/exec"
    "path/filepath"
)

// CNIManager handles CNI plugin execution and state management

type CNIManager struct {
    configPath  string
    pluginPath  string
    cacheDir    string
    networks    map[string]*CNICConfig
}

// CNICConfig represents a CNI network configuration

type CNICConfig struct {
    CNIVersion string           `json:"cniVersion"`
    Name        string           `json:"name"`
    Type        string           `json:"type"`
    Bridge     map[string]interface{} `json:"bridge,omitempty"`
    IPAM       map[string]interface{} `json:"ipam,omitempty"`
    DNS        map[string]interface{} `json:"dns,omitempty"`
}
```

```
}

// CNIResult represents the result of CNI plugin execution

type CNIResult struct {

    CNIVersion string      `json:"cniVersion"`

    Interfaces []Interface `json:"interfaces"`

    IPs        []IPConfig   `json:"ips"`

    Routes     []Route      `json:"routes"`

    DNS        DNSConfig   `json:"dns"`

}

// NewCNIManager creates CNI manager with configuration discovery

func NewCNIManager(configPath, pluginPath, cacheDir string) (*CNIManager, error) {

    return &CNIManager{

        configPath: configPath,

        pluginPath: pluginPath,

        cacheDir:   cacheDir,

        networks:   make(map[string]*CNICConfig),

    }, nil

}

// SetupContainerNetwork executes CNI ADD command for container

func (cm *CNIManager) SetupContainerNetwork(ctx context.Context, containerID, networkName string, netnsPath string) (*CNIResult, error) {

    // TODO 1: Load CNI configuration from configPath directory

    // TODO 2: Find CNI plugin binary in pluginPath directory

    // TODO 3: Prepare CNI environment variables (CNI_COMMAND=ADD, CNI_CONTAINERID, etc.)

    // TODO 4: Execute CNI plugin with configuration JSON on stdin

    // TODO 5: Parse CNI result JSON from plugin stdout

}
```

```
// TODO 6: Cache network state in cacheDir for cleanup

    return nil, fmt.Errorf("not implemented")

}

// CleanupContainerNetwork executes CNI DEL command for container

func (cm *CNIManager) CleanupContainerNetwork(ctx context.Context, containerID, networkName
string) error {

    // TODO 1: Load cached network state from cacheDir

    // TODO 2: Prepare CNI environment variables (CNI_COMMAND=DEL)

    // TODO 3: Execute CNI plugin DEL command with original configuration

    // TODO 4: Remove cached network state from cacheDir

    // TODO 5: Handle plugin errors gracefully (plugin may be missing)

    return fmt.Errorf("not implemented")

}
```

Orchestration Integration Core Types

```
// Package orchestration provides integration with container orchestration systems      GO
package orchestration

import (
    "context"
    "time"
    "google.golang.org/grpc"
    pb "k8s.io/cri-api/pkg/apis/runtime/v1alpha2"
)

// CIRuntimeService implements Kubernetes Container Runtime Interface

type CIRuntimeService struct {
    pb.UnimplementedRuntimeServiceServer
    containerManager *ContainerManager
    podManager       *PodManager
    imageManager     *ImageManager
}

// PodSandboxConfig extends our container model for Kubernetes pods

type PodSandboxConfig struct {
    PodID          string
    Name           string
    Namespace      string
    SharedNetNS    bool
    SharedIPCNS   bool
    SharedPIDNS   bool
    Annotations    map[string]string
}
```

```

Labels      map[string]string

}

// RunPodSandbox creates the pod infrastructure container

func (cri *CIRuntimeService) RunPodSandbox(ctx context.Context, req
*pb.RunPodSandboxRequest) (*pb.RunPodSandboxResponse, error) {

    // TODO 1: Parse pod sandbox configuration from request

    // TODO 2: Create shared namespaces for pod (network, IPC, PID if configured)

    // TODO 3: Set up pod-level networking using CNI or built-in networking

    // TODO 4: Create pod sandbox container to hold namespaces

    // TODO 5: Apply pod-level security policies and resource limits

    // TODO 6: Register pod sandbox in pod manager state

    return nil, fmt.Errorf("not implemented")
}

// CreateContainer creates container within existing pod sandbox

func (cri *CIRuntimeService) CreateContainer(ctx context.Context, req
*pb.CreateContainerRequest) (*pb.CreateContainerResponse, error) {

    // TODO 1: Validate pod sandbox exists and is running

    // TODO 2: Join container to pod's shared namespaces

    // TODO 3: Apply container-specific resource limits and security policies

    // TODO 4: Set up container filesystem with image layers

    // TODO 5: Create container using our existing container creation logic

    // TODO 6: Register container in CRI state tracking

    return nil, fmt.Errorf("not implemented")
}

```

Milestone Checkpoints for Extensions

Security Enhancement Checkpoint:

- Test seccomp filter installation: `sudo strace -e trace=prctl ./container-runtime run --seccomp=restrictive test-container`
- Verify capability dropping: Container should fail to execute `ping` with network capabilities dropped
- Check MAC policy application: `ps -ez | grep container-process` should show correct SELinux context

CNI Integration Checkpoint:

- Install basic CNI plugins: `bridge` and `host-local` IPAM
- Test CNI network setup: `./container-runtime run --network=cni test-container`
- Verify connectivity: Container should get IP from CNI IPAM and reach other containers

Advanced Networking Checkpoint:

- Set up multi-host overlay: Deploy containers on different hosts, verify cross-host connectivity
- Test network policies: Create policy blocking container communication, verify enforcement
- Validate QoS: Apply bandwidth limits, verify traffic shaping with `iperf3`

Extension Debugging Tips

Symptom	Likely Cause	Diagnosis	Fix
Seccomp filter blocks required syscalls	Overly restrictive profile	<code>strace</code> container process, check for <code>EPERM</code> errors	Add required syscalls to profile whitelist
CNI plugin execution fails	Missing plugin binary or configuration	Check plugin exists in <code>CNI_PATH</code> , validate config JSON	Install plugin, fix configuration syntax
Container can't join pod network	Pod sandbox networking setup failed	Check pod sandbox status, inspect network namespaces	Debug CNI plugin execution, check bridge config
MAC policy denials	Incorrect security context	<code>ausearch -m AVC</code> for SELinux denials	Update security policy or container context
Multi-host networking broken	Overlay network misconfiguration	Check VXLAN interface status, validate routing	Verify overlay network setup, check firewall rules

These extensions transform our educational container runtime into a production-capable system while maintaining the clean architecture and clear component boundaries we established in the core implementation.

Glossary

Milestone(s): All milestones (1-4) - This glossary provides comprehensive definitions for container runtime terminology, Linux kernel concepts, and system administration terms used throughout all sections, helping readers understand the technical vocabulary essential for implementing process isolation, resource control, filesystem layering, and container networking.

This glossary serves as a comprehensive reference for all technical terms, concepts, and terminology used throughout the container runtime design document. Understanding these definitions is crucial for successfully implementing the four milestones of our container runtime project.

Mental Model: Technical Dictionary as Navigation Map

Think of this glossary as a navigation map for exploring a new technical territory. Just as a map helps you understand the landscape by defining landmarks, terrain features, and pathways, this glossary defines the technical landmarks (Linux kernel features), terrain features (system resources and isolation mechanisms), and pathways (APIs and interfaces) you'll encounter while building your container runtime. Each definition provides not just the meaning, but the context of how that concept fits into the larger ecosystem of containerization technology.

Container Runtime Core Concepts

The foundation of container technology rests on several key concepts that work together to create isolated execution environments.

Term	Category	Definition
container runtime	System Component	A system that manages the complete lifecycle of containers, including creation, execution, monitoring, and cleanup. Responsible for orchestrating Linux kernel features like namespaces, cgroups, and overlay filesystems to provide process isolation and resource control. Examples include Docker Engine, containerd, and our custom implementation.
container specification	Data Structure	A declarative blueprint defining how a container should be configured, including the image to run, resource limits, network settings, environment variables, and command to execute. Represented by the <code>ContainerSpec</code> structure with fields for image, command, resources, and networking configuration.
runtime state	Data Structure	The current operational status and resource tracking information for an active container. Includes process ID, namespace handles, cgroup paths, network configuration, and lifecycle timestamps. Maintained by the <code>StateManager</code> component for persistence across runtime restarts.
container lifecycle	State Machine	The progression of a container through distinct states: Created (resources allocated but process not started), Running (process executing), Stopped (process terminated but resources preserved), and Removed (all resources cleaned up). Each transition involves specific setup or cleanup operations.
process isolation	Isolation Mechanism	The fundamental container property where processes inside a container have a restricted view of system resources, seeing only what they're permitted to access. Achieved through Linux namespaces that create separate views of PIDs, filesystems, network interfaces, and other system resources.
resource limits	Control Mechanism	Constraints enforced by the Linux kernel to prevent containers from consuming unlimited system resources. Implemented using cgroups to limit memory, CPU time, I/O bandwidth, and process count. Essential for multi-tenant systems and preventing resource exhaustion.

Linux Kernel Features

Our container runtime builds upon several advanced Linux kernel features that provide the foundation for containerization.

Term	Category	Definition
namespace	Kernel Feature	A Linux kernel feature that provides process isolation by creating separate instances of global system resources. Each namespace type isolates a different category of resources (PIDs, filesystems, network interfaces, hostnames, users). Processes in different namespaces see different views of the same system.
cgroup	Kernel Feature	A Linux control group mechanism for organizing processes into hierarchical groups and applying resource limits and accounting. Cgroups v2 provides a unified hierarchy with controllers for memory, CPU, I/O, and device access. Essential for preventing containers from monopolizing system resources.
overlays	Filesystem Type	A union filesystem implementation in the Linux kernel that efficiently combines multiple directory trees into a single merged view. Supports copy-on-write semantics where changes to files are stored in an upper layer without modifying lower layers. Foundation for container image layering.
pivot_root	System Call	A Linux system call that atomically changes the root filesystem of a process and its children. More secure than <code>chroot</code> because it ensures the old root is unmounted. Used during container startup to switch from the host filesystem to the container's filesystem view.
netlink	Kernel Interface	A socket-based communication mechanism between the Linux kernel and userspace processes. Used for network configuration operations like creating network interfaces, configuring routes, and managing network namespaces. Provides programmatic access to networking functionality.

Namespace Types and Isolation

Each namespace type provides isolation for a specific category of system resources, creating the illusion of dedicated resources for each container.

Term	Category	Definition
mount namespace	Namespace Type	Isolates the filesystem mount points visible to processes, allowing each container to have its own filesystem hierarchy and mount configuration. Created with <code>CLONE_NEWNS</code> flag. Enables containers to have different root filesystems and mount different volumes without affecting the host.
PID namespace	Namespace Type	Isolates the process ID number space, making processes in the namespace see only other processes in the same namespace. The first process in a PID namespace appears as PID 1. Created with <code>CLONE_NEWPID</code> flag. Provides process isolation and enables clean process tree management.
network namespace	Namespace Type	Isolates network interfaces, routing tables, firewall rules, and network statistics. Each network namespace has its own loopback interface and can have its own network interfaces. Created with <code>CLONE_NEWWNET</code> flag. Enables containers to have isolated network stacks.
UTS namespace	Namespace Type	Isolates the system hostname and NIS domain name, allowing each container to have its own hostname without affecting the host system. Created with <code>CLONE_NEWUTS</code> flag. Enables containers to identify themselves with unique hostnames.
user namespace	Namespace Type	Isolates user and group ID number spaces and privileges, allowing processes to have different user IDs inside the namespace than outside. Enables unprivileged containers where root inside the container maps to a non-root user outside. Created with <code>CLONE_NEWUSER</code> flag.
IPC namespace	Namespace Type	Isolates System V IPC objects (message queues, semaphores, shared memory segments) and POSIX message queues. Prevents containers from interfering with each other's inter-process communication. Created with <code>CLONE_NEWIPC</code> flag.

Cgroup Resource Control

Cgroups provide hierarchical resource management and accounting, essential for fair resource allocation in multi-container environments.

Term	Category	Definition
cgroups v2	Cgroup Version	The modern unified hierarchy implementation of control groups, replacing the multiple separate hierarchies of cgroups v1. Provides a single tree structure with controllers that can be selectively enabled. Offers improved consistency and easier management for container runtimes.
memory controller	Cgroup Controller	A cgroup subsystem that manages memory allocation, usage tracking, and enforcement of memory limits. Can limit anonymous memory, page cache, and total memory usage. Provides memory pressure notifications and configurable out-of-memory behavior for fine-grained memory management.
CPU controller	Cgroup Controller	A cgroup subsystem that manages CPU time allocation through quotas and weights. Can limit CPU usage to specific percentages and implement fair scheduling policies. Provides both hard limits (quotas) and proportional sharing (weights) for flexible CPU resource management.
device controller	Cgroup Controller	A cgroup subsystem that controls access to device files, determining which devices processes can access and with what permissions (read, write, create). Essential for container security by preventing unauthorized access to host devices.
OOM killer	Kernel Mechanism	The Linux kernel's out-of-memory killer that terminates processes when memory is exhausted. In containerized environments, can be configured per-cgroup to kill only processes within the offending container rather than affecting the entire system. Critical for system stability under memory pressure.
memory pressure	Resource Condition	A condition where memory usage approaches configured limits, potentially leading to performance degradation or out-of-memory conditions. Cgroups provide pressure stall information (PSI) metrics to detect memory pressure before critical situations occur.
CPU throttling	Resource Control	The process of limiting CPU time allocation when a cgroup exceeds its configured CPU quota. Implemented by the kernel scheduler to enforce CPU limits while maintaining fairness. Can cause performance degradation if limits are set too restrictively.
resource monitoring	Monitoring Process	The continuous tracking of container resource consumption over time, including memory usage, CPU utilization, I/O operations, and process counts. Enables resource trend analysis, capacity planning, and automated scaling decisions based on historical usage patterns.

Term	Category	Definition
cgroup hierarchy	Organizational Structure	The tree structure organizing control groups, where child cgroups inherit resource limits from parents and can apply additional restrictions. Enables hierarchical resource management where system resources are allocated first to major subsystems, then subdivided for individual containers.

Filesystem and Storage Concepts

Container filesystems use sophisticated layering and copy-on-write mechanisms to efficiently share common data while allowing independent modifications.

Term	Category	Definition
copy-on-write	Filesystem Optimization	A storage optimization technique where multiple containers can share the same base files until one container modifies a file, at which point a private copy is created. Reduces storage usage and improves startup time by avoiding unnecessary file duplication.
layer stacking	Filesystem Architecture	The technique of combining multiple read-only filesystem layers with a single writable layer to create a complete filesystem view. Lower layers contain base system files and application dependencies, while the upper layer captures container-specific changes.
whiteout files	Filesystem Mechanism	Special marker files in overlayfs that indicate deleted files in lower layers. When a file is deleted in the upper layer, a whiteout file is created to mask the file's presence in lower layers. Essential for proper deletion semantics in layered filesystems.
upper directory	Filesystem Layer	The writable layer in an overlay filesystem where all container modifications are stored. Contains new files created by the container and modified versions of files from lower layers. Typically unique per container to ensure isolation.
lower directories	Filesystem Layers	The read-only layers in an overlay filesystem that contain base image content. Multiple lower directories are stacked to create the base filesystem view. Shared between containers using the same base image to minimize storage usage.
work directory	Filesystem Component	A scratch directory used internally by overlayfs for atomic operations during copy-on-write. Must be on the same filesystem as the upper directory and should be empty when mounting. Hidden from container processes but essential for overlay functionality.
merged view	Filesystem Abstraction	The combined filesystem view presented to container processes, showing the result of merging all lower directories with the upper directory. Files in upper layers hide files with the same path in lower layers, creating the illusion of a single unified filesystem.
layer cleanup	Maintenance Process	The process of removing unused filesystem layers and directories when containers are destroyed. Includes unmounting overlay filesystems, removing temporary directories, and updating reference counts for shared layers to prevent storage leaks.

Container Networking

Container networking creates isolated network environments while enabling communication between containers and external systems.

Term	Category	Definition
veth pair	Network Interface	A virtual ethernet pair consisting of two connected network interfaces that act like opposite ends of a network cable. Traffic sent to one interface is received by the other. Used to connect container network namespaces to host network namespaces or bridges.
bridge networking	Network Architecture	A layer-2 switching mechanism that connects multiple network interfaces, allowing containers to communicate with each other and external networks. Linux bridges forward traffic between connected interfaces based on MAC addresses, similar to physical network switches.
port forwarding	Network Translation	The process of redirecting network traffic from host ports to container ports using network address translation (NAT) rules. Enables external clients to access services running inside containers by mapping host IP:port combinations to container IP:port combinations.
port mapping	Configuration Specification	A configuration that defines how host ports should be forwarded to container ports, specified by the <code>PortMapping</code> structure with host port, container port, and protocol. Essential for making container services accessible from outside the container network.
DNAT	Network Translation	Destination Network Address Translation - a type of NAT that modifies the destination IP address and/or port of packets. Used in container port forwarding to redirect traffic from host interfaces to container interfaces.
container bridge	Network Infrastructure	A virtual network switch that connects container network interfaces, typically named <code>docker0</code> or similar. Provides layer-2 connectivity between containers and can be connected to host interfaces for external access. Central component of container networking architecture.
IP address pool	Address Management	A managed range of IP addresses allocated for container use, typically from private address space (e.g., 172.17.0.0/16). The container runtime assigns unique addresses from this pool to prevent IP conflicts and enable proper routing.
NAT rules	Network Configuration	Network address translation rules configured in the Linux netfilter framework (<code>iptables</code>) to modify packet headers for container networking. Includes DNAT rules for inbound traffic and SNAT rules for outbound traffic from containers.

Container Lifecycle and Operations

Container operations follow specific patterns and sequences to ensure reliable resource management and cleanup.

Term	Category	Definition
graceful termination	Shutdown Process	The process of allowing a container process to shut down cleanly by sending SIGTERM and waiting for the process to exit voluntarily before forcing termination. Gives applications time to save state, close connections, and perform cleanup operations.
forced removal	Cleanup Process	Aggressive container cleanup that bypasses graceful shutdown and immediately removes all container resources. Used when graceful termination fails or times out. May result in data loss or resource leaks if not implemented carefully.
resource handle	Abstraction	A data structure that encapsulates references to created resources (namespaces, cgroups, network interfaces) and provides methods for cleanup. Enables proper resource management and prevents resource leaks when containers are removed.
operation log	Persistence Mechanism	A persistent record of multi-step container operations that tracks which steps have completed successfully. Enables recovery and rollback if operations are interrupted by crashes or failures. Essential for maintaining consistent state across runtime restarts.
rollback mechanism	Error Recovery	The process of cleaning up successfully created resources when a container operation fails partway through. Ensures the system remains in a consistent state and prevents resource leaks when container creation or modification fails.
component coordination	Orchestration Pattern	The process of organizing multiple specialized components (namespace handler, cgroup controller, filesystem manager, network manager) to work together during container operations. Manages dependencies and ensures operations occur in the correct order.
dependency graph	Operational Structure	The ordering requirements between component operations, such as creating network namespaces before configuring network interfaces. Represents which operations must complete before others can begin, enabling parallel execution where possible.
atomic operations	Operation Property	Operations that either complete entirely or have no effect, preventing partial state that could lead to resource leaks or inconsistencies. Critical for container operations that involve multiple kernel resources requiring coordinated setup.
idempotent cleanup	Cleanup Property	Cleanup operations that can be safely repeated multiple times without causing errors or side effects. Essential for reliable resource cleanup in the presence of failures, crashes, or redundant cleanup attempts.

Error Handling and Recovery

Robust container runtimes must handle various failure modes and provide mechanisms for detection and recovery.

Term	Category	Definition
partial failure recovery	Error Recovery	The process of handling cases where container creation partially succeeds but some components fail, requiring cleanup of successfully created resources while leaving the system in a consistent state. Prevents resource leaks and orphaned kernel objects.
resource exhaustion scenarios	Failure Mode	Situations where system resources (PIDs, memory, network interfaces, file descriptors) approach or exceed their limits, potentially causing container operations to fail. Requires detection, graceful degradation, and recovery strategies.
graceful degradation	Resilience Pattern	The strategy of reducing functionality when resources are constrained rather than failing completely. For example, reducing container resource limits or disabling non-essential features when system resources are scarce.
circuit breaker	Resilience Pattern	A pattern that prevents retry loops from amplifying failures by temporarily stopping operations when failure rates exceed thresholds. Protects the system from cascading failures and allows time for recovery.

Testing and Validation

Comprehensive testing ensures container isolation, resource control, and networking function correctly across various scenarios.

Term	Category	Definition
integration testing	Testing Approach	Validation of multiple components working together to ensure the complete container runtime functions correctly. Tests end-to-end scenarios like container creation, execution, resource enforcement, and cleanup across all system components.
milestone checkpoint	Validation Point	A verification point after implementing specific functionality, with concrete tests to run and expected behavior to verify progress. Provides incremental validation that each implementation milestone meets its acceptance criteria.
test environment	Testing Infrastructure	An isolated testing setup with cleanup capabilities that provides reproducible conditions for testing container operations. Includes temporary directories, test containers, and automated cleanup to prevent test pollution.
race condition	Concurrency Bug	Timing-dependent bugs in concurrent operations where the outcome depends on the relative timing of events. Common in container runtimes due to parallel resource creation and cleanup operations requiring careful synchronization.
resource exhaustion	Testing Scenario	Test scenarios designed to verify system behavior when resources (memory, PIDs, network interfaces) approach their limits. Ensures the container runtime handles resource pressure gracefully without crashing or corrupting state.

Debugging and Diagnostics

Effective debugging techniques are essential for diagnosing issues in the complex interactions between kernel features and container runtime components.

Term	Category	Definition
namespace debugging	Debugging Technique	Systematic inspection of Linux namespace configuration and isolation to verify that containers have the expected view of system resources. Includes checking <code>/proc/PID/ns/</code> entries, mount tables, and process visibility.
cgroup debugging	Debugging Technique	Analysis of resource control group setup and enforcement to verify that containers are properly limited and monitored. Includes inspecting cgroup hierarchies, resource usage statistics, and limit enforcement.
network debugging	Debugging Technique	Diagnosis of container networking connectivity and configuration to identify why containers cannot communicate or access external resources. Uses tools like <code>ip</code> , <code>bridge</code> , <code>iptables</code> , and <code>netstat</code> to inspect network state.
process isolation	Debugging Focus	Verification that container processes are properly separated from host processes and can only see other processes in their namespace. Critical for container security and proper resource accounting.
resource pressure	Diagnostic Indicator	Metrics showing when resource usage approaches limits, potentially leading to performance degradation or failures. Includes memory pressure stall information, CPU throttling indicators, and I/O wait times.
connectivity testing	Diagnostic Process	Validation of network paths and communication capabilities to ensure containers can reach required services. Tests both inter-container communication and external network access through various protocols.
health checking	Monitoring Process	Automated validation of container runtime state and configuration to detect problems before they cause failures. Includes checking resource usage trends, network connectivity, and filesystem integrity.
mount table inspection	Debugging Technique	Examination of filesystem mount points visible to containers to verify that the expected filesystems are mounted and accessible. Uses <code>/proc/PID/mounts</code> and <code>/proc/PID/mountinfo</code> to inspect mount state.
veth pair verification	Debugging Technique	Checking that virtual ethernet interface creation and linkage function correctly, ensuring containers have proper network connectivity. Verifies interface creation, namespace assignment, and traffic flow.
iptables rule analysis	Debugging Technique	Inspection of network address translation and firewall rules to verify that port forwarding and traffic filtering work correctly. Essential for diagnosing container network access issues.

Security and Advanced Features

Advanced container runtimes implement additional security mechanisms and features beyond basic isolation.

Term	Category	Definition
seccomp filters	Security Mechanism	System call level security restrictions that limit which system calls container processes can execute. Implemented using Berkeley Packet Filter (BPF) programs loaded into the kernel to intercept and filter system calls.
mandatory access control	Security Framework	Policy-based security mechanisms beyond traditional Unix permissions, including SELinux and AppArmor. Provide additional layers of access control based on security policies rather than just file ownership and permissions.
capability management	Security Feature	Fine-grained privilege control system that divides root privileges into distinct capabilities (e.g., CAP_NET_ADMIN, CAP_SYS_ADMIN). Allows containers to perform specific privileged operations without full root access.
defense in depth	Security Strategy	The approach of implementing multiple independent security layers so that if one layer is compromised, other layers continue to provide protection. Combines namespaces, cgroups, seccomp, capabilities, and MAC for comprehensive security.
supply chain security	Security Concern	Verification of software component authenticity and integrity throughout the development and deployment pipeline. Includes image signing, vulnerability scanning, and provenance tracking for container images.

Networking Extensions

Advanced networking features enable sophisticated container network topologies and service mesh integration.

Term	Category	Definition
overlay networking	Network Architecture	Network virtualization that creates logical networks spanning multiple hosts using encapsulation protocols like VXLAN or GRE. Enables containers on different hosts to communicate as if on the same network segment.
Container Network Interface	Standard Interface	A specification and plugin architecture for configuring network interfaces in containers. Provides a standard way to integrate different networking solutions with container runtimes and orchestrators like Kubernetes.
network policies	Security Feature	Declarative rules that control traffic flow between containers and external endpoints. Implemented at the network level to provide microsegmentation and traffic filtering based on labels, namespaces, and other criteria.
quality of service	Network Feature	Traffic prioritization and bandwidth management mechanisms that ensure critical applications receive adequate network resources. Includes traffic shaping, priority queuing, and bandwidth guarantees.
service mesh	Infrastructure Layer	A dedicated infrastructure layer that handles service-to-service communication, providing features like traffic management, security, and observability. Often implemented using sidecar proxies alongside application containers.
traffic interception	Network Feature	The ability to redirect network traffic through proxies or filters for processing, monitoring, or security scanning. Essential for service mesh implementations and network security appliances.

Storage and Image Management

Container storage systems provide persistent data management and efficient image distribution mechanisms.

Term	Category	Definition
image registry	Storage System	A repository for storing and distributing container images, supporting features like authentication, authorization, and content distribution. Examples include Docker Hub, Amazon ECR, and private registry implementations.
vulnerability scanning	Security Process	Automated security analysis of container images to identify known vulnerabilities in installed packages and dependencies. Integrates with image registries and CI/CD pipelines to prevent deployment of vulnerable images.
persistent volumes	Storage Abstraction	Storage that survives container lifecycle events, allowing data to persist across container restarts and migrations. Implemented through volume mounts that connect containers to external storage systems.
content-addressable storage	Storage Architecture	Storage organized by content hash rather than filename, ensuring data integrity and enabling efficient deduplication. Used in container image storage to share common layers between images and detect corruption.
dynamic provisioning	Storage Feature	Automatic storage allocation based on demand rather than pre-allocated storage pools. Enables containers to request storage resources dynamically and have them provisioned from available storage systems.

Container Orchestration Interface

Modern container runtimes often integrate with orchestration systems and provide standardized interfaces.

Term	Category	Definition
Container Runtime Interface	Standard Interface	A Kubernetes standard interface that defines how container orchestrators communicate with container runtimes. Enables different container runtimes to work with Kubernetes without requiring orchestrator-specific integration code.
sidecar injection	Deployment Pattern	The automatic deployment of auxiliary containers alongside application containers to provide additional functionality like networking, security, or monitoring. Common in service mesh implementations for proxy injection.

This glossary provides the essential vocabulary needed to understand and implement our container runtime system. Each term is carefully chosen to represent concepts you'll encounter while working through the four implementation milestones, from basic namespace isolation to advanced networking features.

Implementation Guidance

Understanding these terms is crucial for successful implementation of your container runtime. As you work through each milestone, refer back to these definitions to ensure you're using terminology correctly and

understanding the underlying concepts.

Key Term Categories for Each Milestone:

Milestone	Primary Term Categories	Focus Areas
Milestone 1	Namespace Types, Kernel Features, Process Isolation	Understanding how namespaces create isolated views of system resources
Milestone 2	Cgroup Controllers, Resource Control, Monitoring	Learning how cgroups enforce resource limits and provide usage statistics
Milestone 3	Filesystem Concepts, Layer Stacking, Copy-on-Write	Mastering overlayfs mechanics and layer management
Milestone 4	Container Networking, NAT Rules, Interface Management	Implementing virtual networking with bridges and port forwarding

Terminology Usage Guidelines:

When implementing your container runtime, consistently use these terms in your code comments, documentation, and variable names. This consistency helps maintain clarity and makes your code easier to understand and maintain.

For example, when implementing namespace creation, use terms like "namespace handle" for data structures that track namespace resources, "process isolation" when describing the security benefits, and "mount namespace" when specifically referring to filesystem isolation.

Common Terminology Mistakes:

⚠ Pitfall: Mixing cgroups v1 and v2 terminology Using cgroups v1 terms like "memory.limit_in_bytes" when implementing cgroups v2, which uses "memory.max". Always verify you're using the correct terminology for your target cgroup version.

⚠ Pitfall: Confusing overlay filesystem layers Referring to "upper layers" (plural) when overlayfs has only one upper directory but multiple lower directories. The terminology is specific: one upper directory, multiple lower directories, one merged view.

⚠ Pitfall: Misusing network namespace terminology Calling veth pairs "virtual interfaces" without clarifying they're specifically ethernet pairs. The term "veth pair" is precise and indicates the bidirectional nature of the connection.

Reference Implementation Notes:

When reading container runtime source code (Docker, containerd, runc), you'll encounter these same terms used consistently across implementations. Understanding this standard terminology helps you read and contribute to existing projects beyond your own implementation.

The Linux kernel documentation uses these exact terms, so mastering this vocabulary helps you understand system documentation, manual pages, and kernel interfaces more effectively.