Build Your Own Docker: Design Document

Overview

This document outlines the architecture for a container runtime that isolates processes using Linux primitives like namespaces, cgroups, and union filesystems. The key architectural challenge is orchestrating these low-level OS features into a coherent, secure, and user-friendly containerization system while maintaining compatibility with OCI standards.

> 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):** All milestones (provides foundational understanding)

Mental Model: Apartments in a Building

Think of your computer's operating system as a large apartment building. Each **process** (a running program like a web server or database) is a **tenant** living in this building. By default, all tenants share common spaces and resources:

- They see the same **directory of residents** (process table)
- They share the same **mailroom and packages** (filesystem)
- They use the same **utility meters** (CPU, memory)
- They're connected to the same **phone system** (network)
- They have the same **building nameplate** (hostname)

This shared arrangement causes problems. A noisy tenant (a buggy process consuming all memory) disturbs everyone. A tenant rummaging through the mailroom (a process reading sensitive files) violates privacy. If one tenant changes the building's nameplate (hostname), it changes for everyone.

Containerization is like giving each tenant their own fully-contained apartment unit within the same building:

Apartment Feature	Container Equivalent	Why It Matters
Private walls	**Namespaces** (PID, network, mount)	Tenants can't see or interfere with each other's activities
Individual utility meters	**cgroups** (CPU, memory limits)	No single tenant can monopolize building resources
Custom furniture layout	**Root filesystem** (chroot/pivot_root)	Each tenant can decorate their space without affecting others
Layered wallpaper	**Union filesystems** (OverlayFS)	Common decorative elements are shared, saving space
Private phone line	**Network namespace** (veth pairs)	Tenants have separate phone numbers and can't eavesdrop

The key architectural challenge is that while these apartments exist within the same physical building (the same Linux kernel), they must appear to be completely independent living spaces. The building superintendent (our container runtime) must:

1. Construct these apartments on demand
2. Install appropriate isolation features
3. Connect utilities while maintaining separation
4. Clean up when tenants leave

This mental model helps understand why we need multiple Linux primitives working together: namespaces provide the walls, cgroups provide the resource meters, and filesystem isolation provides the private furnishings.

The Isolation Problem in Concrete Terms

Consider a typical deployment scenario: you need to run a web application consisting of a Node.js API server, a PostgreSQL database, and a Redis cache on a single physical server. Without isolation, these three processes:

1. **See each other's processes** - The Node.js process can see and send signals to PostgreSQL workers
2. **Share filesystem access** - A bug in Redis could delete PostgreSQL data files

3. **Compete for resources** - A memory leak in Node.js could starve PostgreSQL of memory
4. **Conflict on network ports** - All three try to listen on port 80 or use the same loopback interface
5. **Share host identity** - All three report the same hostname to monitoring systems

The isolation problem manifests in these concrete technical requirements:

Requirement 1: Process Visibility Isolation

- **Problem**: `ps aux` run from within the Node.js process shows PostgreSQL and Redis workers
- **Consequence**: Accidental termination, security exposure of process arguments
- **Solution Needed**: Private process ID space where the containerized process sees itself as PID 1

Requirement 2: Filesystem Boundary Enforcement

- **Problem**: A path traversal bug in the web application allows reading `/etc/shadow`
- **Consequence**: Complete system compromise via credential theft
- **Solution Needed**: A virtual root directory that appears to be the entire filesystem

Requirement 3: Resource Fairness Guarantees

- **Problem**: Redis cache grows uncontrollably and triggers system-wide Out-Of-Memory killer
- **Consequence**: Random important processes (SSH daemon, system logger) get killed
- **Solution Needed**: Hard memory limits that only affect the offending container

Requirement 4: Network Stack Separation

- **Problem**: Redis binds to `127.0.0.1:6379`, preventing PostgreSQL from using loopback for replication
- **Consequence**: Service conflicts, inability to run multiple instances
- **Solution Needed**: Private network interfaces and routing tables per container

Requirement 5: Host Identity Segregation

- **Problem**: All services report the same hostname to centralized logging
- **Consequence**: Debugging nightmare, monitoring confusion
- **Solution Needed**: Independent UTS (Unix Timesharing System) namespace

The following table illustrates how these isolation failures manifest in practice:

Isolation Failure	Symptom	Attack Vector	Business Impact
PID namespace	Container process kills host `systemd`	Malicious container sends `SIGKILL` to PID 1	System crash, requiring physical access
Mount namespace	Container modifies host `/etc/passwd`	Path traversal or symlink attack	Complete system compromise
Network namespace	Container sniffs other containers' traffic	ARP spoofing in shared network	Data breach across tenants
cgroup failure	Container triggers host OOM killer	Memory exhaustion attack	Service outage for all containers
UTS namespace	Container changes system hostname	Service masquerading	Monitoring/alerting system failure

> **Key Insight**: Isolation isn't just about security—it's about predictability. The fundamental value proposition of containers is that they provide **predictable, reproducible execution environments** regardless of the underlying host system configuration.

Existing Isolation Approaches Comparison

Before containers became mainstream, several approaches addressed isolation with different trade-offs in security, performance, and complexity. Understanding these alternatives clarifies why the Linux namespace+cgroup approach prevailed for application packaging and deployment.

Approach 1: Physical Servers The most complete isolation but maximum overhead.

Aspect	Physical Server	Container
Isolation Level	Complete physical separation	Kernel-level logical separation
Startup Time	Minutes to hours	Milliseconds to seconds
Resource Efficiency	Poor (idle resources wasted)	Excellent (shared kernel, dynamic allocation)
Density	1 application per server	10s-100s per server
Operational Overhead	High (individual maintenance)	Low (orchestrated management)
Use Case	Legacy applications, maximum security	Microservices, CI/CD, PaaS

Approach 2: Virtual Machines (VMs) Hardware-level virtualization with guest operating systems.

Aspect	Virtual Machine	Container
Abstraction Level	Hardware (CPU, memory, devices)	Operating system (processes, files, network)
Guest OS	Full independent kernel	Shared host kernel
Isolation Mechanism	Hypervisor + VM boundaries	Linux namespaces + cgroups
Overhead	Higher (full OS + virtualization)	Lower (direct syscalls)
Image Size	GBs (entire OS)	MBs (app + dependencies)
Boot Time	Seconds to minutes	Sub-second
Security Boundary	Strong (hardware isolation)	Weaker (kernel shared)
Use Case	Multi-OS environments, strong isolation	Single-OS app packaging, high density

Approach 3: chroot Jails Filesystem isolation only—the historical precursor.

Aspect	chroot Jail	Container
Filesystem Isolation	Yes (but escapable)	Yes (with mount namespace)
Process Isolation	No (shared PID space)	Yes (PID namespace)
Network Isolation	No (shared stack)	Yes (network namespace)
Resource Limits	No	Yes (cgroups)
Security	Weak (many escape vectors)	Moderate (with seccomp, SELinux)
Complexity	Low	Moderate
Modern Relevance	Legacy, specialized use cases	Standard application deployment

Approach 4: BSD Jails / Solaris Zones OS-level containerization in other Unix variants.

Aspect	BSD Jails	Linux Containers
OS Family	BSD derivatives (FreeBSD)	Linux
Isolation Model	Single unified jail mechanism	Composition of independent namespaces
Configuration	Centralized jail configuration	Distributed across multiple subsystems
Ecosystem	Smaller tooling ecosystem	Vast (Docker, Kubernetes, etc.)
Adoption	Niche, specialized deployments	Industry standard

Approach 5: Application Sandboxing (Firejail, Bubblewrap) User-space containment for desktop applications.

Aspect	Application Sandbox	System Container
Target Use Case	Desktop applications	Server applications
Privilege Level	Often user namespace only	Full root capabilities
Orchestration	Manual, per-application	Automated, cluster-wide
Networking	Typically shared	Configurable topologies
Image Format	None (uses host files)	Standardized (OCI)

> **Architecture Decision Record: Choosing Linux Primitives Over Alternatives**

Decision: Use Linux namespaces + cgroups for container isolation

Context: We need to build a container runtime for educational purposes that demonstrates real-world containerization principles while running efficiently on commodity Linux systems.

Options Considered:

1. **Full virtualization (KVM/QEMU)** - Complete isolation via hardware virtualization
2. **User-space virtualization (gVisor)** - Syscall interception for stronger security
3. **Linux namespaces + cgroups** - Native kernel features for OS-level virtualization

Decision: Implement using Linux namespaces and cgroups.

Rationale:

1. **Educational value**: Namespaces and cgroups are the foundational primitives underlying Docker, LXC, and Kubernetes, making this knowledge transferable to production systems.
2. **Minimal overhead**: No hypervisor layer or syscall translation means near-native performance for learning resource management concepts.
3. **Incremental complexity**: Each primitive (PID, mount, network namespaces) can be implemented and tested independently, matching our milestone-based approach.
4. **Ubiquitous availability**: Available on any modern Linux kernel without special hardware support (VT-x/AMD-V).
5. **Compatibility with OCI**: The Open Container Initiative runtime specification expects these Linux primitives.

Consequences:

- **Positive**: Learners understand the actual mechanisms used by Docker, can debug production container issues, and gain low-level Linux systems knowledge.
- **Negative**: Limited to Linux hosts, requires root privileges, and provides weaker security boundaries than virtualization (though can be enhanced with seccomp, SELinux).
- **Trade-off**: We accept platform limitation for deeper understanding of the dominant containerization technology.

Option	Pros	Cons	Why Not Chosen
Full Virtualization	Strong isolation, multi-OS support, mature security model	High overhead, slow startup, complex implementation	Overkill for application packaging, obscures the Linux primitives we want to teach
User-space Virtualization	Strong security via syscall filtering, no kernel sharing	Performance overhead, compatibility issues with some syscalls	Adds abstraction layer that hides the underlying Linux mechanisms
Linux Primitives	Native performance, direct kernel interaction, industry standard	Weaker isolation, Linux-only, requires privileged operations	**CHOSEN**: Best for educational goals, matches real-world container runtimes

The Evolution to Modern Containers:

The containerization approach we're implementing represents an evolutionary convergence of several historical threads:

1. **2000: FreeBSD Jails** - Introduced the concept of OS-level virtualization
2. **2005: Solaris Zones** - Commercial implementation with resource controls
3. **2008: LXC (Linux Containers)** - First complete container implementation for Linux
4. **2013: Docker** - Developer-friendly tooling and image format
5. **2015: OCI Specification** - Standardization of runtime and image formats
6. **2017: Kubernetes dominance** - Orchestration ecosystem maturation

Our implementation follows the architectural pattern established by LXC and standardized by OCI: composing discrete Linux kernel features into a coherent isolation boundary. This composition is both a strength (flexibility, incremental adoption) and a complexity (multiple subsystems to configure correctly).

Common Misconceptions About Container Isolation:

⚠️ Pitfall: Assuming containers provide VM-level security

- **Misconception**: "My containers are as isolated as virtual machines"
- **Reality**: Containers share the kernel, making kernel exploits catastrophic
- **Fix**: Use additional hardening (seccomp profiles, SELinux, user namespaces) for production

⚠️ Pitfall: Believing cgroups prevent all resource exhaustion

- **Misconception**: "Memory limit protects against all memory issues"
- **Reality**: cgroups memory limit doesn't cover page cache, swap, or kernel memory
- **Fix**: Set multiple limits (`memory.limit_in_bytes` , `memory.kmem.limit_in_bytes`)

⚠️ Pitfall: Thinking network namespace equals network security

- **Misconception**: "Network namespace prevents all network attacks"
- **Reality**: Containers on same bridge can ARP spoof each other
- **Fix**: Use network policies, ebtables rules, or separate bridges

The following table summarizes how our container runtime addresses these historical approaches' limitations:

Historical Limitation	Our Solution	Implementation Component
chroot escape via `..`	Mount namespace + pivot_root	Milestone 3
Resource monopolization	cgroups with memory/CPU limits	Milestone 2
Process visibility	PID namespace	Milestone 1
Network port conflicts	Network namespace + veth pairs	Milestone 5
Storage inefficiency	OverlayFS layered filesystem	Milestone 4
Non-standard formats	OCI image specification	Milestone 6

By understanding this context, learners appreciate that containerization isn't a revolutionary new technology but rather a clever composition of existing Linux features with developer-friendly tooling. This composition approach is what makes containers simultaneously powerful (leveraging decades of OS research) and accessible (hiding the complexity behind simple commands).

> **Milestone(s):** All milestones (providing foundational requirements and scope boundaries)

Goals and Non-Goals

This section establishes clear boundaries for what our container runtime will and will not accomplish. Building a production-grade container runtime like Docker involves immense complexity; our educational implementation focuses on the core isolation primitives while deliberately excluding enterprise features. These definitions serve as guardrails throughout development, ensuring we build something complete enough to demonstrate containerization concepts while remaining achievable within educational constraints.

Must-Have Goals

The following capabilities represent the non-negotiable requirements for our container runtime. Each goal maps directly to one or more milestones and implements a fundamental containerization concept.

Goal	Milestone	Concept Demonstrated	User Benefit	Implementation Requirements
Process Isolation via Namespaces	1	Kernel-level isolation of global system resources	Processes cannot see or interfere with other containers or the host system	Create PID, UTS, mount, network, and IPC namespaces using `clone()` or `unshare()` system calls
Resource Limiting via cgroups	2	Enforce CPU, memory, and process count limits	Prevent any single container from monopolizing system resources	Create cgroup hierarchy, write limit values to control files, attach container processes
Filesystem Isolation via pivot_root	3	Provide containers with their own root filesystem	Container sees only its own files, cannot access host filesystem	Extract base image, set up mount namespace, call `pivot_root()`, mount `/proc` and `/sys`
Copy-on-Write Layers via OverlayFS	4	Efficient storage and sharing of container images	Multiple containers can share common base layers, saving disk space	Configure OverlayFS mounts with lower (read-only), upper (writable), and work directories
Container Networking via veth pairs	5	Isolated network stack with external connectivity	Containers get private IP addresses and can communicate with each other and the internet	Create network namespace, set up veth pair, configure bridge, add iptables NAT rules
OCI Image Compatibility	6	Standardized container image format	Can run containers from images built by Docker and other OCI-compliant tools	Parse OCI image manifests, download layers from registries, follow layer extraction specifications
Basic CLI Lifecycle Management	6	Command-line interface for container operations	Users can create, start, stop, and remove containers with simple commands	Implement `create`, `start`, `stop`, `remove` commands with appropriate state transitions
Clean Resource Cleanup	All	Proper cleanup of all isolation primitives	No orphaned namespaces, cgroups, or network interfaces after container removal	Track all created resources and implement cleanup handlers for all termination paths
Educational Clarity	All	Code demonstrates concepts clearly	Learners understand how each Linux primitive contributes to containerization	Well-commented code, minimal abstraction layers, explicit error handling

> **Key Insight:** These goals form a **minimum viable container runtime**—enough to demonstrate how Docker works under the hood without implementing the complete feature set of production container runtimes. Each goal directly corresponds to a specific Linux kernel feature that containers rely upon.

Architecture Decision Record: Scope of Implementation

Decision: Build an Educational Implementation Rather Than Production-Grade

- **Context**: This project aims to teach container internals, not to compete with Docker, containerd, or runc. Learners need to understand the fundamental Linux primitives without getting overwhelmed by enterprise features.
- **Options Considered**:
  1. **Full OCI Runtime Implementation**: Implement complete OCI runtime specification including all optional features, hooks, and security configurations.
  2. **Educational Subset**: Implement only the core primitives that demonstrate container concepts, skipping complex edge cases and optimization.
  3. **Docker-Compatible CLI**: Build a Docker CLI clone with identical command structure and flags.
- **Decision**: Choose option 2 (Educational Subset) with basic OCI compatibility for image handling.
- **Rationale**:
  - Learning objectives focus on understanding namespaces, cgroups, and filesystem isolation, not production deployment concerns.
  - A simpler implementation reduces cognitive load, allowing learners to focus on core concepts.
  - Basic OCI compatibility demonstrates real-world standards without requiring full spec compliance.
- **Consequences**:
  - Cannot run all Docker images (some require features we don't implement).
  - Performance and security are not production-grade.
  - Learners gain deep understanding of fundamentals that apply to all container runtimes.

The Must-Have Goals in Practice:

For each goal, we define concrete success criteria:

1. **Process Isolation Success**: When running `ps aux` inside a container, the process list shows only processes within that container, with the container process as PID 1. Changing the hostname inside the container (`hostname mycontainer`) doesn't affect the host.
2. **Resource Limiting Success**: A container configured with 100MB memory limit gets terminated by the OOM killer when it allocates 101MB. A container with CPU shares of 512 gets approximately half the CPU time of a container with 1024 shares under contention.
3. **Filesystem Isolation Success**: A container cannot read `/etc/hostname` from the host filesystem. The container's `/` directory shows files from the container image, not host files.
4. **Layered Filesystem Success**: Two containers using the same base Ubuntu image share the read-only layers on disk. When one container modifies a file, the other container still sees the original version.
5. **Networking Success**: Containers get IP addresses like `172.17.0.2` and can ping each other. Containers can access the internet via NAT. Host can forward port 8080 to a container's port 80.
6. **Image Compatibility Success**: Can pull and run `alpine:latest` from Docker Hub. The container runs the correct entrypoint command with the expected environment.
7. **CLI Success**: Users can run `./byod run -it ubuntu:latest /bin/bash` and get an interactive shell in a container, then exit and clean up with `./byod rm <container-id>` .

Explicit Non-Goals

The following capabilities are explicitly **not** part of our implementation. Understanding what we're omitting is as important as understanding what we're including—it defines the boundaries of our learning exercise.

Non-Goal	Reason for Exclusion	Real-World Implementation (for context)	What We Do Instead
Production Security Hardening	Security is a complex domain requiring expertise beyond educational scope. Our implementation runs as root and doesn't implement security best practices.	Docker uses seccomp, SELinux/AppArmor, capabilities dropping, user namespaces with ID mapping	Run containers as root without additional security constraints; assume trusted environment
Orchestration Features	Container orchestration (scheduling, service discovery, load balancing) is a separate domain from container runtime fundamentals.	Kubernetes, Docker Swarm, Nomad manage container lifecycle across hosts	Single-host runtime only; no clustering or scheduling
Volume Management	Persistent storage involves complex lifecycle management and driver plugins beyond core isolation concepts.	Docker volumes with various storage drivers (local, NFS, cloud storage)	No persistent volume support; all container filesystem changes are ephemeral (in upper layer)
Container Registry Server	Building a registry server duplicates effort and doesn't teach container runtime fundamentals.	Docker Registry, Amazon ECR, Google Container Registry	Client-only implementation that pulls from existing registries (Docker Hub)
Windows/MacOS Support	Container internals differ fundamentally on non-Linux systems; our focus is Linux primitives.	Docker Desktop uses Linux VMs on Windows/Mac	Linux-only implementation requiring Linux kernel features
GPU/Device Passthrough	Specialized hardware access requires complex driver integration and security considerations.	NVIDIA Container Toolkit, device plugins in Kubernetes	No device passthrough; containers access only standard virtual devices
Live Migration	Process migration across hosts involves complex memory and filesystem synchronization beyond educational scope.	CRIU (Checkpoint/Restore In Userspace) for container migration	Single-host containers that cannot be checkpointed or migrated
Logging and Monitoring Infrastructure	Production observability requires distributed systems design for log aggregation and metrics collection.	Docker logging drivers, Prometheus metrics, Fluentd log forwarding	Simple stdout/stderr capture to terminal; no log persistence or metrics
Image Build System	Building images involves Dockerfile parsing, layer caching, and optimization strategies that warrant separate focus.	Docker Build, BuildKit, Kaniko	Pull and run pre-built images only; no `docker build` equivalent
Networking Plugins	Advanced networking (overlay networks, service mesh, network policies) requires complex network virtualization.	CNI plugins, Calico, Flannel, Weave Net	Single bridge network with NAT; no plugin system or advanced topologies
Resource Quota Enforcement	Fine-grained resource accounting (disk I/O, network bandwidth) requires complex cgroup controllers.	Docker uses blkio, net_cls, and net_prio cgroup controllers	Basic CPU and memory limits only; no I/O or network bandwidth limiting
Health Checks and Liveness Probes	Container health monitoring requires process supervision and restart policies beyond simple process execution.	Docker HEALTHCHECK instruction, Kubernetes liveness/readiness probes	No health monitoring; container runs until main process exits

> **Important Boundary:** By explicitly stating these non-goals, we acknowledge that real container runtimes are far more complex than our implementation. This allows learners to focus on the fundamentals while understanding where additional complexity would be needed for production use.

Common Pitfalls to Avoid Given Our Non-Goals:

1. ⚠️ **Pitfall: Attempting to Implement Security Features Prematurely**
   - **Description**: Learners might try to add user namespace mapping or seccomp profiles before mastering basic namespace isolation.
   - **Why It's Wrong**: Adds unnecessary complexity when we're still learning fundamentals. Security without understanding the base isolation is fragile.
   - **Fix**: Focus on making basic isolation work first. Add security features only after core functionality is stable.
2. ⚠️ **Pitfall: Over-Engineering the CLI**
   - **Description**: Spending excessive time building a Docker-compatible CLI with all flags and subcommands.
   - **Why It's Wrong**: Distracts from the core learning objective of understanding Linux primitives. The CLI is just an interface to the runtime.
   - **Fix**: Implement minimal CLI with just `run`, `ps`, `stop`, and `rm` commands. Use simple flag parsing.

3. ⚠️ **Pitfall: Adding Orchestration Features**
    - **Description**: Trying to implement multi-container applications or inter-container networking beyond basic bridge networking.
    - **Why It's Wrong**: Orchestration builds upon container runtime fundamentals but is a separate layer of abstraction.
    - **Fix**: Focus on single-container isolation. Multi-container coordination is out of scope.

Design Accommodations for Future Extensions:

While these are non-goals for the initial implementation, our architecture should not preclude adding them later:

1. **Security Features**: The `ContainerConfig` type includes fields that could be extended for security settings (capabilities, seccomp profiles, user namespace mappings).
2. **Volume Support**: The `FilesystemManager.SetupRootfs` method could be extended to mount additional volumes before container start.
3. **Advanced Networking**: The `NetworkConfig` type's `Mode` field could support values beyond "bridge" (e.g., "host", "none", "overlay").
4. **Logging**: The `Container` struct could include log file paths, and the runtime could redirect stdout/stderr to files.

These accommodations mean we choose extensible designs even for features we don't implement, following good software design principles while maintaining focus on educational objectives.

Implementation Guidance

Technology Recommendations Table:

Component	Simple Option (Recommended)	Advanced Option (If Time Permits)
CLI Parsing	`flag` package (Go standard library)	`cobra` library for richer CLI experience
Image Registry Client	HTTP client with basic auth for Docker Hub	Full OCI distribution client with token auth
JSON Parsing	`encoding/json` for OCI manifests	Streaming JSON parser for large manifests
Process Management	`os/exec` for host process management	Direct syscall usage for finer control
Network Configuration	`netlink` library for veth creation	Raw socket operations for custom networking

Recommended File/Module Structure:

```
build-your-own-docker/
├── cmd/
│   └── byod/                    # CLI entry point
│       └── main.go              # CLI command parsing and routing
├── internal/
│   ├── runtime/                 # Core container runtime
│   │   ├── runtime.go           # ContainerRuntime implementation
│   │   ├── container.go         # Container struct and lifecycle methods
│   │   └── state.go             # ContainerState enum and state machine logic
│   ├── namespaces/              # Milestone 1: Process isolation
│   │   ├── manager.go           # NamespaceManager implementation
│   │   └── flags.go             # Namespace flag constants and helpers
│   ├── cgroups/                 # Milestone 2: Resource limits
│   │   ├── manager.go           # CgroupManager implementation
│   │   ├── v1/                  # cgroup v1 implementation
│   │   └── v2/                  # cgroup v2 implementation
│   ├── filesystem/              # Milestones 3 & 4: Filesystem isolation
│   │   ├── manager.go           # FilesystemManager implementation
│   │   ├── rootfs.go            # pivot_root and mount setup
│   │   └── overlay.go           # OverlayFS layer management
│   ├── network/                 # Milestone 5: Container networking
│   │   ├── manager.go           # NetworkManager implementation
│   │   ├── veth.go              # Virtual ethernet pair creation
│   │   └── bridge.go            # Bridge network setup
│   ├── image/                   # Milestone 6: OCI image handling
│   │   ├── manager.go           # Image pull and parse
│   │   ├── oci/                 # OCI spec structures
│   │   └── registry/            # Registry client
│   └── storage/                 # Container and image storage
│       ├── store.go             # Persistent storage interface
│       └── jsonfile/            # JSON file-based storage implementation
├── pkg/
│   └── types/                   # Shared data types
│       ├── container.go         # Container, ContainerConfig types
│       ├── image.go             # Image, ImageLayer types
│       └── network.go           # NetworkConfig, PortMapping types
└── scripts/                     # Helper scripts for testing
    ├── test-isolation.sh        # Test namespace isolation
    └── setup-bridge.sh          # Setup bridge network for testing
```

Infrastructure Starter Code (Complete, ready to use):

```go
// pkg/types/container.go

package types

import "time"

type ContainerState string

const (
    StateCreated ContainerState = "created"
    StateRunning ContainerState = "running"
    StatePaused  ContainerState = "paused"
    StateStopped ContainerState = "stopped"
    StateRemoved ContainerState = "removed"
)

type Container struct {
    ID        string         `json:"id"`
    Name      string         `json:"name"`
    State     ContainerState `json:"state"`
    Config    ContainerConfig `json:"config"`
    Pid       int            `json:"pid,omitempty"`
    CreatedAt time.Time      `json:"created_at"`
}

type ContainerConfig struct {
    Image      string         `json:"image"`
    Cmd        []string       `json:"cmd"`
    Env        []string       `json:"env"`
    WorkingDir string         `json:"working_dir"`
    Hostname   string         `json:"hostname"`
    Limits     ResourceLimits `json:"limits"`
    Network    NetworkConfig  `json:"network"`
}

type ResourceLimits struct {
    MemoryMB  int `json:"memory_mb"`
    CPUShares int `json:"cpu_shares"`
    PidsLimit int `json:"pids_limit"`
}

type NetworkConfig struct {
    Mode         string        `json:"mode"`
```

```
    BridgeName   string        `json:"bridge_name"`
    IPAddress    string        `json:"ip_address"`
    PortMappings []PortMapping `json:"port_mappings"`
}

type PortMapping struct {
    HostPort      int    `json:"host_port"`
    ContainerPort int    `json:"container_port"`
    Protocol      string `json:"protocol"`
}
```

Core Logic Skeleton Code:

GO

```go
// internal/runtime/runtime.go

package runtime


import (

    "errors"

    "github.com/your-org/byod/pkg/types"

)


type ContainerRuntime struct {

    containers map[string]*types.Container

    // TODO: Add other managers as fields

}


func NewContainerRuntime() *ContainerRuntime {

    // TODO 1: Initialize the ContainerRuntime struct

    // TODO 2: Set up the containers map

    // TODO 3: Initialize any sub-managers (namespace, cgroup, filesystem, network)

    // TODO 4: Return the initialized runtime

    return nil

}


func (r *ContainerRuntime) CreateContainer(config types.ContainerConfig) (types.Container, error) {

    // TODO 1: Validate the ContainerConfig (check required fields)

    // TODO 2: Generate a unique container ID (use UUID or hash)

    // TODO 3: Create Container struct with initial state (StateCreated)

    // TODO 4: Store the container in runtime's container map

    // TODO 5: Set up container root directory in filesystem

    // TODO 6: Return the created container or error

    return types.Container{}, errors.New("not implemented")

}


func (r *ContainerRuntime) StartContainer(id string) error {

    // TODO 1: Look up container by ID, return error if not found

    // TODO 2: Check container is in StateCreated or StateStopped (valid start states)

    // TODO 3: Update container state to StateRunning

    // TODO 4: Create namespaces for the container (using NamespaceManager)

    // TODO 5: Set up cgroups with resource limits (using CgroupManager)

    // TODO 6: Set up root filesystem with pivot_root (using FilesystemManager)

    // TODO 7: Configure network namespace (using NetworkManager)

    // TODO 8: Execute the container's entrypoint command inside the namespaces

    // TODO 9: Record the container's PID
```

```
    // TODO 10: Handle errors and cleanup if any step fails
    return errors.New("not implemented")
}

func (r *ContainerRuntime) StopContainer(id string) error {
    // TODO 1: Look up container by ID, return error if not found
    // TODO 2: Check container is in StateRunning (can't stop non-running containers)
    // TODO 3: Send SIGTERM to container process (use the recorded PID)
    // TODO 4: Wait for process to exit (with timeout)
    // TODO 5: If timeout, send SIGKILL
    // TODO 6: Update container state to StateStopped
    // TODO 7: Clean up network namespace (but keep other resources for possible restart)
    // TODO 8: Handle errors appropriately
    return errors.New("not implemented")
}

func (r *ContainerRuntime) RemoveContainer(id string) error {
    // TODO 1: Look up container by ID, return error if not found
    // TODO 2: Check container is in StateStopped or StateCreated (can't remove running)
    // TODO 3: If container has root filesystem, clean it up
    // TODO 4: Remove cgroup hierarchy
    // TODO 5: Remove container from runtime's container map
    // TODO 6: Update container state to StateRemoved
    // TODO 7: Handle errors and ensure cleanup is idempotent
    return errors.New("not implemented")
}
```

Language-Specific Hints:

1. **Go syscall Package**: Use `syscall.Syscall` or `syscall.RawSyscall` for direct system calls like `clone()`, `unshare()`, and `pivot_root()`.
2. **Namespace Flags**: Use the exact constants from the naming conventions: `syscall.CLONE_NEWPID`, `syscall.CLONE_NEWNET`, etc.
3. **Process Execution**: Use `syscall.Exec` to replace the current process with the container's entrypoint, not `os/exec.Command` which creates a child process.
4. **Error Handling**: Check `errno` values after syscalls using `syscall.Errno` type to determine specific failure reasons.
5. **File Operations**: Use `os` package for file operations, but remember `os.RemoveAll` for cleanup of container directories.
6. **JSON Serialization**: Use `encoding/json` with struct tags for OCI manifest parsing and container state persistence.

Milestone Checkpoint - Goals Validation:

After implementing all six milestones, verify our must-have goals by running:

```bash
# Test process isolation
sudo ./byod run --hostname mycontainer alpine:latest hostname
# Should output: mycontainer (not the host's hostname)

# Test resource limits
sudo ./byod run --memory 100m alpine:latest sh -c "tail /dev/zero"
# Should be OOM-killed within a few seconds

# Test filesystem isolation
sudo ./byod run alpine:latest ls /
# Should show alpine rootfs contents, not host root files

# Test layered filesystem
sudo ./byod run alpine:latest touch /testfile
sudo ./byod run alpine:latest ls /testfile
# First command creates file, second should NOT see it (separate containers)

# Test networking
sudo ./byod run --net bridge alpine:latest ip addr show
# Should show veth interface with private IP (172.17.0.x)

# Test OCI image compatibility
sudo ./byod pull ubuntu:latest
sudo ./byod run ubuntu:latest echo "Hello from Ubuntu"
# Should successfully download and run Ubuntu image
```

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Container exits immediately	Missing or incorrect entrypoint command	Check OCI image config for `Entrypoint` and `Cmd`	Use image's entrypoint or provide command in `ContainerConfig.Cmd`
`ps aux` shows host processes	PID namespace not properly isolated	Check `clone()` flags include `CLONE_NEWPID`	Ensure PID namespace is created before process starts
Container cannot access internet	Missing NAT rules or DNS	Check `iptables -t nat -L` for MASQUERADE rule	Add iptables masquerade rule for bridge network
"Operation not permitted" errors	Running without root privileges	Check `id` command output	Run container runtime with `sudo`
OverlayFS mount fails	Missing kernel module or incorrect options	Check `dmesg | grep overlay`	Load overlay module: `sudo modprobe overlay`

High-Level Architecture

> **Milestone(s):** All milestones (this section provides the overall architectural framework that unifies all six implementation milestones)

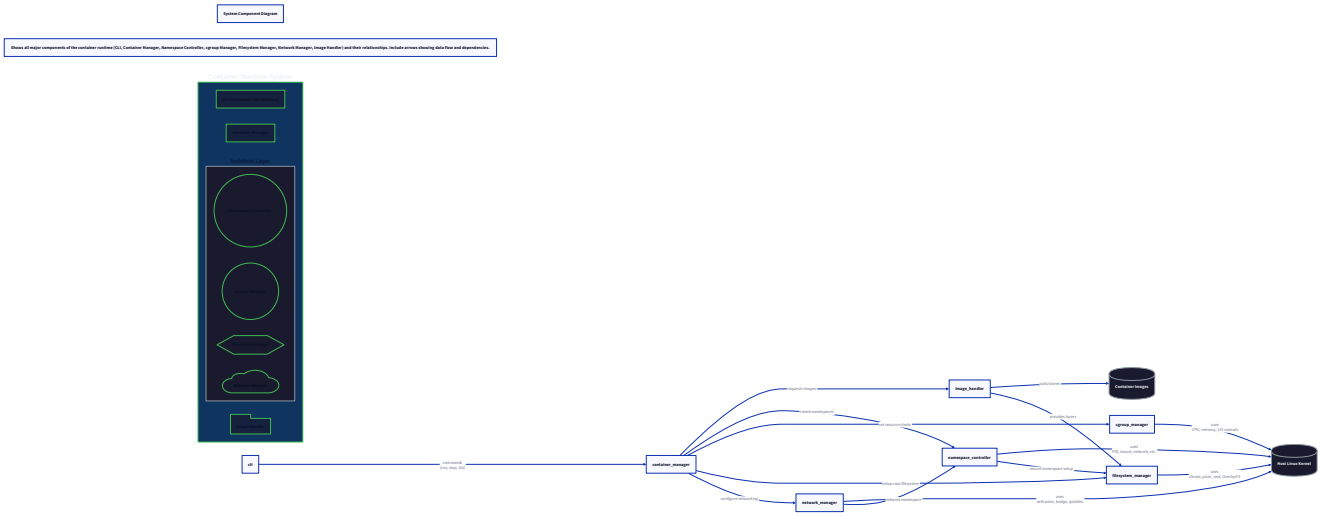
This section describes the macro-architecture of our container runtime—how all the Linux primitives (namespaces, cgroups, filesystem isolation) come together into a cohesive system. Think of this as the blueprint for a **container factory** with specialized assembly stations: each component handles one aspect of isolation, and the runtime orchestrates them in the correct sequence to produce fully isolated container environments.

Component Overview and Relationships

Imagine our container runtime as a **specialized manufacturing assembly line** for creating isolated process environments. Each workstation in this factory has a specific job:

1. **The Design Office (CLI)** - Takes customer orders (commands) and translates them into work orders
2. **The Factory Manager (Container Manager)** - Coordinates all workstations and ensures the assembly line flows correctly
3. **The Privacy Room Constructor (Namespace Manager)** - Builds private rooms with mirrored walls (isolated views of system resources)
4. **The Resource Meter (cgroup Manager)** - Installs smart meters that limit electricity, water, and occupancy in each room
5. **The Furniture Installer (Filesystem Manager)** - Delivers and arranges customized furniture sets (root filesystems)
6. **The Telephone System Technician (Network Manager)** - Sets up private phone lines with switchboards and external connections
7. **The Inventory and Parts Department (Image Handler)** - Manages prefabricated furniture kits (container images) and their assembly instructions

Each component has a single responsibility and communicates through well-defined interfaces managed by the Container Manager. The overall architecture follows a **coordinator pattern**: the Container Manager orchestrates all other components but delegates specific technical work to specialized managers.



Core Components and Their Responsibilities

Component	Primary Responsibility	Key Data Owned	Collaborates With
CLI (Command Line Interface)	User interaction, command parsing, output formatting	Command arguments, flags	Container Manager (via API)
Container Manager	Container lifecycle orchestration, state management	`Container` registry, state transitions	All managers (delegates work)
Namespace Manager	Linux namespace creation and isolation	Namespace file descriptors, isolation flags	Container Manager, Network Manager
cgroup Manager	Resource limit enforcement via control groups	cgroup hierarchy paths, limit configurations	Container Manager, OS kernel
Filesystem Manager	Root filesystem setup, layer management, mount operations	Rootfs paths, layer directories, mount points	Image Handler, Container Manager
Network Manager	Network namespace configuration, virtual networking	veth pairs, bridge interfaces, IP allocations	Namespace Manager, Container Manager
Image Handler	OCI image downloading, parsing, layer extraction	Image cache, layer manifests, configuration	Filesystem Manager, external registries

Component Communication Patterns

The system uses two primary communication patterns:

1. **Orchestration Flow (Top-Down)**: The Container Manager receives requests from the CLI and sequentially invokes specialized managers in a specific order:

```
CLI → Container Manager → Namespace Manager → cgroup Manager →
Filesystem Manager → Network Manager → Process Execution
```

2. **Data Flow (Configuration Passing)**: Each component receives a subset of the `ContainerConfig` relevant to its domain:
   - `NamespaceManager` receives namespace flags
   - `CgroupManager` receives `ResourceLimits`
   - `FilesystemManager` receives `Image` and mount specifications
   - `NetworkManager` receives `NetworkConfig`

> **Architectural Insight**: The Container Manager acts as a **facade** over the complex Linux kernel interfaces. Users and the CLI interact with a simple container abstraction while the manager translates these operations into the appropriate sequence of low-level system calls and configurations.

Key Architectural Decisions

> **Decision: Coordinator Pattern Over Microservices**
>
> - **Context**: We need to manage multiple Linux isolation primitives that must be applied in a specific sequence during container creation. The system runs on a single host and doesn't require distributed coordination.
> - **Options Considered**:
>   1. **Microservices architecture**: Each manager runs as a separate process with IPC
>   2. **Monolithic runtime**: All logic in a single binary with internal modules
>   3. **Coordinator pattern**: Main orchestrator with pluggable manager components
> - **Decision**: Coordinator pattern with in-process managers
> - **Rationale**:
>   - Microservices add unnecessary IPC overhead for operations that require tight sequencing
>   - Monolithic design makes testing and replacement of components difficult
>   - Coordinator pattern provides clear separation of concerns while maintaining performance
>   - In-process communication avoids serialization overhead for configuration passing
> - **Consequences**:
>   - Easier to test components in isolation via mock interfaces
>   - Can swap implementations (e.g., cgroup v1 vs v2) without changing the coordinator
>   - All components share the same memory space and life cycle
>   - Single point of failure (if the coordinator crashes, all operations halt)

Architecture Option	Pros	Cons	Why Chosen?
Microservices	Fault isolation, independent scaling, language polyglot	IPC overhead, complex deployment, sequencing difficulty	Overkill for single-host container runtime
Monolithic	Performance, simplicity, no IPC	Hard to test, tight coupling, difficult to extend	Too rigid for educational project with clear component boundaries
Coordinator Pattern	Clear separation, testable components, flexible implementations	Still single process, coordinator can become complex	**CHOSEN**: Balances separation with practical sequencing needs

Decision: Immutable Container Configuration

- **Context**: Container properties (resource limits, network setup) must be established before process execution and shouldn't change during runtime for predictability and security.
- **Options Considered**:
  1. **Mutable configuration**: Allow runtime modifications to limits, network, etc.
  2. **Immutable configuration**: Freeze configuration after container creation
- **Decision**: Immutable `ContainerConfig` after container creation
- **Rationale**:
  - Changing cgroup limits during runtime can cause unpredictable behavior
  - Network reconfiguration while processes are running is complex and error-prone
  - Matches Docker/OCI runtime behavior (config changes require container restart)
  - Simplifies state management and validation
- **Consequences**:
  - Users must recreate containers to change most settings
  - Runtime code doesn't need to handle reconfiguration edge cases
  - Clearer audit trail of container properties

System Entry Points and Control Flow

The system has three primary entry points that users interact with:

1. **Container Lifecycle Commands**: `create`, `start`, `stop`, `remove`
2. **Image Management Commands**: `pull`, `images`, `rmi`
3. **System Inspection Commands**: `ps`, `inspect`, `logs`

Each command follows a similar flow through the architecture:

1. **CLI Parsing**: User command → flag parsing → validation
2. **Request Routing**: CLI invokes appropriate method on `ContainerRuntime`
3. **Orchestration**: `ContainerManager` sequences the required operations
4. **Specialist Execution**: Specialized managers perform their isolated tasks
5. **State Management**: Results are recorded in the container registry
6. **Response**: Status/errors returned to user

Data Persistence and State Management

The runtime maintains two primary forms of persistent state:

State Type	Storage Location	Purpose	Managed By
Container Metadata	JSON files in `/var/lib/container-runtime/containers/`	Track container state, configuration, PID	`ContainerManager`
Image Layers	Directory structure in `/var/lib/container-runtime/images/`	Cache downloaded image layers for reuse	`ImageHandler`
cgroup Hierarchies	Virtual filesystem at `/sys/fs/cgroup/`	Enforce resource limits across restarts	`cgroupManager` (with OS)
Network Namespaces	Persistent references in `/var/run/netns/`	Maintain network isolation across processes	`NetworkManager`

Critical Design Insight: The `ContainerManager` serves as the **system of record** for container state. All other managers are stateless relative to container lifecycle—they apply configurations when requested but don't track what they've applied. This centralization simplifies recovery and cleanup.

Recommended File/Module Structure

Organizing the codebase clearly from the start is crucial for managing the complexity of interacting with multiple Linux kernel subsystems. Think of this structure as **department offices in our factory**—each team (component) has its own workspace with clear boundaries and defined interfaces for

collaboration.

Project Root Layout

```
build-your-own-docker/
├── cmd/
│   └── containerd/                 # Main CLI binary (named to avoid confusion with actual Docker)
│       └── main.go                 # Entry point: CLI parsing and routing
├── internal/                       # Private application code (not for external use)
│   ├── runtime/                    # Container Manager and core orchestration
│   │   ├── container.go            # Container struct and state management
│   │   ├── manager.go              # ContainerManager implementation
│   │   ├── runtime.go              # ContainerRuntime interface implementation
│   │   └── registry.go             # Container storage and retrieval
│   ├── namespaces/                 # Namespace Manager
│   │   ├── manager.go              # NamespaceManager implementation
│   │   ├── flags.go                # Namespace flag constants and helpers
│   │   └── unix.go                 # Low-level syscall wrappers
│   ├── cgroups/                    # cgroup Manager
│   │   ├── manager.go              # CgroupManager implementation
│   │   ├── v1/                     # cgroup v1 implementation
│   │   │   └── controller.go       # v1-specific controller logic
│   │   ├── v2/                     # cgroup v2 implementation
│   │   │   └── controller.go       # v2-specific controller logic
│   │   └── common.go               # Shared cgroup utilities
│   ├── filesystem/                 # Filesystem Manager
│   │   ├── manager.go              # FilesystemManager implementation
│   │   ├── rootfs.go               # Root filesystem setup (pivot_root)
│   │   ├── overlay.go              # OverlayFS mounting and management
│   │   └── mounts.go               /proc, /sys, /dev mounting
│   ├── network/                    # Network Manager
│   │   ├── manager.go              # NetworkManager implementation
│   │   ├── veth.go                 # Virtual ethernet pair creation
│   │   ├── bridge.go               # Linux bridge setup and management
│   │   ├── iptables.go             # NAT and firewall rules
│   │   └── dns.go                  # DNS/resolv.conf configuration
│   └── image/                      # Image Handler
│       ├── manager.go              # ImageHandler implementation
│       ├── oci/                    # OCI specification handling
│       │   ├── manifest.go         # Manifest parsing and validation
│       │   └── config.go           # Image config parsing
│       ├── registry/               # Registry client
│       │   ├── client.go           # HTTP client for Docker Registry API
│       │   └── auth.go             # Authentication handling
│       └── storage.go              # Layer storage and extraction
├── pkg/                            # Public libraries (potentially reusable)
│   ├── spec/                       # OCI specification types
│   │   ├── container.go            # Container runtime spec types
│   │   └── image.go                # Image spec types
│   └── utils/                      # General utilities
│       ├── exec.go                 # Process execution helpers
│       └── signal.go               # Signal handling utilities
├── var/                            # Runtime data (created at runtime)
│   ├── lib/container-runtime/      # Persistent data
│   │   ├── containers/             # Container state JSON files
│   │   ├── images/                 # Extracted image layers
│   │   └── overlay/                # OverlayFS work and upper directories
│   └── run/container-runtime/      # Ephemeral runtime data
│       └── netns/                  # Network namespace bind mounts
├── scripts/                        # Build and development scripts
│   ├── setup.sh                    # Environment setup
│   └── test.sh                     # Test runner
├── examples/                       # Example configurations and usage
│   ├── simple-container/           # Minimal container example
│   └── custom-network/             # Network configuration example
├── go.mod                          # Go module definition
└── README.md                       # Project documentation
```

Module Dependencies and Import Relationships

The dependency flow follows the orchestration pattern: higher-level modules import lower-level ones, but not vice versa.

```
cmd/containerd/ → internal/runtime/ → [all other internal/ components]
                                ↳ internal/namespaces/
                                ↳ internal/cgroups/
                                ↳ internal/filesystem/
                                ↳ internal/network/
                                ↳ internal/image/
```

Key Dependency Rules:

1. **Horizontal dependencies are forbidden**: `internal/namespaces/` cannot import `internal/network/`
2. **Upward dependencies are forbidden**: No internal component imports `cmd/`
3. **Shared types live in `pkg/spec/`**: All managers import common types from here
4. **Utilities are shared via `pkg/utils/`**: Common helpers used by multiple components

Component Interface Definitions

Each manager component implements a well-defined interface that the `ContainerManager` uses:

Interface	File Location	Key Methods	Used By
`NamespaceManager`	`internal/namespaces/manager.go`	`CreateNamespaces(flags int) (int, error)`	`ContainerManager` during container creation
`CgroupManager`	`internal/cgroups/manager.go`	`CreateGroup(path string) error`, `SetLimits(path string, limits ResourceLimits) error`	`ContainerManager` after namespace creation
`FilesystemManager`	`internal/filesystem/manager.go`	`SetupRootfs(image Image, containerID string) (string, error)`	`ContainerManager` before process execution
`NetworkManager`	`internal/network/manager.go`	`SetupNetwork(nsPath string, config NetworkConfig) error`	`ContainerManager` after filesystem setup
`ImageHandler`	`internal/image/manager.go`	`Pull(imageName string) (Image, error)`, `Extract(image Image) error`	`ContainerManager` and `FilesystemManager`

> **Design Principle**: Each interface represents a **capability** not an implementation. This allows us to swap implementations (e.g., cgroup v1 vs v2) without changing the orchestration logic.

State Directory Structure

At runtime, the system creates and manages the following directory structure under `/var/lib/container-runtime/`:

```
/var/lib/container-runtime/
├── containers/                  # One subdirectory per container
│   ├── {container-id}/          # Container-specific data
│   │   ├── config.json          # ContainerConfig serialized
│   │   ├── state.json           # Current ContainerState, PID, etc.
│   │   └── rootfs/              # Container's root filesystem (if not using overlay)
│   └── metadata.db              # Optional: SQLite for faster lookups
├── images/                      # Extracted image layers
│   ├── {image-digest}/          # One directory per image layer
│   │   ├── layer.tar            # Original layer archive
│   │   └── fs/                  # Extracted files
│   └── manifests/               # Downloaded image manifests
└── overlay/                     # OverlayFS directories
    ├── {container-id}/          # Per-container overlay directories
    │   ├── upper/               # Writable layer (changes)
    │   ├── work/                # Work directory for OverlayFS
    │   └── merged/              # Unified view (actual rootfs)
    └── layers/                  # Shared read-only lower layers
```

This structure ensures:

- **Isolation**: Each container's data is separate
- **Persistence**: Container state survives runtime restarts
- **Sharing**: Image layers are shared between containers

- **Cleanup**: Removing a container deletes its directory entirely

Implementation Guidance

> **Transition Note**: The following implementation guidance bridges the gap between architectural design and actual code. It provides concrete starting points for implementing the high-level structure described above.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option	Recommendation for Learning
CLI Framework	Standard `flag` package	`cobra` or `urfave/cli`	**Standard `flag`** (minimal dependencies, focuses on concepts)
Configuration Storage	JSON files in directories	SQLite database	**JSON files** (transparent, easy to debug, matches OCI spec)
Image Registry Client	Basic HTTP with `net/http`	Full OCI distribution client	**Basic HTTP** with authentication support (teaches registry API)
Process Management	`os/exec` package	Direct syscall wrappers	**Mix**: `os/exec` for orchestration, syscalls for isolation
File System Operations	`os` and `io` packages	Lower-level syscalls	**Standard library** (safer, portable across Unix-like systems)
Networking Setup	`net` package + `exec` for `ip` commands	Direct netlink socket calls	**`exec` for `ip` and `iptables`** (simpler, mirrors actual admin work)

B. Recommended File/Module Structure (Starter Implementation)

Let's begin with the minimal viable structure to get started. Create these files first:

```
build-your-own-docker/
├── cmd/
│   └── byod/                    # "Build Your Own Docker" CLI
│       └── main.go              # Start here - basic CLI skeleton
├── internal/
│   └── runtime/
│       ├── container.go         # Core data structures
│       └── manager.go           # ContainerManager skeleton
├── pkg/
│   └── spec/
│       └── types.go             # All type definitions
└── go.mod                       # Initialize with: go mod init build-your-own-docker
```

C. Infrastructure Starter Code

File: `pkg/spec/types.go` - Complete type definitions matching the naming conventions:

GO

```go
// Package spec defines the core data types for the container runtime.
package spec

import (
    "time"
)

// ContainerState represents the lifecycle state of a container.
type ContainerState string

const (
    StateCreated ContainerState = "created"
    StateRunning ContainerState = "running"
    StatePaused  ContainerState = "paused"
    StateStopped ContainerState = "stopped"
    StateRemoved ContainerState = "removed"
)

// ResourceLimits defines resource constraints for a container.
type ResourceLimits struct {
    MemoryMB  int // Memory limit in megabytes
    CPUShares int // CPU shares (relative weight)
    PidsLimit int // Maximum number of processes
}

// PortMapping defines a port forwarding rule.
type PortMapping struct {
    HostPort      int    // Port on the host
    ContainerPort int    // Port in the container
    Protocol      string // "tcp" or "udp"
}

// NetworkConfig defines network settings for a container.
type NetworkConfig struct {
    Mode         string       // "bridge", "host", "none"
    BridgeName   string       // Name of the bridge interface
    IPAddress    string       // Container IP address (optional)
    PortMappings []PortMapping // Port forwarding rules
}

// ContainerConfig defines the configuration for creating a container.
type ContainerConfig struct {
```

```
    Image      string         // Image name or ID
    Cmd        []string       // Command to run (overrides image entrypoint)
    Env        []string       // Environment variables
    WorkingDir string         // Working directory inside container
    Hostname   string         // Container hostname
    Limits     ResourceLimits // Resource limits
    Network    NetworkConfig  // Network configuration
}

// Container represents a running or stopped container.
type Container struct {
    ID        string          // Unique container identifier
    Name      string          // Human-readable name
    State     ContainerState  // Current state
    Config    ContainerConfig // Creation configuration
    Pid       int             // Process ID of container process (if running)
    CreatedAt time.Time       // Creation timestamp
}

// ImageLayer represents a single filesystem layer in an image.
type ImageLayer struct {
    Digest string // SHA256 digest of layer content
    Size   int64  // Size in bytes
    Path   string // Local filesystem path to extracted layer
}

// ImageConfig defines the runtime configuration from an image.
type ImageConfig struct {
    Entrypoint []string // Entrypoint command
    Cmd        []string // Default arguments to entrypoint
    Env        []string // Default environment variables
    WorkingDir string   // Default working directory
}

// Image represents a container image.
type Image struct {
    Name   string        // Image name (e.g., "alpine:latest")
    Digest string        // SHA256 digest of image manifest
    Layers []ImageLayer  // Filesystem layers
    Config ImageConfig   // Image configuration
```

```
}
```

File: `cmd/byod/main.go` - Basic CLI skeleton:

```go
// Package main implements the Build Your Own Docker CLI.
package main

import (
    "flag"
    "fmt"
    "log"
    "os"

    "build-your-own-docker/internal/runtime"
)

func main() {
    // Initialize the container runtime
    rt := runtime.NewContainerRuntime()

    // Parse command line arguments
    if len(os.Args) < 2 {
        printUsage()
        os.Exit(1)
    }

    command := os.Args[1]
    args := os.Args[2:]

    switch command {
    case "run":
        runCommand(rt, args)
    case "create":
        createCommand(rt, args)
    case "start":
        startCommand(rt, args)
    case "stop":
        stopCommand(rt, args)
    case "rm":
        rmCommand(rt, args)
    case "ps":
        psCommand(rt, args)
    case "pull":
```

```
        pullCommand(rt, args)
    case "images":
        imagesCommand(rt, args)
    default:
        fmt.Printf("Unknown command: %s\n", command)
        printUsage()
        os.Exit(1)
    }
}

func printUsage() {
    fmt.Println(`Build Your Own Docker - A minimal container runtime

Usage:
  byod run [OPTIONS] IMAGE [COMMAND] [ARG...]
  byod create [OPTIONS] IMAGE [COMMAND] [ARG...]
  byod start [OPTIONS] CONTAINER
  byod stop [OPTIONS] CONTAINER
  byod rm [OPTIONS] CONTAINER
  byod ps [OPTIONS]
  byod pull [OPTIONS] IMAGE
  byod images [OPTIONS]

Examples:
  byod run alpine:latest /bin/sh
  byod ps
  byod pull ubuntu:20.04`)
}

// TODO: Implement individual command handlers
func runCommand(rt runtime.ContainerRuntime, args []string) {
    log.Fatal("run command not yet implemented")
}

func createCommand(rt runtime.ContainerRuntime, args []string) {
    log.Fatal("create command not yet implemented")
}

func startCommand(rt runtime.ContainerRuntime, args []string) {
    log.Fatal("start command not yet implemented")
}
```

```go
func stopCommand(rt runtime.ContainerRuntime, args []string) {
    log.Fatal("stop command not yet implemented")
}

func rmCommand(rt runtime.ContainerRuntime, args []string) {
    log.Fatal("rm command not yet implemented")
}

func psCommand(rt runtime.ContainerRuntime, args []string) {
    log.Fatal("ps command not yet implemented")
}

func pullCommand(rt runtime.ContainerRuntime, args []string) {
    log.Fatal("pull command not yet implemented")
}

func imagesCommand(rt runtime.ContainerRuntime, args []string) {
    log.Fatal("images command not yet implemented")
}
```

File: `internal/runtime/container.go` - Container registry and state management:

```go
// Package runtime implements the container lifecycle manager.
package runtime

import (
    "encoding/json"
    "fmt"
    "os"
    "path/filepath"
    "sync"
    "time"

    "build-your-own-docker/pkg/spec"
)

// containerStore manages persistent container state.
type containerStore struct {
    sync.RWMutex
    baseDir string               // Directory for container state files
    containers map[string]*spec.Container // In-memory cache
}

// newContainerStore creates a new container store.
func newContainerStore(baseDir string) (*containerStore, error) {
    // Ensure base directory exists
    if err := os.MkdirAll(baseDir, 0755); err != nil {
        return nil, fmt.Errorf("failed to create container store directory: %w", err)
    }

    return &containerStore{
        baseDir:    baseDir,
        containers: make(map[string]*spec.Container),
    }, nil
}

// save writes a container's state to disk.
func (s *containerStore) save(container *spec.Container) error {
    s.Lock()
    defer s.Unlock()

    // Create container directory
```

```
    containerDir := filepath.Join(s.baseDir, container.ID)
    if err := os.MkdirAll(containerDir, 0755); err != nil {
        return fmt.Errorf("failed to create container directory: %w", err)
    }

    // Write config.json
    configPath := filepath.Join(containerDir, "config.json")
    configData, err := json.MarshalIndent(container.Config, "", "  ")
    if err != nil {
        return fmt.Errorf("failed to marshal container config: %w", err)
    }

    if err := os.WriteFile(configPath, configData, 0644); err != nil {
        return fmt.Errorf("failed to write container config: %w", err)
    }

    // Write state.json
    statePath := filepath.Join(containerDir, "state.json")
    stateData, err := json.MarshalIndent(struct {
        ID        string
        Name      string
        State     spec.ContainerState
        Pid       int
        CreatedAt time.Time
    }{
        ID:        container.ID,
        Name:      container.Name,
        State:     container.State,
        Pid:       container.Pid,
        CreatedAt: container.CreatedAt,
    }, "", "  ")

    if err != nil {
        return fmt.Errorf("failed to marshal container state: %w", err)
    }

    if err := os.WriteFile(statePath, stateData, 0644); err != nil {
        return fmt.Errorf("failed to write container state: %w", err)
```

```
    }

    // Update in-memory cache
    s.containers[container.ID] = container

    return nil
}

// load reads all containers from disk into memory.
func (s *containerStore) load() error {
    s.Lock()
    defer s.Unlock()

    entries, err := os.ReadDir(s.baseDir)
    if err != nil {
        return fmt.Errorf("failed to read container store directory: %w", err)
    }

    for _, entry := range entries {
        if !entry.IsDir() {
            continue
        }

        containerID := entry.Name()
        containerDir := filepath.Join(s.baseDir, containerID)

        // Load config
        configPath := filepath.Join(containerDir, "config.json")
        configData, err := os.ReadFile(configPath)
        if err != nil {
            // Skip corrupt containers
            continue
        }

        var config spec.ContainerConfig
        if err := json.Unmarshal(configData, &config); err != nil {
            continue
        }
```

```
        // Load state
        statePath := filepath.Join(containerDir, "state.json")
        stateData, err := os.ReadFile(statePath)
        if err != nil {
            continue
        }

        var state struct {
            ID        string
            Name      string
            State     spec.ContainerState
            Pid       int
            CreatedAt time.Time
        }

        if err := json.Unmarshal(stateData, &state); err != nil {
            continue
        }

        container := &spec.Container{
            ID:        state.ID,
            Name:      state.Name,
            State:     state.State,
            Config:    config,
            Pid:       state.Pid,
            CreatedAt: state.CreatedAt,
        }

        s.containers[container.ID] = container
    }

    return nil
}

// get retrieves a container by ID.
func (s *containerStore) get(id string) (*spec.Container, error) {
    s.RLock()
    defer s.RUnlock()
```

```
    container, exists := s.containers[id]
    if !exists {
        return nil, fmt.Errorf("container %s not found", id)
    }

    return container, nil
}

// list returns all containers.
func (s *containerStore) list() []*spec.Container {
    s.RLock()
    defer s.RUnlock()

    containers := make([]*spec.Container, 0, len(s.containers))
    for _, container := range s.containers {
        containers = make([]*spec.Container, 0, len(s.containers))
        for _, container := range s.containers {
            containers = append(containers, container)
        }
    }

    return containers
}

// remove deletes a container from the store.
func (s *containerStore) remove(id string) error {
    s.Lock()
    defer s.Unlock()

    // Delete from filesystem
    containerDir := filepath.Join(s.baseDir, id)
    if err := os.RemoveAll(containerDir); err != nil {
        return fmt.Errorf("failed to remove container directory: %w", err)
    }

    // Delete from memory
    delete(s.containers, id)

    return nil
```

```
}
```

D. Core Logic Skeleton Code

File: `internal/runtime/manager.go` - ContainerManager skeleton with detailed TODOs:

```go
package runtime


import (

    "fmt"

    "math/rand"

    "time"



    "build-your-own-docker/pkg/spec"

)


// ContainerRuntime defines the interface for container operations.

type ContainerRuntime interface {

    CreateContainer(config spec.ContainerConfig) (spec.Container, error)

    StartContainer(id string) error

    StopContainer(id string) error

    RemoveContainer(id string) error

    ListContainers() ([]spec.Container, error)

}


// ContainerManager implements ContainerRuntime.

type ContainerManager struct {

    store *containerStore

    // TODO: Add references to specialized managers as fields

    // namespaceMgr namespaces.NamespaceManager

    // cgroupMgr    cgroups.CgroupManager

    // fsMgr        filesystem.FilesystemManager

    // networkMgr   network.NetworkManager

    // imageMgr     image.ImageHandler

}


// NewContainerRuntime creates a new container runtime instance.

func NewContainerRuntime() ContainerRuntime {

    // TODO 1: Initialize container store with default path (/var/lib/container-runtime)

    // TODO 2: Load existing containers from disk

    // TODO 3: Initialize specialized managers (nil for now)



    return &ContainerManager{

        store: nil, // Will be initialized in TODO 1

    }

}
```

```go
// CreateContainer creates a new container with given configuration.
func (m *ContainerManager) CreateContainer(config spec.ContainerConfig) (spec.Container, error) {
    // TODO 1: Generate unique container ID (e.g., random 12-character hex string)
    // TODO 2: Create Container struct with initial state (StateCreated)
    // TODO 3: Validate configuration (check image exists, valid resource limits, etc.)
    // TODO 4: Pull image if not already available locally
    // TODO 5: Setup container directory in store
    // TODO 6: Save container to store
    // TODO 7: Return the created container

    return spec.Container{}, fmt.Errorf("not implemented")
}

// StartContainer starts a created container.
func (m *ContainerManager) StartContainer(id string) error {
    // TODO 1: Retrieve container from store by ID
    // TODO 2: Validate container is in StateCreated state
    // TODO 3: Create Linux namespaces (PID, UTS, mount, network, IPC)
    // TODO 4: Create and configure cgroup for container
    // TODO 5: Setup root filesystem using image layers
    // TODO 6: Configure network namespace (veth pair, bridge, IP, etc.)
    // TODO 7: Execute the container process inside the namespaces
    // TODO 8: Update container state to StateRunning and save PID
    // TODO 9: Monitor container process (in goroutine) to detect exit

    return fmt.Errorf("not implemented")
}

// StopContainer stops a running container.
func (m *ContainerManager) StopContainer(id string) error {
    // TODO 1: Retrieve container from store by ID
    // TODO 2: Validate container is in StateRunning state
    // TODO 3: Send SIGTERM to container process (PID)
    // TODO 4: Wait for process termination with timeout
    // TODO 5: If timeout, send SIGKILL
    // TODO 6: Cleanup network namespace
    // TODO 7: Remove cgroup
    // TODO 8: Update container state to StateStopped
    // TODO 9: Save updated state to store
```

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

// RemoveContainer removes a stopped container.
func (m *ContainerManager) RemoveContainer(id string) error {
    // TODO 1: Retrieve container from store by ID
    // TODO 2: Validate container is in StateStopped state
    // TODO 3: Remove container root filesystem (if not using overlay)
    // TODO 4: Remove container directory from store
    // TODO 5: Update container state to StateRemoved in memory (optional)
    // TODO 6: Return success

    return fmt.Errorf("not implemented")
}

// ListContainers returns all containers.
func (m *ContainerManager) ListContainers() ([]spec.Container, error) {
    // TODO 1: Retrieve all containers from store
    // TODO 2: Convert to slice of spec.Container
    // TODO 3: Return containers

    return nil, fmt.Errorf("not implemented")
}

// generateID creates a unique container ID.
func generateID() string {
    const letters = "abcdef0123456789"
    const idLength = 12

    rand.Seed(time.Now().UnixNano())
    b := make([]byte, idLength)
    for i := range b {
        b[i] = letters[rand.Intn(len(letters))]
    }
    return string(b)
}
```

E. Language-Specific Hints

Go-Specific Implementation Tips:

1. **Syscall vs. Exec**: Use the `syscall` package for namespace creation (`unshare` , `setns` , `clone`), but prefer `exec.Command` for configuring network (`ip` , `iptables`) to avoid reimplementing complex logic.
2. **Process Management**: Use `os.StartProcess` instead of `os/exec` when you need fine-grained control over process attributes (like clone flags for namespaces).
3. **File Operations**: Always use absolute paths when dealing with mount operations and namespace references to avoid confusion about which namespace you're operating in.
4. **Error Handling**: Use `fmt.Errorf("... %w", err)` to wrap errors with context, especially for syscalls where the raw error message might be unclear.
5. **Concurrency**: Use `sync.RWMutex` for the container store since multiple goroutines might read container state while only one modifies it.
6. **JSON Serialization**: Use `json.MarshalIndent` for debugging but regular `json.Marshal` for production to save space.

F. Milestone Checkpoint

After implementing the basic structure above, verify your setup:

1. **Build the project**:

```
go build ./cmd/byod
```
BASH

2. **Run the CLI without arguments** to see usage:

```
./byod
```
BASH

Expected output: The usage message defined in `printUsage()`

3. **Test each command stub**:

```
./byod run alpine:latest /bin/sh
```
BASH

Expected: "run command not yet implemented" error

4. **Check that types compile correctly**:

```
go test ./pkg/spec/...
```
BASH

Expected: No compilation errors (even with no tests)

Signs of correct setup:

- Binary compiles without errors
- Each command shows appropriate "not implemented" message
- Directory structure matches the blueprint
- `go mod tidy` runs without dependency issues

Common setup problems:

- **"cannot find module"**: Run `go mod init build-your-own-docker` in project root
- **Import path errors**: Ensure `go.mod` module name matches import paths
- **Permission errors**: You'll need root/sudo for later milestones but not for this structure

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
"Command not found" when running ./byod	File not executable or wrong path	Run `ls -la ./byod`	`chmod +x ./byod` or use `go run cmd/byod/main.go`
Import errors in Go files	Incorrect module name or missing dependencies	Check `go.mod` first line matches import paths	Update `go.mod` or fix import statements
JSON marshaling fails	Struct fields not exported (capitalized)	Check field names start with uppercase	Export fields: `Name` not `name`
Store directory not created	Permission issues or parent doesn't exist	Check `os.MkdirAll` error return	Run with appropriate permissions or create parent directory

---

Data Model

> **Milestone(s):** All milestones (this section defines the core data structures that every component will use to represent containers, images, and runtime state)

This section defines the foundational data structures that our container runtime uses to represent the state of the system. Think of these structures as **blueprints and records** in a construction project: the blueprints (`ContainerConfig` , `Image`) specify what to build, while the records (`Container` , runtime state) track what's actually been built and its current condition. These data models serve as the single source of truth that coordinates all our components—from namespace isolation to network configuration.

Key Types and Structures

The container runtime manipulates several core entity types, each with specific responsibilities and lifecycle. These structures capture everything from container configuration and runtime state to image metadata and network settings.

Container Lifecycle State (`ContainerState`)

Containers transition through a well-defined lifecycle, much like processes in an operating system. The `ContainerState` enumeration tracks where a container is in this lifecycle, enabling proper management operations and preventing invalid state transitions (like trying to start an already running container).

State	Description	Allowed Next States
`Created`	Container configuration has been validated and persisted, but the isolated process hasn't started yet. Namespaces, cgroups, and filesystems may be partially prepared.	`Running` , `Stopped` (if cleanup fails)
`Running`	Container process is actively executing inside its isolated environment. All resources (namespaces, cgroups, network) are active and being used.	`Paused` , `Stopped`
`Paused`	Container process is suspended (via cgroup freezer). The process exists but isn't scheduled for CPU time. Memory and other resources remain allocated.	`Running` (resumed), `Stopped`
`Stopped`	Container process has terminated (either normally or via signal). Resources are still allocated but can be cleaned up. This is a terminal state before removal.	`Removed`
`Removed`	All container resources have been released (namespaces destroyed, cgroups deleted, filesystems unmounted). Only metadata may remain for historical tracking.	(terminal state)

The state machine ensures that operations like `StartContainer` can only be called on `Created` containers, while `RemoveContainer` requires a `Stopped` container, preventing resource leaks or inconsistent states.

Container Configuration (`ContainerConfig`)

The `ContainerConfig` structure serves as the **recipe** for creating a container—it specifies what to run, with what resources, and in what environment. This is the primary input from users (via CLI or API) and contains all the settings needed to create an isolated environment.

Field	Type	Description
`Image`	`string`	Reference to the container image (e.g., "alpine:latest" or "ubuntu:22.04"). This identifies which root filesystem and default configuration to use.
`Cmd`	`[]string`	Command and arguments to execute inside the container (overrides the image's default `Cmd`). Example: `["/bin/sh", "-c", "echo hello"]`.
`Env`	`[]string`	Environment variables in "KEY=VALUE" format that will be available to the container process. These are merged with (and can override) the image's environment variables.
`WorkingDir`	`string`	Initial working directory for the container process. If empty, defaults to the image's `WorkingDir` or root ("/").
`Hostname`	`string`	Hostname to set inside the container's UTS namespace. If empty, defaults to the container ID or a generated name.
`Limits`	`ResourceLimits`	Resource constraints (CPU, memory, process count) to apply via cgroups. Zero values typically mean "no limit."
`Network`	`NetworkConfig`	Network isolation and connectivity settings, including IP address, port mappings, and network mode.

This configuration is **immutable** after container creation—runtime changes to resource limits or network settings would require complex live updates and are beyond our scope.

Resource Limits (`ResourceLimits`)

Resource limits enforce **quotas** on container resource consumption, preventing any single container from monopolizing system resources. Think of these as utility caps in an apartment building—each unit gets a maximum allocation of water, electricity, and gas.

Field	Type	Description
`MemoryMB`	`int`	Maximum resident memory (RAM) in megabytes that the container can use. When exceeded, the OOM killer terminates container processes.
`CPUShares`	`int`	Relative CPU weight (in the cgroups v1 "cpu.shares" scheme). Default is 1024; values are proportional to other containers' shares.
`PidsLimit`	`int`	Maximum number of processes (including threads) the container can create. Prevents fork bombs and process exhaustion attacks.

> **Design Insight:** We use simple integer fields rather than more complex types (like `uint64` or custom units) to keep the API straightforward for learners. In production systems, you'd want finer-grained controls (CPU quotas, memory+swap limits, I/O bandwidth), but these three limits cover the essential isolation concerns.

Network Configuration (`NetworkConfig` and `PortMapping`)

Network configuration defines how the container connects to networks—whether it's fully isolated, bridged to a host network, or uses host networking directly. The `PortMapping` structure enables **port forwarding**, similar to how a hotel switchboard directs external calls to specific room extensions.

`NetworkConfig` structure:

Field	Type	Description
`Mode`	`string`	Network isolation mode: "bridge" (default, private network with NAT), "host" (share host's network namespace), "none" (no network interfaces).
`BridgeName`	`string`	Name of the Linux bridge to connect to (for "bridge" mode). Defaults to "byob0" (Build Your Own Bridge).
`IPAddress`	`string`	Static IPv4 address to assign to the container (e.g., "172.17.0.2"). If empty, uses DHCP or automatic allocation from bridge subnet.
`PortMappings`	`[]PortMapping`	List of port forwardings from host to container (e.g., host port 8080 → container port 80).

`PortMapping` structure:

Field	Type	Description
`HostPort`	`int`	Port number on the host interface (0-65535). Ports below 1024 typically require root privileges.
`ContainerPort`	`int`	Port number inside the container network namespace (0-65535).
`Protocol`	`string`	Transport protocol: "tcp" or "udp". Determines which iptables rule chain to modify.

> **Architecture Decision: Network Configuration Simplicity**
>
> **Context:** We need to provide network connectivity options for containers while keeping implementation complexity manageable for learners.
>
> **Options Considered:**
>
> 1. **Full OCI network specification** with CNI plugins, multiple network interfaces, DNS configurations
> 2. **Simple bridge-only model** with static IP assignment and basic port forwarding
> 3. **Host-only networking** (simplest, but no isolation)
>
> **Decision:** Option 2 (simple bridge model) with extensible `Mode` field.
>
> **Rationale:** Bridge networking provides practical isolation while being implementable with standard Linux tools (ip, brctl, iptables). The `Mode` field allows future extension to "host" and "none" modes without breaking existing configurations. We avoid CNI complexity as it's a significant learning curve beyond core container concepts.
>
> **Consequences:** Containers get reasonable network isolation out-of-the-box, but advanced networking features (multiple networks, custom DNS, IPv6) would require significant extensions to the data model and implementation.

Container Runtime Instance (`Container`)

The `Container` structure represents an **actual running or runnable instance**—it's the runtime manifestation of a `ContainerConfig`. Think of it as the difference between an architectural blueprint (`ContainerConfig`) and the actual constructed building (`Container`).

Field	Type	Description
`ID`	`string`	Unique identifier (typically a random 64-bit hex string). Used to reference the container in all operations.
`Name`	`string`	Human-readable name for the container (e.g., "my-web-server"). Must be unique across running containers.
`State`	`ContainerState`	Current lifecycle state (Created, Running, etc.). Determines which operations are valid.
`Config`	`ContainerConfig`	Immutable configuration used to create this container. Stored for reference and potential restart.
`Pid`	`int`	Process ID of the container's init process on the host. Zero if container isn't running. Used to manage cgroup membership and send signals.
`CreatedAt`	`time.Time`	Timestamp when the container was first created. Used for display (e.g., `docker ps`) and cleanup of old containers.

The `Container` object is persisted to disk (typically as JSON) so the runtime can recover state after a restart. This is critical for operations like `StopContainer` to work—the runtime needs to know which host PID to signal.

Image Metadata (`Image` and `ImageLayer`)

Images are **immutable templates** for container filesystems. An image consists of multiple layers (like a layered cake) that are stacked together using OverlayFS. The `Image` structure represents a downloaded and extracted image, ready to be instantiated into containers.

`Image` structure:

Field	Type	Description
`Name`	`string`	Image reference including registry, repository, and tag (e.g., "docker.io/library/alpine:latest").
`Digest`	`string`	Cryptographic hash of the image manifest (SHA256). Uniquely identifies this exact image version.
`Layers`	`[]ImageLayer`	Ordered list of filesystem layers (bottom to top) that make up the image's root filesystem.
`Config`	`ImageConfig`	Default runtime configuration from the image (entrypoint, environment, working directory).

`ImageLayer` structure:

Field	Type	Description
`Digest`	`string`	Content-addressable hash (SHA256) of the layer tarball. Used for caching and verification.
`Size`	`int64`	Size of the layer tarball in bytes. Used for download progress reporting and disk space management.
`Path`	`string`	Filesystem path where this layer's extracted contents are stored (typically in a content-addressable store).

`ImageConfig` structure:

Field	Type	Description
`Entrypoint`	`[]string`	Default executable and arguments to run when container starts (e.g., `["/bin/myapp"]`).
`Cmd`	`[]string`	Default arguments to `Entrypoint` (overridden by container's `Cmd`).
`Env`	`[]string`	Default environment variables (e.g., `["PATH=/usr/bin", "HOME=/root"]`).
`WorkingDir`	`string`	Default working directory if none specified in container config.

> **Design Insight:** The separation between `Image` metadata and `ContainerConfig` allows for inheritance and override patterns. When a container is created, its `ContainerConfig` merges with the `ImageConfig`—explicit container settings override image defaults, similar to how subclass methods override parent class methods in object-oriented programming.

Relationships Between Entities

The data model entities relate to each other in specific ways that define the overall system architecture. Understanding these relationships is crucial for implementing correct lifecycle management and resource cleanup.

Container-Image Relationship

A container **instantiates** an image, much like an object instantiates a class in programming. The relationship is directional: containers reference images, but images don't know about their running instances.

```
Image (template)
    ↑ referenced by
Container (instance)
```

Key behaviors:

- Multiple containers can reference the same image (sharing layers via OverlayFS)
- Images are immutable; containers add writable layers on top
- Deleting an image fails if any containers still reference it
- Container configuration (`ContainerConfig`) overrides image defaults (`ImageConfig`)

This relationship is managed through the `ContainerConfig.Image` field, which contains the image name or digest. The runtime resolves this to an actual `Image` object during container creation.

Container State Transitions

The container lifecycle defines a **state machine** where each transition corresponds to a management operation. The following table shows all valid transitions and the operations that trigger them:

Current State	Event/Operation	Next State	Actions Performed During Transition
(nonexistent)	`CreateContainer`	`Created`	Validate config, allocate ID, persist metadata, prepare namespaces and rootfs
`Created`	`StartContainer`	`Running`	Create cgroups, apply limits, setup network, start init process
`Created`	`RemoveContainer`	`Removed`	Clean up any prepared resources (namespaces, rootfs)
`Running`	`StopContainer` (or process exits)	`Stopped`	Send termination signal, wait for exit, update state
`Running`	`PauseContainer`	`Paused`	Freeze cgroup, suspend all processes
`Paused`	`ResumeContainer`	`Running`	Unfreeze cgroup, resume processes
`Stopped`	`RemoveContainer`	`Removed`	Delete cgroups, network namespace, mounted filesystems
Any	Error during operation	`Stopped`	Clean up partially created resources, log error

> **Critical Implementation Detail:** The transition from `Stopped` to `Removed` is irreversible and must clean up **all** resources (cgroups, network interfaces, mount points). Resource leaks occur if any resource isn't properly released during this transition.

Layer Composition and Sharing

Images use a **layered composition** model where each layer builds upon previous ones. Containers then add a writable layer on top of the image stack:

```
Container Writable Layer (upperdir)
    ↑
Image Layer N (topmost read-only layer)
    ↑
Image Layer N-1
    ↑
...
    ↑
Image Layer 1 (base layer)
    ↑
Base Filesystem (scratch)
```

Sharing implications:

- Identical layers across images are stored once and referenced multiple times (content-addressable storage)
- When a container modifies a file from a read-only layer, OverlayFS performs **copy-on-write** to the writable layer
- Deleting an image only removes its unique layers; shared layers remain until all referencing images are deleted
- The runtime must track layer reference counts to implement garbage collection

This relationship is captured through the `Image.Layers` field (ordered list) and the filesystem manager's layer stacking logic.

Network Configuration Inheritance

Network settings follow a **hierarchical override** pattern where container-specific settings override network defaults:

```
Network Defaults (bridge: byob0, subnet: 172.17.0.0/16)
    ↑ can be overridden by
Container NetworkConfig
    ↑ can be overridden by
Runtime State (actual assigned IP, port mappings in iptables)
```

For example, if the container doesn't specify an `IPAddress`, the network manager assigns one from the bridge's DHCP range. If no `BridgeName` is specified, the default bridge is used. This inheritance chain ensures sensible defaults while allowing customization.

Data Persistence Relationships

Different parts of the data model have different persistence requirements:

Entity	Storage Location	Format	Lifetime
`Container` metadata	`/var/lib/byob/containers/{id}/config.json`	JSON	Until explicit removal
`Container` runtime state	In-memory + above JSON	Go struct + JSON	Runtime + persisted
`Image` metadata	`/var/lib/byob/images/{digest}/manifest.json`	JSON	Until all referencing containers removed and image deleted
`ImageLayer` contents	`/var/lib/byob/layers/{digest}/`	Extracted files	Until all referencing images removed
Network allocation state	`/var/lib/byob/network/allocations.json`	JSON	Until host reboot or network cleanup

These persistence relationships ensure the runtime can:

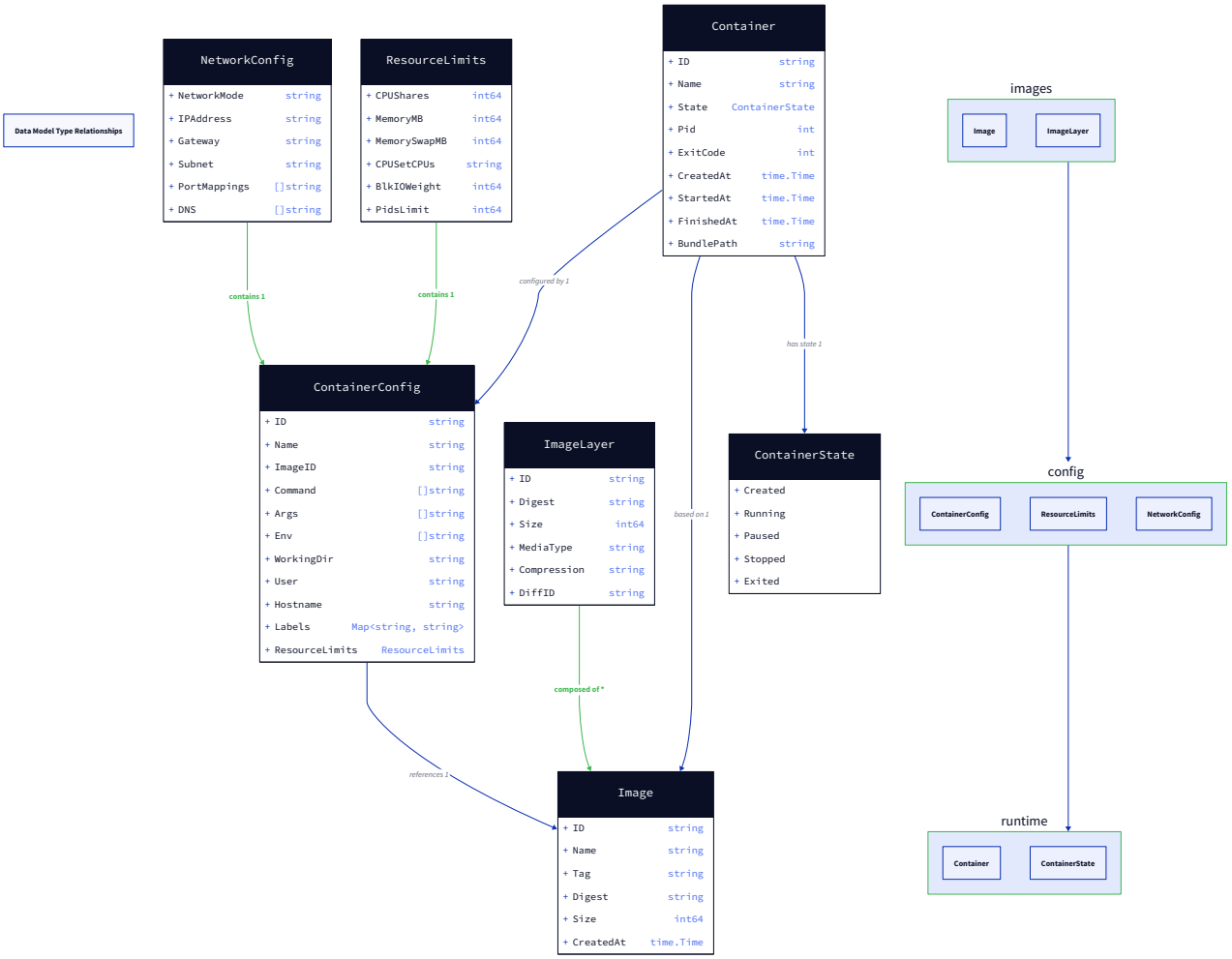
1. Recover container state after a restart (critical for managing long-running containers)
2. Share images across multiple containers efficiently
3. Avoid IP address conflicts by tracking allocations
4. Implement garbage collection for unused layers

Component Ownership Relationships

Each component in the high-level architecture owns specific parts of the data model:

Component	Owned Data	Responsibility
Container Manager	`Container` objects, state transitions	Lifecycle management, persistence
Image Handler	`Image` objects, layer storage	Image download, extraction, caching
Filesystem Manager	Layer directories, mount points	OverlayFS stacking, rootfs preparation
Network Manager	IP allocations, veth interfaces	Network namespace setup, connectivity
cgroup Manager	cgroup directories, process memberships	Resource limit enforcement

This ownership model prevents data races and ensures each component has authority over its domain. For example, only the cgroup Manager should write to cgroup control files, while only the Network Manager should modify iptables rules.



Implementation Guidance

This section provides concrete implementation starting points for the data model structures and their persistence.

Technology Recommendations Table:

Component	Simple Option	Advanced Option
Data serialization	JSON (`encoding/json`)	Protocol Buffers (`google.golang.org/protobuf`)
Unique ID generation	Random hex (`crypto/rand`)	ULID or UUID v4
Time handling	`time.Time` with `time.RFC3339`	Monotonic timestamps for ordering
File storage	Direct filesystem I/O	Embedded database (BoltDB, SQLite)

Recommended File/Module Structure:

```
project-root/
  cmd/byob/                    # CLI entry point
    main.go
  internal/
    runtime/                   # Container Manager component
      container.go             # Container struct and methods
      manager.go               # Runtime interface implementation
      state.go                 # State machine logic
    image/                     # Image Handler component
      image.go                 # Image struct and methods
      store.go                 # Image storage and retrieval
    config/                    # Configuration structures
      types.go                 # All data model structs (ContainerConfig, etc.)
    storage/                   # Persistence layer
      persistence.go           # JSON serialization/deserialization
      paths.go                 # Filesystem path management
  pkg/
    uuid/                      # ID generation utilities
      uuid.go
```

Infrastructure Starter Code (Complete):

Here's a complete, ready-to-use implementation for the core data structures and their JSON persistence:

```go
// internal/config/types.go

package config

import (
    "encoding/json"
    "os"
    "path/filepath"
    "time"
)

type ContainerState string

const (
    StateCreated ContainerState = "created"
    StateRunning ContainerState = "running"
    StatePaused  ContainerState = "paused"
    StateStopped ContainerState = "stopped"
    StateRemoved ContainerState = "removed"
)

type ContainerConfig struct {
    Image      string         `json:"image"`
    Cmd        []string       `json:"cmd,omitempty"`
    Env        []string       `json:"env,omitempty"`
    WorkingDir string         `json:"working_dir,omitempty"`
    Hostname   string         `json:"hostname,omitempty"`
    Limits     ResourceLimits `json:"limits"`
    Network    NetworkConfig  `json:"network"`
}

type ResourceLimits struct {
    MemoryMB  int `json:"memory_mb"`
    CPUShares int `json:"cpu_shares"`
    PidsLimit int `json:"pids_limit"`
}

type NetworkConfig struct {
    Mode         string        `json:"mode"`
    BridgeName   string        `json:"bridge_name,omitempty"`
    IPAddress    string        `json:"ip_address,omitempty"`
    PortMappings []PortMapping `json:"port_mappings,omitempty"`
```

```
}

type PortMapping struct {
    HostPort      int    `json:"host_port"`
    ContainerPort int    `json:"container_port"`
    Protocol      string `json:"protocol"` // "tcp" or "udp"
}

type Container struct {
    ID        string          `json:"id"`
    Name      string          `json:"name"`
    State     ContainerState  `json:"state"`
    Config    ContainerConfig `json:"config"`
    Pid       int             `json:"pid,omitempty"` // 0 if not running
    CreatedAt time.Time       `json:"created_at"`
}

type Image struct {
    Name   string       `json:"name"`
    Digest string       `json:"digest"`
    Layers []ImageLayer `json:"layers"`
    Config ImageConfig  `json:"config"`
}

type ImageLayer struct {
    Digest string `json:"digest"`
    Size   int64  `json:"size"`
    Path   string `json:"path"` // Filesystem path to extracted layer
}

type ImageConfig struct {
    Entrypoint []string `json:"entrypoint,omitempty"`
    Cmd        []string `json:"cmd,omitempty"`
    Env        []string `json:"env,omitempty"`
    WorkingDir string   `json:"working_dir,omitempty"`
}

// internal/storage/persistence.go
package storage

import (
    "encoding/json"
```

```
    "fmt"
    "os"
    "path/filepath"

    "github.com/yourusername/byob/internal/config"
)

type ContainerStore struct {
    basePath string
}

func NewContainerStore(basePath string) *ContainerStore {
    return &ContainerStore{basePath: basePath}
}

func (s *ContainerStore) Save(container *config.Container) error {
    containerPath := filepath.Join(s.basePath, container.ID)
    if err := os.MkdirAll(containerPath, 0755); err != nil {
        return fmt.Errorf("create container directory: %w", err)
    }

    configPath := filepath.Join(containerPath, "config.json")
    data, err := json.MarshalIndent(container, "", "  ")
    if err != nil {
        return fmt.Errorf("marshal container config: %w", err)
    }

    if err := os.WriteFile(configPath, data, 0644); err != nil {
        return fmt.Errorf("write container config: %w", err)
    }

    return nil
}

func (s *ContainerStore) Load(id string) (*config.Container, error) {
    configPath := filepath.Join(s.basePath, id, "config.json")
    data, err := os.ReadFile(configPath)
    if err != nil {
        return nil, fmt.Errorf("read container config: %w", err)
    }
```

```
    var container config.Container
    if err := json.Unmarshal(data, &container); err != nil {
        return nil, fmt.Errorf("unmarshal container config: %w", err)
    }

    return &container, nil
}

func (s *ContainerStore) Delete(id string) error {
    containerPath := filepath.Join(s.basePath, id)
    // Only delete if container is in "removed" state
    configPath := filepath.Join(containerPath, "config.json")
    data, err := os.ReadFile(configPath)
    if err != nil && !os.IsNotExist(err) {
        return fmt.Errorf("check container state: %w", err)
    }

    if err == nil {
        var container config.Container
        if err := json.Unmarshal(data, &container); err == nil {
            if container.State != config.StateRemoved {
                return fmt.Errorf("container %s is not in removed state", id)
            }
        }
    }

    return os.RemoveAll(containerPath)
}

// pkg/uuid/uuid.go
package uuid

import (
    "crypto/rand"
    "encoding/hex"
    "fmt"
)

func Generate() (string, error) {
```

```
    bytes := make([]byte, 16)
    if _, err := rand.Read(bytes); err != nil {
        return "", fmt.Errorf("generate random bytes: %w", err)
    }
    return hex.EncodeToString(bytes), nil
}

func Short() string {
    bytes := make([]byte, 8)
    if _, err := rand.Read(bytes); err != nil {
        // Fallback to timestamp-based ID if crypto fails
        return fmt.Sprintf("%x", time.Now().UnixNano())
    }
    return hex.EncodeToString(bytes)
}
```

Core Logic Skeleton Code:

```go
// internal/runtime/manager.go

package runtime


import (

    "fmt"

    "time"



    "github.com/yourusername/byob/internal/config"

    "github.com/yourusername/byob/pkg/uuid"

)


type ContainerRuntime struct {

    store *storage.ContainerStore

    // Other component managers will be added here

}


func NewContainerRuntime(storePath string) (*ContainerRuntime, error) {

    store := storage.NewContainerStore(storePath)

    // TODO 1: Create store directory if it doesn't exist

    // TODO 2: Initialize other managers (network, cgroup, filesystem)

    // TODO 3: Load existing containers and restore their state

    return &ContainerRuntime{store: store}, nil

}


func (r *ContainerRuntime) CreateContainer(containerConfig config.ContainerConfig, name string) (*config.Container, error) {

    // TODO 1: Validate containerConfig (check image exists, valid resource limits)

    // TODO 2: Generate unique container ID using uuid.Generate()

    // TODO 3: Merge image config with container config (image defaults → container overrides)

    // TODO 4: Create Container struct with State = StateCreated

    // TODO 5: Set CreatedAt to current time

    // TODO 6: Save container to persistence store

    // TODO 7: Prepare root filesystem (but don't start process yet)

    // TODO 8: Return the created container object

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

}


func (r *ContainerRuntime) StartContainer(id string) error {

    // TODO 1: Load container from store

    // TODO 2: Validate container is in StateCreated state

    // TODO 3: Create namespaces using namespace manager

    // TODO 4: Setup cgroups with limits from container.Config.Limits
```

```
    // TODO 5: Configure network using network manager
    // TODO 6: Fork/exec the container process inside the namespaces
    // TODO 7: Update container State = StateRunning, set Pid to child PID
    // TODO 8: Save updated container to store
    // TODO 9: Monitor container process for exit (in goroutine)
    return fmt.Errorf("not implemented")
}

func (r *ContainerRuntime) StopContainer(id string, force bool) error {
    // TODO 1: Load container from store
    // TODO 2: Validate container is in StateRunning or StatePaused state
    // TODO 3: Send SIGTERM (or SIGKILL if force) to container.Pid
    // TODO 4: Wait for process to exit with timeout
    // TODO 5: Clean up resources (cgroups, network, etc.)
    // TODO 6: Update container State = StateStopped, clear Pid field
    // TODO 7: Save updated container to store
    return fmt.Errorf("not implemented")
}

func (r *ContainerRuntime) RemoveContainer(id string) error {
    // TODO 1: Load container from store
    // TODO 2: Validate container is in StateStopped or StateCreated state
    // TODO 3: If StateCreated, clean up prepared resources
    // TODO 4: Delete container root filesystem
    // TODO 5: Update container State = StateRemoved
    // TODO 6: Save updated container to store
    // TODO 7: Delete container directory from store (optional cleanup)
    return fmt.Errorf("not implemented")
}
```

Language-Specific Hints:

1. **JSON serialization:** Use `json.MarshalIndent(container, "", " ")` for human-readable config files. The `omitempty` tag prevents zero values from cluttering the JSON.
2. **Time handling:** Always use `time.Now().UTC()` for timestamps to ensure consistency across timezones. Store as RFC3339 format in JSON.
3. **Error wrapping:** Use `fmt.Errorf("operation: %w", err)` pattern throughout to create actionable error messages with context.
4. **File permissions:** Use `0755` for directories, `0644` for config files, and `0700` for sensitive data.
5. **Path construction:** Always use `filepath.Join()` instead of string concatenation for cross-platform compatibility.

Milestone Checkpoint (Data Model):

After implementing the data model structures and persistence layer, verify functionality with:

```bash
# Create and run a simple test
cd project-root
go test ./internal/config/... ./internal/storage/... -v

# Expected output should show:
# PASS: TestContainerSaveLoad
# PASS: TestImageSerialization
# PASS: TestStateTransitions

# Manual verification:
mkdir -p /tmp/byob-test
go run cmd/byob/main.go create --name test-container alpine:latest echo "hello"
# Check that /tmp/byob-test/containers/ contains a directory with config.json
# The config.json should have all fields properly serialized
```

Debugging Tips for Data Model Issues:

Symptom	Likely Cause	How to Diagnose	Fix
Container config fails to save	Permission denied on store directory	Check `os.MkdirAll` error return	Ensure runtime has write permissions to store path
JSON unmarshal fails	Type mismatch or extra fields	Print raw JSON before unmarshaling	Ensure struct tags match JSON fields exactly
Container ID collisions	Poor random number generation	Test `uuid.Generate()` in a loop	Use crypto/rand with proper error handling
State transitions violating rules	Missing validation logic	Add state transition matrix check	Implement `canTransition(from, to State) bool` helper
Resource leaks on removal	Missing cleanup in RemoveContainer	Add logging for each resource cleanup step	Ensure all managers have cleanup methods called

---

> **Milestone(s):** Milestone 1: Process Isolation (Namespaces)

Component Design: Process Isolation (Milestone 1)

Process isolation is the foundational component of our container runtime. It leverages Linux namespaces to create separate, isolated environments for containerized processes, providing the illusion that each container is running on its own system. This section details the design of the namespace isolation component.

Mental Model: Private Rooms with Mirrored Walls

Imagine a large, open-plan office (the host operating system). To create private work areas, we build enclosed rooms (containers) with special "mirrored walls" (namespaces). Each room has:

1. **Private Bulletin Board (PID Namespace):** A list of everyone in the room (processes). People inside only see their own list. The person who enters first is designated as the "Manager" (PID 1).
2. **Private Nameplate (UTS Namespace):** The room has its own name and company sign (hostname and NIS domain name), independent of the building's main signage.
3. **Private Furniture & Layout (Mount Namespace):** The room can have its own furniture arrangement (filesystem mount points). Adding or removing a desk (mounting/unmounting) in this room doesn't affect the open-plan office or other rooms.
4. **Private Phone System (Network Namespace):** The room has its own phone lines, internal extensions, and switchboard. Calls within the room are private, and external calls must go through the building's main reception (host network).

5. **Private Meeting Room (IPC Namespace):** The room has its own whiteboards and message boards (System V IPC objects, POSIX message queues) that only its occupants can use.

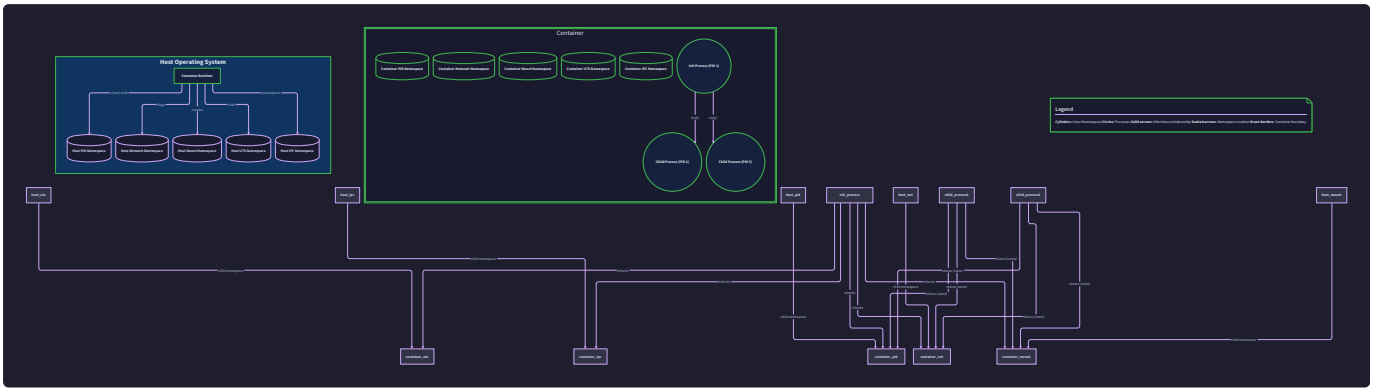
These "mirrored walls" create a powerful illusion: from inside the room, it appears you are in your own, self-contained building. The namespace component is responsible for constructing these private rooms for each container.

Architecture Decision Records for Namespace Strategy

Decision: Use the `clone()` Syscall with Combined Flags for Initial Process Isolation

- **Context:** We need to launch the container's initial process (the `init` process) in a set of isolated namespaces. The two primary Linux APIs for namespace creation are `clone()` (creates a new process in new namespaces) and `unshare()` (moves the calling process into new namespaces). We must choose the primary method for the initial container launch.
- **Options Considered:**
  1. **`clone()` with namespace flags:** Create the container process directly in new namespaces.
  2. **`fork()` + `unshare()`:** Fork the current process, then have the child call `unshare()` to enter new namespaces before executing the container command.
  3. **Combination (clone for most, user ns first):** Use `clone()` with `CLONE_NEWUSER` first, then use `setns()` to join other namespaces for enhanced security.
- **Decision:** Use `clone()` with a combined set of namespace flags (e.g., `CLONE_NEWPID | CLONE_NEWUTS | CLONE_NEWNS | CLONE_NEWIPC | CLONE_NEWNET`) to create the container's first process. This is a single, atomic operation.
- **Rationale:** The `clone()` approach is simpler, more atomic, and aligns with the standard pattern used by runc and other OCI runtimes. It cleanly creates a new process that is immediately isolated, without the intermediate state and potential race conditions of a `fork()` + `unshare()` sequence. It also provides a clear parent-child relationship between the runtime manager and the container `init` process.
- **Consequences:** This decision centralizes namespace creation logic at process launch. It requires the runtime process to have appropriate privileges (typically root) to create most namespaces. It also means the container `init` process will be the direct child of the runtime, simplifying state tracking and signal forwarding.

Option	Pros	Cons	Chosen?
`clone()` with flags	Atomic, simple, standard pattern.	Requires CAP_SYS_ADMIN for most namespaces (except user).	**Yes**
`fork()` + `unshare()`	More flexibility in multi-stage setup.	Non-atomic, complex state management, potential for leaks.	No
User ns first	Enhanced security via privilege dropping.	Significantly more complex, requires mapping UIDs/GIDs.	No (considered for future extension)



Decision: Implement a `NamespaceManager` Component for Lifecycle Operations

- **Context:** Namespace creation, management, and cleanup are distinct operations that must be coordinated with the container's lifecycle. These operations also differ for the initial `init` process versus subsequent `exec` operations (like `docker exec`).
- **Options Considered:**
  1. **Inline in `StartContainer`:** Place all namespace logic directly within the container start sequence.
  2. **Dedicated `NamespaceManager`:** Create a component responsible for namespace operations, providing a clean API.
- **Decision:** Implement a dedicated `NamespaceManager` component. It will expose methods like `CreateNamespaces(flags int)` to launch processes in namespaces and `JoinNetworkNamespace(path string)` for attaching to existing namespaces.
- **Rationale:** A dedicated manager promotes separation of concerns, testability, and code reuse. It encapsulates the low-level syscall details and provides a clear interface for the `ContainerManager`. This pattern also aligns with our other managers (`CgroupManager`, `FilesystemManager`).
- **Consequences:** Introduces an additional component to the architecture but creates a cleaner, more modular design. The `ContainerManager` orchestrates the managers but does not need to know the specifics of `clone()` flags or `/proc/<pid>/ns` file handling.

Option	Pros	Cons	Chosen?
Inline Logic	Fewer components, direct control.	Blurs responsibilities, harder to test and reuse.	No
Dedicated Manager	Separation of concerns, testable, reusable API.	Slight increase in architectural complexity.	**Yes**

Common Pitfalls in Namespace Implementation

⚠️ Pitfall: Incorrect PID Namespace Setup Leading to Host PID Visibility

- **Description:** A process is placed in a new PID namespace (`CLONE_NEWPID`), but running `ps aux` or `ls /proc` inside the container still shows all host processes.
- **Why it's Wrong:** The `/proc` filesystem is a view into the kernel's process information. A new PID namespace only controls the PID numbers; you must also have a separate mount namespace and (re-)mount `/proc` *inside* that namespace for `ps` to reflect the isolated PID view.
- **Fix:** Ensure the container has its own mount namespace (`CLONE_NEWNS`) and mount a new `proc` filesystem on `/proc` after setting up the root filesystem. Do this before executing the final container command.

⚠️ Pitfall: Forgetting to Isolate the Mount Namespace (`CLONE_NEWNS`)

- **Description:** The container can mount or unmount filesystems, and these changes are visible and affect the host and other containers.
- **Why it's Wrong:** Mount operations should be contained within the container. Without `CLONE_NEWNS`, the container shares the global mount table, breaking isolation and creating a security risk.
- **Fix:** Always include `CLONE_NEWNS` in the set of flags passed to `clone()` or `unshare()`. Note: The flag's name is historical; it is the flag for creating a new mount namespace.

⚠️ Pitfall: `CLONE_NEWPID` and the Parent's View

- **Description:** A developer expects the parent process (the container runtime) to see the container's `init` process as PID 1. Instead, the parent still sees the child with a regular host PID.
- **Why it's Wrong:** PID namespaces are hierarchical. A process's PID is relative to its namespace. The parent resides in the *initial* PID namespace, so it sees the child's PID in that namespace. The child, inside the new PID namespace, sees itself as PID 1.
- **Fix:** This is not a bug but a key property to understand. The runtime must track the child's host PID for signaling and cgroup management. Use the PID returned by `clone()` (or `fork()`).

⚠️ Pitfall: Not Cleaning Up Orphaned Namespaces

- **Description:** When a container exits, its network namespace (or others) might persist if not explicitly cleaned up, leading to resource leaks (dangling veth interfaces, IP addresses).
- **Why it's Wrong:** Linux maintains a namespace as long as at least one process remains inside or a reference (like an open file descriptor) exists. The runtime must ensure all references are closed.
- **Fix:** The `NamespaceManager` or `ContainerManager` should be responsible for cleaning up. This includes terminating all processes in the namespace and closing any open file descriptors to `/proc/<pid>/ns/*` that were kept for later `setns()` operations.

Implementation Guidance for Namespace Isolation

This section bridges the design concepts to concrete Go code. We'll structure the namespace management logic within a dedicated component.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Namespace Creation	`syscall.SysProcAttr` in `cmd.SysProcAttr` for `clone()`	Raw `syscall.Clone()` with more granular control
Namespace File Handling	`os.Open` on `/proc/self/ns/*`	`syscall.Open` with `O_CLOEXEC` for safer file descriptors
Process Execution	`exec.Command` with `SysProcAttr`	Low-level `syscall.ForkExec`

B. Recommended File/Module Structure Integrate the `NamespaceManager` into the project structure defined in the High-Level Architecture.

```
build-your-own-docker/
├── cmd/
│   └── byod/                       # CLI entry point
├── internal/
│   ├── container/                  # Container lifecycle and state management
│   │   ├── manager.go              # ContainerManager (orchestrator)
│   │   └── store.go                # ContainerStore (persistence)
│   ├── namespaces/                 # Namespace isolation component (NEW)
│   │   ├── manager.go              # NamespaceManager
│   │   └── util.go                 # Helpers for ns paths, flags
│   ├── cgroups/                    # Resource limits (Milestone 2)
│   ├── fs/                         # Filesystem isolation (Milestone 3,4)
│   ├── network/                    # Networking (Milestone 5)
│   └── image/                      # Image handling (Milestone 6)
└── pkg/
    └── uuid/                       # ID generation utilities
```

C. Infrastructure Starter Code The following is a complete, reusable helper for executing a function in a new set of namespaces. It handles the `clone()` call and the transition into the child's context.

GO

```go
// internal/namespaces/runner.go

package namespaces

import (
    "fmt"
    "os"
    "os/exec"
    "syscall"
)

// ChildFunc is a function to be executed inside the new namespaces.
// The return value will be passed back to the parent as the exit status.
type ChildFunc func() int

// RunInNamespaces runs the provided childFunc in a new set of Linux namespaces.
// The flags parameter is a bitmask of CLONE_NEW* flags (e.g., syscall.CLONE_NEWPID).
// It returns the exit status of the child process as seen by the parent.
func RunInNamespaces(flags int, childFunc ChildFunc) (int, error) {
    // Create a pipe for the child to communicate its exit status.
    // This avoids relying on the child's process exit status which can be
    // ambiguous with signals.
    r, w, err := os.Pipe()
    if err != nil {
        return -1, fmt.Errorf("create pipe: %w", err)
    }
    defer r.Close()
    defer w.Close()

    cmd := &exec.Cmd{
        Path: "/proc/self/exe", // Re-execute the current program
        Args: []string{"byod-child"}, // Special argument for child mode
        SysProcAttr: &syscall.SysProcAttr{
            Cloneflags: syscall.Cloneflags(flags),
        },
        ExtraFiles: []*os.File{w}, // Pipe write end is passed to child
        Stdin:      os.Stdin,
        Stdout:     os.Stdout,
        Stderr:     os.Stderr,
    }

    // Start the child process. It will execute the code path for "byod-child".
```

```
    if err := cmd.Start(); err != nil {
        return -1, fmt.Errorf("start child process: %w", err)
    }

    // Parent: close our copy of the write end.
    w.Close()

    // Wait for the child to finish and read its exit code from the pipe.
    var childStatus int
    if _, err := fmt.Fscanf(r, "%d", &childStatus); err != nil {
        // If the pipe read fails, fall back to cmd.Wait().
        waitErr := cmd.Wait()
        if waitErr != nil {
            if exitErr, ok := waitErr.(*exec.ExitError); ok {
                childStatus = exitErr.ExitCode()
            } else {
                childStatus = -1
            }
        }
    }
    // Ensure the child process is reaped.
    cmd.Wait()
    return childStatus, nil
}

// ChildEntryPoint is called from the main function when args[0] == "byod-child".
// It reads the exit status from the child function and writes it to the pipe.
func ChildEntryPoint(childFunc ChildFunc) {
    // The pipe write end is file descriptor 3 (ExtraFiles[0]).
    pipe := os.NewFile(3, "pipe")
    if pipe == nil {
        os.Exit(255)
    }
    defer pipe.Close()

    status := childFunc()
    fmt.Fprintf(pipe, "%d\n", status)
    os.Exit(status)
}
```

D. Core Logic Skeleton Code The main logic for the `NamespaceManager` involves creating namespaces and providing a way to launch processes within them.

```go
// internal/namespaces/manager.go

package namespaces


import (

    "fmt"

    "os"

    "syscall"

)


// Manager handles creation and management of Linux namespaces.

type Manager struct {

    // We may store paths to namespace files for later joining (e.g., for exec).

    // For simplicity, this initial version focuses on creation.

}


// NewManager creates a new NamespaceManager.

func NewManager() *Manager {

    return &Manager{}

}


// CreateNamespaces launches a process in new namespaces.

// It configures the provided exec.Cmd to run inside the specified namespaces.

// Returns the host PID of the created process.

func (m *Manager) CreateNamespaces(cmd *exec.Cmd, flags int) (int, error) {

    if cmd.SysProcAttr == nil {

        cmd.SysProcAttr = &syscall.SysProcAttr{}

    }

    cmd.SysProcAttr.Cloneflags = syscall.Cloneflags(flags)


    // TODO 1: Ensure the process will have a controlling terminal if needed.

    // TODO 2: Set the Pdeathsig attribute to ensure child dies if parent dies (optional).

    // TODO 3: Start the process using cmd.Start().

    // TODO 4: Return the process's PID (cmd.Process.Pid).


    return 0, fmt.Errorf("not implemented")

}


// GetNamespacePath returns the path to a namespace file for a given process.

// Example: GetNamespacePath(pid, "pid") returns "/proc/pid/ns/pid".

func (m *Manager) GetNamespacePath(pid int, nsType string) (string, error) {

    // TODO 1: Validate nsType against known types: pid, net, mnt, uts, ipc, user.

    // TODO 2: Construct the path: fmt.Sprintf("/proc/%d/ns/%s", pid, nsType).
```

```
    // TODO 3: Check if the path exists (os.Stat).
    // TODO 4: Return the path.

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

// JoinNetworkNamespace configures an exec.Cmd to join an existing network namespace.
// This is used for operations like `docker exec`.
func (m *Manager) JoinNetworkNamespace(cmd *exec.Cmd, nsPath string) error {
    // TODO 1: Open the namespace file at nsPath with syscall.Open.
    // TODO 2: Ensure the file descriptor is closed on exec (FD_CLOEXEC).
    // TODO 3: Set the cmd.SysProcAttr.Cloneflags to avoid creating new namespaces.
    // TODO 4: Set the cmd.SysProcAttr.Setns to the opened file descriptor.
    // TODO 5: Close the file descriptor after cmd.Start().

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

E. Language-Specific Hints

- **Syscall Flags:** Use the constants from the `syscall` package: `syscall.CLONE_NEWPID`, `syscall.CLONE_NEWNS`, `syscall.CLONE_NEWUTS`, `syscall.CLONE_NEWIPC`, `syscall.CLONE_NEWNET`.
- **Process Execution:** The standard `exec.Command` combined with `cmd.SysProcAttr` is sufficient for most namespace operations. For very advanced scenarios (like `clone()` with a custom stack), you would use `syscall.Clone()` directly.
- **Namespace Files:** You can open `/proc/<pid>/ns/<type>` as a regular file. Holding an open file descriptor keeps the namespace alive, which is useful for keeping a network namespace alive after the `init` process dies.

F. Milestone Checkpoint After implementing the namespace isolation component, you should be able to run a simple test program inside an isolated environment.

1. **Expected Output:** Create a test program that prints its PID, hostname, and list of mount points.
2. **Test Command:** Run it using your `RunInNamespaces` helper with flags for PID, UTS, and Mount namespaces.
3. **Verification:**
    - The program should report its PID as 1.
    - It should be able to change its hostname without affecting the host.
    - Running `mount -t proc proc /proc` and then `ls /proc` should only show PIDs 1 and maybe a few kernel threads.
4. **Signs of Trouble:**
    - **PID is not 1:** Ensure you included `syscall.CLONE_NEWPID`.
    - **Host hostname changes:** You forgot `syscall.CLONE_NEWUTS`.
    - **Cannot mount /proc:** You likely need `syscall.CLONE_NEWNS` (mount namespace). Also, the process needs `CAP_SYS_ADMIN` capability (run as root).

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Child process exits immediately with no output.	The child function is not being executed, or the `exec.Command` arguments are wrong.	Add debug prints before and inside the child function. Check `cmd.Path` and `cmd.Args`.	Ensure `ChildEntryPoint` is called correctly in your main function when the special argument is present.
`ps aux` inside container shows all host processes.	`/proc` is not remounted inside the new PID namespace.	Check if you have `CLONE_NEWNS` and are mounting proc after `chroot` / `pivot_root`.	Mount a new procfs: `syscall.Mount("proc", "/proc", "proc", 0, "")` inside the container's mount namespace.
Container can still see host network interfaces.	`CLONE_NEWNET` flag was not included.	Check the flags passed to `RunInNamespaces` or `CreateNamespaces`.	Add `syscall.CLONE_NEWNET` to the flag set.
Getting "operation not permitted" errors.	Insufficient privileges (not running as root).	Run your runtime with `sudo`.	Ensure the binary is executed as root (or has the necessary capabilities via `setcap`).

> **Milestone(s):** Milestone 2: Resource Limits (cgroups)

Component Design: Resource Limits via cgroups (Milestone 2)

This component is responsible for enforcing resource constraints on containerized processes using Linux **cgroups** (control groups). While namespaces provide isolation boundaries, cgroups provide the resource accounting and limiting mechanisms that prevent any single container from monopolizing system resources like CPU, memory, or process slots. This component translates the abstract `ResourceLimits` configuration into concrete kernel-level resource controls.

Mental Model: Resource Quotas in an Apartment Building

Imagine an apartment building where each tenant (container) has access to shared building resources. Without controls, one tenant could:

1. Use all the building's water pressure (CPU)
2. Generate so much garbage that the dumpsters overflow (memory)
3. Invite hundreds of guests who block the hallways (processes)

The building manager (cgroup manager) solves this by installing:

- **Individual water meters with flow restrictors** (CPU limits): Each apartment gets a maximum flow rate
- **Designated garbage bin capacity** (memory limits): Each apartment gets a specific bin size; overflowing bins get removed (OOM-killed)
- **Guest limit per apartment** (PIDs limit): Each apartment can only register a set number of residents

These quotas apply collectively to everyone in the apartment, not per person. If you have five roommates sharing an apartment, they collectively share the apartment's quotas. Similarly, all processes within a container (including child processes and their descendants) share the container's cgroup resource limits.

The key insight is that cgroups don't just monitor resource usage—they actively enforce hard limits through kernel mechanisms that throttle, deny, or terminate processes that exceed their allocations. This enforcement happens at the kernel level, making it both efficient and unavoidable for processes within the cgroup.

Architecture Decision Records for cgroup Version

Decision: cgroup v2 as Primary Target with v1 Fallback

Context: Linux has two major cgroup implementations: the original cgroup v1 (with multiple, sometimes inconsistent controllers) and the unified hierarchy of cgroup v2. Modern distributions (Ubuntu 22.04+, Fedora 31+, RHEL 8+) default to cgroup v2, but many production systems still run cgroup v1. Our container runtime must work reliably across both versions.

Options Considered:

1. **Exclusive cgroup v2 support**: Assume modern Linux and only implement v2 interfaces
2. **Exclusive cgroup v1 support**: Target older systems but miss modern features like memory pressure notifications
3. **Dual-stack with automatic detection**: Detect available cgroup version and use appropriate implementation
4. **User-configurable with automatic fallback**: Allow explicit configuration but fall back based on system detection

Option	Pros	Cons	Chosen?
Exclusive cgroup v2	Cleaner API, unified hierarchy, better memory management	Doesn't work on older systems (~30% of production)	No
Exclusive cgroup v1	Maximum compatibility with existing systems	Misses v2 improvements, inconsistent controller mounting	No
Dual-stack auto-detection	Works everywhere, uses best available version	More complex implementation, testing burden	**Yes**
Configurable with fallback	User control when needed, still works automatically	Complexity of config parsing and validation	No

Decision: Implement dual-stack cgroup management with automatic version detection. The runtime will check `/sys/fs/cgroup/cgroup.controllers` to determine if cgroup v2 is mounted; if present, use v2 interfaces; otherwise, fall back to cgroup v1 with separate controller hierarchies.

Rationale:

1. **User experience**: Most users won't know or care about cgroup versions; they expect containers to "just work"
2. **Forward compatibility**: As systems upgrade to cgroup v2, our runtime automatically uses the better API
3. **Maintainability**: We can implement a common interface with version-specific backends, keeping logic clean
4. **Production readiness**: Many container orchestration systems (Kubernetes, Docker) already use this approach

Consequences:

- We must implement and test two code paths
- Resource limit configuration may need translation between versions (e.g., CPU shares vs CPU weight)
- Some advanced features (like cgroup v2's memory pressure stall information) may only be available in v2 mode
- The implementation must handle mixed systems where some controllers are v1 and others v2 (hybrid mode)

Decision: Hierarchical cgroup Organization by Container ID

Context: cgroups can be organized in hierarchies. We need to decide how to structure our cgroups relative to each other and to system cgroups.

Options Considered:

1. **Flat organization**: All containers at the same level under a runtime parent
2. **Hierarchical by container ID**: Each container gets its own nested cgroup
3. **Pool-based organization**: Group containers by resource class (high-memory, low-CPU, etc.)
4. **Integration with systemd**: Delegate cgroup management to systemd via `systemd-run`

Option	Pros	Cons	Chosen?
Flat organization	Simple to implement, easy to enumerate	No hierarchy for resource delegation, messy cleanup	No
Hierarchical by container ID	Clean isolation, natural cleanup path, supports delegation	Slightly more complex to create nested groups	**Yes**
Pool-based	Could optimize resource allocation	Complex to manage, requires scheduling logic	No
systemd integration	Leverages system's cgroup manager, consistent with OS	Adds systemd dependency, harder to debug	No

Decision: Create hierarchical cgroups using the container ID as the cgroup name, nested under a runtime-managed parent cgroup (e.g., `/sys/fs/cgroup/container_runtime/{container_id}` for v2 or per-controller paths for v1).

Rationale:

1. **Clean resource accounting**: Each container's resource usage is clearly isolated and measurable
2. **Simplified cleanup**: Removing the container cgroup recursively removes all resource tracking
3. **Delegation support**: Future extensions could delegate sub-cgroups to containers (for nested containers)
4. **Consistency with Docker**: Docker uses similar hierarchical organization, aiding user familiarity

Consequences:

- Must ensure proper cleanup on container exit to avoid "cgroup leakage"
- Need to handle cgroup filesystem permissions (typically require root)
- Nested cgroups add path complexity that must be managed in both v1 and v2

Component Responsibilities and Interfaces

The cgroup management component has three primary responsibilities:

1. **cgroup Creation and Configuration**: Establish cgroup hierarchies and apply resource limits before process execution

2. **Process Attachment**: Move processes (and their future descendants) into the appropriate cgroups
3. **Cleanup and Resource Reclamation**: Remove cgroups when containers are destroyed

Data Structures

Type	Fields	Description
`CgroupManager`	`version int`, `basePath string`, `controllers map[string]bool`	Manages cgroup operations, detects version, tracks available controllers
`ResourceLimits`	`MemoryMB int`, `CPUShares int`, `PidsLimit int`	User-specified resource constraints (from `ContainerConfig`)
`CgroupStats`	`MemoryUsage int64`, `CPUUsage int64`, `PidCount int`	Runtime statistics collected from cgroup interface files

Interface Methods

Method	Parameters	Returns	Description
`NewCgroupManager()`	none	`(*CgroupManager, error)`	Factory function that detects cgroup version and available controllers
`CreateCgroup(containerID string)`	`containerID string`	`error`	Creates cgroup hierarchy for the container
`ApplyLimits(containerID string, limits ResourceLimits)`	`containerID string`, `limits ResourceLimits`	`error`	Writes resource limits to cgroup control files
`AddProcess(containerID string, pid int)`	`containerID string`, `pid int`	`error`	Moves a process (and its future children) into the container's cgroup
`RemoveCgroup(containerID string)`	`containerID string`	`error`	Recursively removes the container's cgroup and all sub-cgroups
`GetStats(containerID string)`	`containerID string`	`(*CgroupStats, error)`	Reads current resource usage statistics from cgroup interface

Common Pitfalls in cgroup Implementation

⚠️ Pitfall: Forgetting to Add the Process to the cgroup

Description: Creating cgroups and setting limits but never actually moving the container process into them.

Why it's wrong: cgroup limits only apply to processes within the cgroup. If the container process remains in the root cgroup (or a different cgroup), it won't be subject to the limits you configured.

How to fix: Always call `AddProcess` with the container's PID **after** `fork` but **before** `exec` in the child process. Better yet, use the cgroup's `cgroup.procs` file (v2) or `tasks` file (v1) which automatically includes all child processes.

⚠️ Pitfall: Not Handling cgroup v1 Controller Mounts

Description: Assuming all cgroup controllers are mounted in standard locations in cgroup v1.

Why it's wrong: In cgroup v1, each controller (cpu, memory, pids) can be mounted separately, and systems may have different mount points or hierarchies. Some controllers might not be mounted at all.

How to fix: Parse `/proc/mounts` to find controller mount points, check `/proc/cgroups` for available controllers, and provide clear error messages if required controllers are missing.

⚠️ Pitfall: Memory Limits Don't Account for Kernel Memory

Description: Setting only `memory.limit_in_bytes` without considering `memory.kmem.limit_in_bytes` in cgroup v1.

Why it's wrong: Containers can use kernel memory (page cache, sockets, etc.) that isn't counted against the normal memory limit. A container could exceed total memory through kernel allocations and not be OOM-killed.

How to fix: In cgroup v1, also set `memory.kmem.limit_in_bytes` (typically to the same value as `memory.limit_in_bytes`). In cgroup v2, the unified `memory.max` includes both user and kernel memory.

⚠️ Pitfall: Not Cleaning Up cgroups on Container Exit

Description: Leaving cgroup directories after container termination.

Why it's wrong: Accumulating empty cgroups wastes inodes, complicates monitoring, and may hit filesystem limits. It also leaks information about past containers.

How to fix: Implement `RemoveCgroup` and call it during container removal. Use `SetFinalizer` or defer statements to ensure cleanup even on unexpected exits. Consider using cgroup notification to detect when all processes in a cgroup have exited.

⚠️ Pitfall: Misunderstanding CPU Shares vs Quota/Period

Description: Confusing the relative `cpu.shares` with the absolute `cpu.cfs_quota_us` / `cpu.cfs_period_us`.

Why it's wrong: CPU shares only matter when there's contention; a container with 1024 shares gets twice as much CPU as one with 512 shares **only when the CPU is saturated**. Quota/period provides absolute limits (e.g., 0.5 CPU cores maximum).

How to fix: Understand your use case: use shares for fair sharing among containers, use quota for hard limits. Document this clearly for users configuring `ResourceLimits`.

Implementation Algorithm

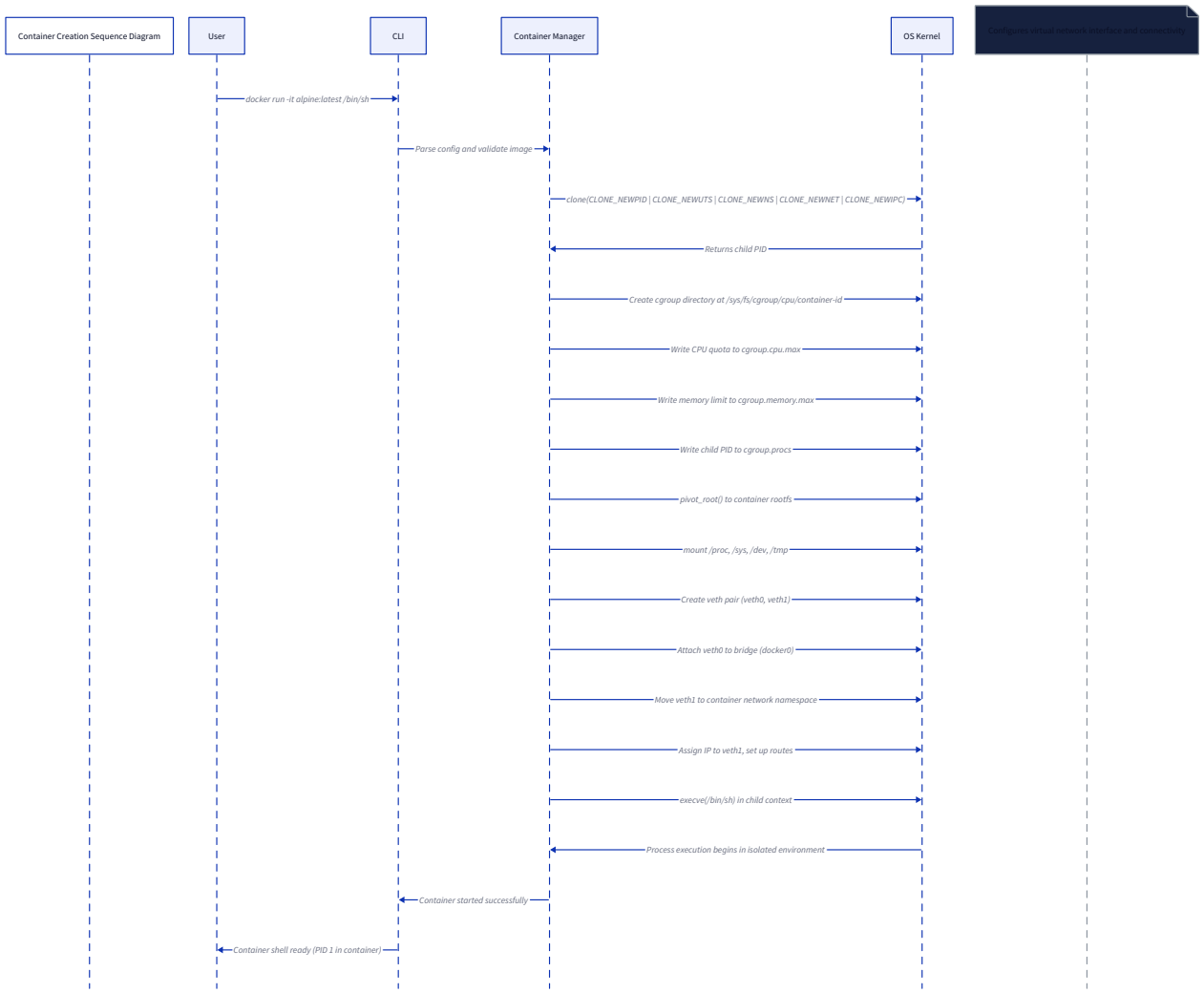
The cgroup setup follows this sequence during container creation:

1. **Version Detection** (once at runtime initialization):
    1. Check if `/sys/fs/cgroup/cgroup.controllers` exists
    2. If yes, parse it to determine available controllers (cgroup v2)
    3. If no, check `/proc/cgroups` and `/proc/mounts` for cgroup v1 controllers
2. **cgroup Creation** (per container):
    1. Generate cgroup path: `/sys/fs/cgroup/container_runtime/{container_id}` (v2) or per-controller paths (v1)
    2. Create directory with appropriate permissions (typically 0755)
    3. For cgroup v1, repeat for each required controller (cpu, memory, pids)
3. **Limit Application**:
    1. Convert `ResourceLimits` to cgroup-specific values:
        - Memory: MB → bytes (multiply by 1,048,576)
        - CPU shares: Keep as is (1024 = default weight)
        - PIDs limit: Set directly
    2. Write to appropriate control files:
        - cgroup v2: `memory.max`, `cpu.weight`, `pids.max`
        - cgroup v1: `memory.limit_in_bytes`, `memory.kmem.limit_in_bytes`, `cpu.shares`, `pids.max`
    3. Verify writes by reading back and comparing
4. **Process Attachment**:
    1. After `fork` but before `exec` in the child, get the child's PID
    2. Write PID to `cgroup.procs` (v2) or each controller's `tasks` file (v1)
    3. Verify the process appears in the cgroup by reading the file back
5. **Cleanup** (on container removal):
    1. Kill all processes in the cgroup (if not already dead)
    2. Remove all subdirectories recursively
    3. Remove the container's cgroup directory
    4. Verify removal by checking directory existence

> **Design Insight**: The critical timing requirement is that processes must be added to cgroups **before they start executing user code**. If you add them after `exec`, there's a race window where the process could allocate memory or fork child processes outside the limits. Always use the pattern: `fork` → `add to cgroup` → `apply namespaces` → `exec`.

Integration with Container Creation Sequence

Referencing the



, cgroup operations occur between namespace creation and filesystem setup:

1. CLI receives `run` command with `--memory`, `--cpus` flags
2. Container Manager creates `ContainerConfig` with `ResourceLimits`
3. Namespace Manager creates new namespaces via `clone()` or `unshare()`
4. **Cgroup Manager creates cgroup and applies limits**
5. **Cgroup Manager adds the forked child process to the cgroup**
6. Filesystem Manager sets up rootfs with `pivot_root()`
7. Network Manager configures veth pair and network namespace
8. Container process executes entrypoint within all constraints

This ordering ensures that:

- Resource limits are active before the process starts meaningful work
- The process inherits the cgroup membership, so all its children are automatically in the same cgroup
- If cgroup setup fails, we can abort before setting up more complex resources like network

Error Handling and Recovery

Failure Mode	Detection	Recovery
cgroup filesystem not mounted	`open()` on cgroup path returns `ENOENT` or `EACCES`	Check `/proc/mounts`, suggest mounting cgroups, or fall back to no resource limits (with warning)
Insufficient permissions	`open()` or `write()` returns `EACCES`	Require root/sudo, or suggest configuring cgroup delegation
Invalid limit value	`write()` succeeds but kernel rejects value (may be silent)	Read back after write to verify, validate limits against system capacity first
Controller not available	Controller file doesn't exist in cgroup directory	Skip that controller, warn user, continue without that limit
Process already in cgroup	`write()` to `cgroup.procs` returns `EBUSY` (v2)	Check if PID is already in correct cgroup, continue if so
cgroup cleanup fails	`rmdir()` returns `EBUSY` (processes still exist)	Attempt to kill remaining processes, retry removal

Example Walkthrough: Memory Limit Enforcement

Consider a container configured with `MemoryMB: 100` (100MB memory limit):

1. **Configuration**: `ResourceLimits{MemoryMB: 100, CPUShares: 512, PidsLimit: 100}` is created
2. **Translation**: CgroupManager converts 100MB to 104,857,600 bytes
3. **Writing**: For cgroup v2, writes "104857600" to `/sys/fs/cgroup/container_runtime/abc123/memory.max`
4. **Enforcement**: Kernel tracks container's resident memory usage via RSS (Resident Set Size)
5. **Exceedance**: If container allocates 101MB, kernel invokes OOM killer on processes in the cgroup
6. **Notification**: CgroupManager could watch `memory.events` file for "oom_kill" events
7. **Cleanup**: When container exits, cgroup directory is removed

> **Key Insight**: Memory limits in cgroups use RSS, which measures physical memory pages. This differs from virtual memory size (which can be much larger due to memory-mapped files, shared libraries, and swap). A container with 100MB limit might have 200MB of virtual memory but only 80MB of RSS.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
cgroup Detection	Parse `/proc/mounts` manually	Use `github.com/containerd/cgroups` library
File Operations	Standard `os.Open`, `os.WriteFile`	Memory-mapped files for frequent stats polling
Process Tracking	Periodic scanning of `cgroup.procs`	Inotify on cgroup directory for real-time updates
Statistics Collection	Read interface files on demand	Background goroutine with caching for dashboard

B. Recommended File/Module Structure

```
project-root/
  cmd/
    runtime/           # CLI entry point
      main.go
  internal/
    cgroup/            # cgroup management component
      manager.go       # Main CgroupManager implementation
      detector.go      # cgroup version and controller detection
      v1/              # cgroup v1 specific implementation
        controller.go  # Per-controller operations
      v2/              # cgroup v2 specific implementation
        unified.go     # Unified hierarchy operations
      common.go        # Shared utilities and types
      manager_test.go  # Unit tests
    container/         # Container lifecycle management
      manager.go
    namespace/         # Namespace isolation
      manager.go
    # ... other components
```

C. Infrastructure Starter Code

Here's a complete, ready-to-use cgroup detector that handles both v1 and v2:

```go
// internal/cgroup/detector.go

package cgroup


import (

        "fmt"

        "io/ioutil"

        "os"

        "path/filepath"

        "strings"

)


// CgroupVersion represents detected cgroup version

type CgroupVersion int


const (

        CgroupV1 CgroupVersion = 1

        CgroupV2 CgroupVersion = 2

        CgroupUnknown CgroupVersion = 0

)


// ControllerInfo holds detected controller availability

type ControllerInfo struct {

        Version     CgroupVersion

        Controllers map[string]bool  // Available controllers

        MountPoints map[string]string // Controller -> mount path (v1 only)

}


// DetectCgroupInfo detects cgroup version and available controllers

func DetectCgroupInfo() (*ControllerInfo, error) {

        info := &ControllerInfo{

                Controllers: make(map[string]bool),

                MountPoints: make(map[string]string),

        }



        // Check for cgroup v2 unified hierarchy

        cgroupV2Path := "/sys/fs/cgroup"

        if controllers, err := ioutil.ReadFile(filepath.Join(cgroupV2Path, "cgroup.controllers")); err == nil {

                info.Version = CgroupV2

                for _, controller := range strings.Fields(string(controllers)) {

                        info.Controllers[controller] = true

                }
```

```
        return info, nil
    }

    // Fall back to cgroup v1 detection
    info.Version = CgroupV1

    // Parse /proc/cgroups for available controllers
    procCgroups, err := ioutil.ReadFile("/proc/cgroups")
    if err != nil {
        return nil, fmt.Errorf("cannot read /proc/cgroups: %v", err)
    }

    lines := strings.Split(string(procCgroups), "\n")
    for _, line := range lines[1:] { // Skip header
        fields := strings.Fields(line)
        if len(fields) >= 4 && fields[3] == "1" { // Enabled controller
            controller := fields[0]
            info.Controllers[controller] = true
        }
    }

    // Find mount points from /proc/mounts
    mounts, err := ioutil.ReadFile("/proc/mounts")
    if err != nil {
        return nil, fmt.Errorf("cannot read /proc/mounts: %v", err)
    }

    for _, line := range strings.Split(string(mounts), "\n") {
        fields := strings.Fields(line)
        if len(fields) >= 3 && fields[2] == "cgroup" {
            options := strings.Split(fields[3], ",")
            for _, opt := range options {
                if opt != "cgroup" && !strings.HasPrefix(opt, "__") {
                    info.MountPoints[opt] = fields[1]
                }
            }
        }
    }
```

```
        // Verify we have required controllers
        required := []string{"memory", "cpu", "pids"}
        for _, req := range required {
                if !info.Controllers[req] {
                        return nil, fmt.Errorf("required cgroup controller %s not available", req)
                }
                if _, mounted := info.MountPoints[req]; !mounted && info.Version == CgroupV1 {
                        return nil, fmt.Errorf("cgroup controller %s not mounted", req)
                }
        }

        return info, nil
}
```

D. Core Logic Skeleton Code

```go
// internal/cgroup/manager.go

package cgroup


import (

	"fmt"

	"os"

	"path/filepath"

	"strconv"

)


// CgroupManager manages cgroup operations for containers

type CgroupManager struct {

	version      CgroupVersion

	basePath     string            // Base path for cgroup v2 or parent for v1

	controllers  map[string]bool   // Available controllers

	mountPoints  map[string]string // Controller mount points (v1 only)

}


// NewCgroupManager creates a new cgroup manager with auto-detection

func NewCgroupManager() (*CgroupManager, error) {

	info, err := DetectCgroupInfo()

	if err != nil {

		return nil, fmt.Errorf("failed to detect cgroup info: %v", err)

	}



	manager := &CgroupManager{

		version:     info.Version,

		controllers: info.Controllers,

		mountPoints: info.MountPoints,

	}



	// Set base path based on version

	if manager.version == CgroupV2 {

		manager.basePath = "/sys/fs/cgroup"

	} else {

		// For v1, we'll use a parent directory under each controller

		manager.basePath = "/sys/fs/cgroup"

	}
```

```go
        return manager, nil
}

// CreateCgroup creates a cgroup for the given container ID
func (m *CgroupManager) CreateCgroup(containerID string) error {
        if m.version == CgroupV2 {
                // TODO 1: Construct full path: filepath.Join(m.basePath, "container_runtime", containerID)
                // TODO 2: Create directory with os.MkdirAll, permissions 0755
                // TODO 3: Verify directory was created and is writable
                // TODO 4: For v2, enable controllers by writing to cgroup.subtree_control
        } else {
                // TODO 5: For v1, create directory under each controller's mount point
                // TODO 6: For each required controller (memory, cpu, pids):
                //   - Build path: filepath.Join(mountPoints[controller], "container_runtime", containerID)
                //   - Create directory with os.MkdirAll
                // TODO 7: Verify all required controller directories were created
        }
        return nil
}

// ApplyLimits writes resource limits to the container's cgroup
func (m *CgroupManager) ApplyLimits(containerID string, limits ResourceLimits) error {
        if m.version == CgroupV2 {
                // TODO 8: Build cgroup path for this container
                // TODO 9: Apply memory limit: convert MB to bytes, write to "memory.max"
                // TODO 10: Apply CPU weight: convert CPUShares to v2 weight (1-10000), write to "cpu.weight"
                // TODO 11: Apply PIDs limit: write to "pids.max"
                // TODO 12: Verify each write by reading back and comparing
        } else {
                // TODO 13: For v1, apply limits to each controller separately
                // TODO 14: Memory: write to memory.limit_in_bytes and memory.kmem.limit_in_bytes
                // TODO 15: CPU: write cpu.shares to cpu directory
                // TODO 16: PIDs: write to pids.max in pids directory
                // TODO 17: Handle memory swappiness (optional): write to memory.swappiness
        }
        return nil
}

// AddProcess adds a process to the container's cgroup
func (m *CgroupManager) AddProcess(containerID string, pid int) error {
```

```
        pidStr := strconv.Itoa(pid)


        if m.version == CgroupV2 {
                // TODO 18: Write PID to cgroup.procs file in container's cgroup
                // TODO 19: Verify write succeeded by checking if PID appears in file
        } else {
                // TODO 20: For v1, write PID to tasks file in each controller's cgroup
                // TODO 21: Write to memory, cpu, and pids controller tasks files
                // TODO 22: Handle case where process might already be in a cgroup
        }
        return nil
}

// RemoveCgroup recursively removes the container's cgroup
func (m *CgroupManager) RemoveCgroup(containerID string) error {
        if m.version == CgroupV2 {
                // TODO 23: Build full cgroup path
                // TODO 24: Kill any remaining processes in cgroup (read cgroup.procs, send SIGKILL)
                // TODO 25: Remove directory with os.RemoveAll
                // TODO 26: Retry a few times if directory is busy (EBUSY)
        } else {
                // TODO 27: For v1, remove directory under each controller
                // TODO 28: For each controller, kill processes, then remove directory
        }
        return nil
}
```

E. Language-Specific Hints

1. **File Operations**: Use `ioutil.WriteFile` for atomic writes to cgroup control files. The kernel expects entire content at once, not partial writes.
2. **Error Handling**: Check for `os.IsPermission` and `os.IsNotExist` to provide helpful error messages about cgroup configuration.
3. **Concurrency**: Use a mutex (`sync.Mutex`) in `CgroupManager` if multiple goroutines might create cgroups concurrently.
4. **Cleanup**: Implement `RemoveCgroup` as an idempotent operation—calling it multiple times should be safe.
5. **Path Safety**: Always use `filepath.Join` instead of string concatenation to handle path separators correctly across systems.

F. Milestone Checkpoint

Test Command:

```
sudo go test ./internal/cgroup/... -v
```
BASH

Expected Output:

```
=== RUN   TestCgroupDetection
--- PASS: TestCgroupDetection (0.01s)
=== RUN   TestCreateCgroup
--- PASS: TestCreateCgroup (0.02s)
=== RUN   TestApplyMemoryLimit
--- PASS: TestApplyMemoryLimit (0.03s)
=== RUN   TestAddProcess
--- PASS: TestAddProcess (0.05s)
PASS
```

Manual Verification:

1. Run a container with memory limit: `sudo ./runtime run --memory 50 alpine echo "hello"`
2. Check cgroup exists: `sudo cat /sys/fs/cgroup/container_runtime/{container_id}/memory.max`
3. Should show `52428800` (50MB in bytes)
4. Run a memory-hungry process in container and verify OOM kill occurs when exceeding limit

Debugging Tips:

- If cgroup creation fails with "permission denied", ensure running as root
- If memory limit isn't enforced, check if you're using cgroup v1 but forgot `memory.kmem.limit_in_bytes`
- If `AddProcess` fails with "no such process", ensure you're using the correct PID (the child's PID, not the parent's)

G. Debugging Tips Table

Symptom	Likely Cause	How to Diagnose	Fix
Container not subject to memory limit	Process not in cgroup	Check `cat /sys/fs/cgroup/container_runtime/{id}/cgroup.procs`	Call `AddProcess` before `exec`
Memory limit enforced but container uses more	Kernel memory not limited (v1)	Check `cat /sys/fs/cgroup/memory/container_runtime/{id}/memory.kmem.usage_in_bytes`	Set `memory.kmem.limit_in_bytes`
CPU limit has no effect	CPU not saturated on host	Check `top` to see if system CPU usage is low	Shares only matter during contention; use quota/period for hard limits
cgroup cleanup fails	Processes still running in cgroup	`ls -la /proc/{pid}/cgroup` for PIDs in cgroup	Kill processes before removing cgroup
"no such file" errors	cgroup v2 not mounted	Check `mount	grep cgroup`	

> **Milestone(s):** Milestone 3: Filesystem Isolation (chroot/pivot_root)

Component Design: Filesystem Isolation (Milestone 3)

This component is responsible for giving each container its own private view of the filesystem, completely isolated from the host's filesystem hierarchy. While namespaces provide process isolation, filesystem isolation ensures the container can only see and modify files within its designated root filesystem (`rootfs`). This creates the illusion for the containerized process that it's running on a complete, independent operating system with its own `/bin`, `/lib`, `/etc`, and other directories, even though these are just directories on the host.

Mental Model: Giving Each Apartment Its Own Furniture

Think of the host filesystem as a massive, fully-furnished mansion. When we create containers, we're setting up individual apartments within this mansion. Without filesystem isolation, each apartment resident would have access to the entire mansion's furniture, appliances, and personal belongings—a clear security and privacy problem.

Filesystem isolation is like giving each apartment its own complete set of furniture and appliances that only that tenant can see and use:

1. **Private Furniture (Root Filesystem)**: Each container gets its own `/` (root directory) containing all the files it needs to operate. This `rootfs` is typically extracted from a container image (like Alpine or Ubuntu). The container can install packages, create files, and modify configuration within this space without affecting other containers or the host.
2. **Mirrored Utilities (Essential Filesystems)**: While the container has its own furniture, it still needs access to certain "utilities" provided by the building. We mount special filesystems like `/proc` and `/sys` inside the container so it can see its own processes and system information, not the host's. This is like providing each apartment with electricity and water connections that are metered separately.
3. **Locked Doors (Unmounting Host Access)**: After moving the container into its private apartment, we lock the door to the rest of the mansion. This is achieved by unmounting the host's root filesystem from within the container's mount namespace, ensuring the container cannot access any host files even if it knows their paths.
4. **Shared Storage Areas (Bind Mounts)**: Sometimes tenants need access to shared storage areas, like a communal laundry room. We can provide this through bind mounts—taking specific directories from the host and making them available at specific locations within the container. This is controlled and explicit, unlike the unrestricted access of `chroot` without mount namespaces.

The key insight is that filesystem isolation isn't about creating physical copies of files (that would be wasteful), but about creating a **virtual view** where the container sees only its assigned directory tree as the entire filesystem universe.

Architecture Decision Records: pivot_root vs chroot

> **Decision: Use pivot_root with Mount Namespaces Instead of chroot**
>
> - **Context**: We need to give containers an isolated view of the filesystem. The traditional approach is `chroot`, which changes the root directory for a process and its children. However, `chroot` has security limitations and doesn't play well with modern container requirements like proper `/proc` mounting and clean separation from the host.
> - **Options Considered**:
>   1. **chroot only**: Simple system call that changes the root directory
>   2. **chroot with mount namespace**: chroot inside a mount namespace
>   3. **pivot_root with mount namespace**: Modern approach that completely swaps root filesystems
> - **Decision**: Use `pivot_root` combined with a mount namespace (`CLONE_NEWNS`).
> - **Rationale**:
>   - `pivot_root` is designed for containerization and allows us to completely replace the root filesystem while keeping the old root accessible for cleanup
>   - It works atomically and avoids race conditions that `chroot` can have
>   - Combined with a mount namespace, it allows us to unmount the old root entirely, preventing any access to host files
>   - The OCI runtime specification recommends `pivot_root` for proper isolation
> - **Consequences**:
>   - Slightly more complex implementation than `chroot`
>   - Requires creating a mount namespace first
>   - Provides stronger security guarantees and cleaner isolation
>   - Compatible with overlayfs and other union filesystems

Comparison Table: Filesystem Isolation Approaches

Option	Pros	Cons	Why Not Chosen?
chroot only	Simple single system call, widely understood	Process can escape via file descriptors or procfs, doesn't isolate mounts, old root remains accessible	Security vulnerabilities, insufficient isolation for containers
chroot + mount namespace	Better isolation than chroot alone, allows private mount points	Still uses chroot which has escape vectors, old root remains mounted somewhere	`pivot_root` is more secure and designed for this purpose
pivot_root + mount namespace (CHOSEN)	Complete root replacement, old root can be unmounted, atomic operation, OCI-recommended	More complex, requires temporary directories	Best practice for container runtimes, provides proper isolation

Common Pitfalls in Filesystem Isolation

⚠️ Pitfall 1: Using chroot Without Mount Namespace

- **Description**: Using only `chroot()` to change the root directory without first creating a mount namespace.
- **Why it's wrong**: The process can escape the chroot jail using various techniques: keeping file descriptors to host directories, using `..` paths with `/proc/self/fd`, or mounting new filesystems. The host's `/proc` is still visible unless remounted.
- **How to fix**: Always create a mount namespace (`CLONE_NEWNS`) before calling `chroot`, or better yet, use `pivot_root` which requires a mount namespace.

⚠️ Pitfall 2: Forgetting to Mount /proc Inside Container

- **Description**: After isolating the filesystem, not mounting a new `/proc` filesystem inside the container.
- **Why it's wrong**: Many utilities (`ps` , `top` , `ls /proc`) will fail or show host processes. The container's PID namespace needs its own `/proc` to reflect container PIDs, not host PIDs.
- **How to fix**: After `pivot_root`, mount a new proc filesystem at `/proc` inside the container with appropriate options (`nosuid,nodev,noexec`).

⚠️ Pitfall 3: Not Unmounting the Old Root

- **Description**: After `pivot_root`, leaving the old root filesystem accessible at the pivot directory.
- **Why it's wrong**: The container could access host files through the old root mount point, breaking isolation.
- **How to fix**: Always unmount the old root directory (with `MNT_DETACH` flag) after successful `pivot_root` and changing directory to the new root.

⚠️ Pitfall 4: Using Relative Paths for pivot_root

- **Description**: Passing relative paths to `pivot_root` instead of absolute paths.
- **Why it's wrong**: `pivot_root` system call requires both arguments to be absolute paths. Relative paths cause the call to fail with `EINVAL`.
- **How to fix**: Always convert paths to absolute using `filepath.Abs()` or similar before calling `pivot_root`.

⚠️ Pitfall 5: Incomplete Root Filesystem

- **Description**: The extracted `rootfs` doesn't contain essential binaries or libraries needed by the container's entrypoint.
- **Why it's wrong**: Container process fails to start with "executable not found" or shared library errors, even though the binary exists in the path.
- **How to fix**: Ensure the container image is properly extracted with all dependencies. Use minimal but complete base images like Alpine, and verify that required dynamic libraries are present with `ldd`.

Implementation Guidance for Root Filesystem Setup

The filesystem isolation component orchestrates multiple steps to prepare and switch to the container's root filesystem. Here's the complete procedure:

Step-by-Step Filesystem Isolation Procedure

1. **Prepare Container Directories**:
   - Create a unique directory for the container under the runtime's storage path (e.g., `/var/lib/container-runtime/containers/<id>/`)
   - Create subdirectories: `rootfs` (for the final merged view), `workdir` (for OverlayFS), `oldroot` (for pivot temporary)
2. **Extract or Prepare Root Filesystem**:
   - If using a container image, extract all layers to create the `rootfs` (using OverlayFS for Milestone 4)
   - If using a direct rootfs directory, copy or bind mount it to the container's `rootfs` directory
   - Ensure the rootfs contains essential directories (`/dev` , `/proc` , `/sys` , `/tmp` will be created or mounted later)
3. **Create Mount Namespace**:

- Use `CLONE_NEWNS` flag when creating the container process to give it a private mount namespace
- This ensures all subsequent mount operations only affect the container, not the host

4. **Mount Essential Virtual Filesystems**: Before `pivot_root`, mount essential filesystems that should be available in the new root:
   - Mount `tmpfs` at `/dev` for device nodes (or bind mount `/dev/null`, `/dev/zero`, `/dev/random`, etc.)
   - Create basic device nodes if needed (`mknod` for console, null, etc.)
   - Mount `proc` at `rootfs/proc` with `nosuid,nodev,noexec` flags
   - Mount `sysfs` at `rootfs/sys` with `nosuid,nodev,noexec,ro` flags (read-only for security)
   - Mount `tmpfs` at `rootfs/tmp` with appropriate size limits
5. **Perform pivot_root**:
   - Change to the container's `rootfs` directory
   - Create a temporary directory (e.g., `.pivot_root`) inside `rootfs` to hold the old root
   - Call `pivot_root` with arguments: new root = `rootfs`, put old = `.pivot_root`
   - Change directory to the new root (`/`)
   - Unmount the old root directory (now at `/.pivot_root`) with `MNT_DETACH`
   - Remove the temporary `.pivot_root` directory
6. **Final Mount Adjustments**:
   - Remount `/` as private if needed (to prevent mount propagation to host)
   - Ensure `/proc` is mounted correctly (remount if needed)
   - Set up any bind mounts specified in container configuration (for volumes)
7. **Change Working Directory**:
   - Change to the working directory specified in the container configuration (or default to `/`)
   - This ensures the process starts in the correct location within the container

Component Responsibilities

The `FilesystemManager` component handles these responsibilities:

Method	Parameters	Returns	Description
`SetupRootfs(image Image, containerID string)`	`image`: Image metadata, `containerID`: Unique container identifier	`string` (path to rootfs), `error`	Extracts image layers and prepares root filesystem for container
`IsolateFilesystem(rootfsPath string)`	`rootfsPath`: Path to prepared root filesystem	`error`	Performs pivot_root and mounts essential filesystems inside container
`CleanupFilesystem(containerID string)`	`containerID`: Container to clean up	`error`	Unmounts and removes container filesystem resources
`MountProc(rootfsPath string)`	`rootfsPath`: Path to root filesystem	`error`	Mounts proc filesystem at rootfsPath/proc with proper options
`MountSys(rootfsPath string)`	`rootfsPath`: Path to root filesystem	`error`	Mounts sysfs at rootfsPath/sys with read-only options

Data Structures for Filesystem Management

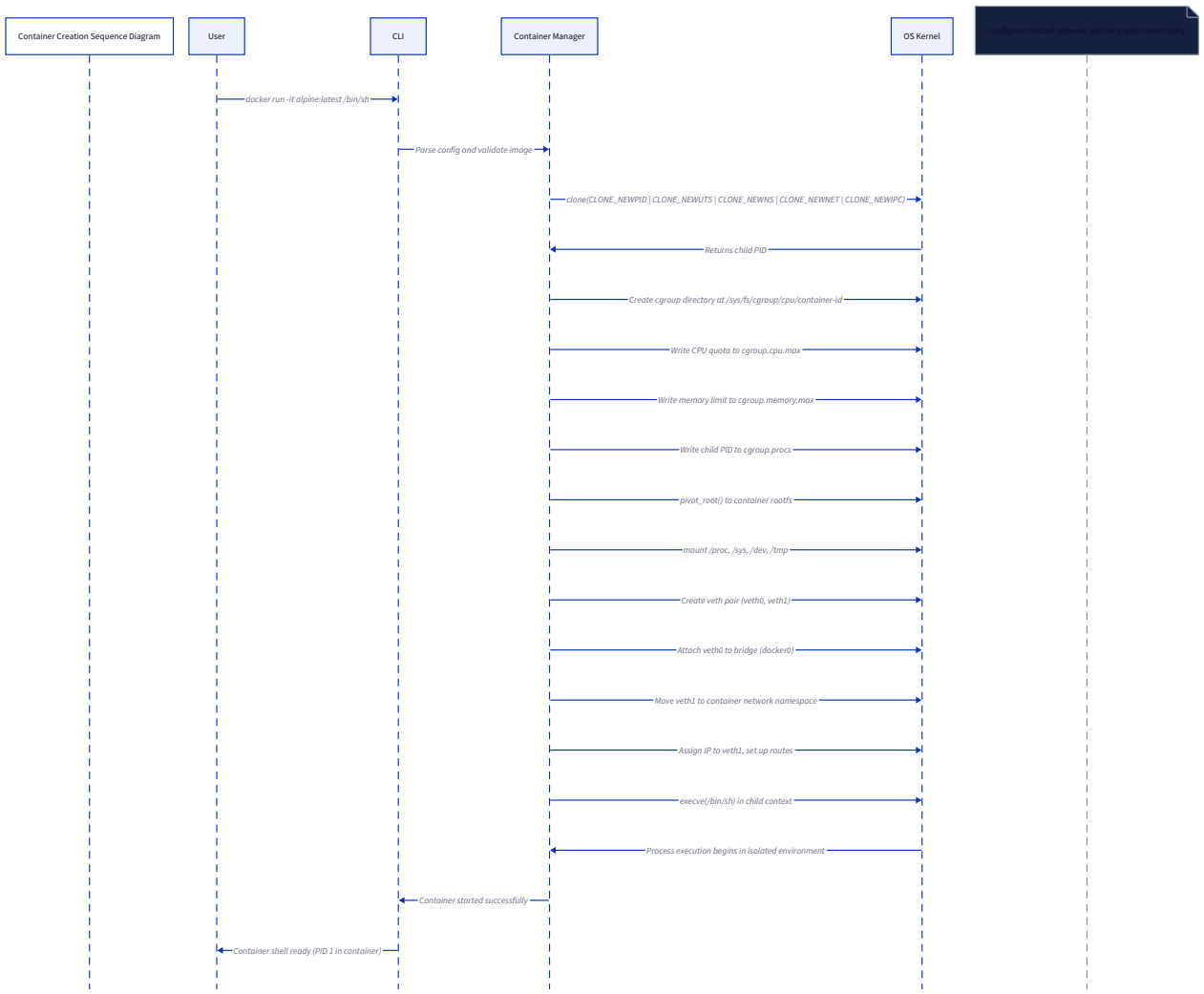
Name	Type	Description
`FilesystemManager`	struct	Manages container filesystem lifecycle
`MountInfo`	struct	Information about a mount operation
`MountConfig`	struct	Configuration for mounting virtual filesystems

The `FilesystemManager` struct will contain:

Field	Type	Description
`basePath`	`string`	Base directory for container storage
`imageStore`	`ImageStore`	Reference to image storage component
`useOverlay`	`bool`	Whether to use OverlayFS (Milestone 4)

Sequence of Operations

Referencing the container creation sequence diagram



, the filesystem isolation occurs at this point in the flow:

1. **User** runs `container run` command with an image name
2. **CLI** parses command and calls `Runtime.CreateContainer()`
3. **Container Manager** creates `Container` record in `StateCreated`
4. **Container Manager** calls `FilesystemManager.SetupRootfs()` to extract image
5. **Container Manager** creates namespaces including `CLONE_NEWNS`
6. **Inside container process**, `FilesystemManager.IsolateFilesystem()` is called
7. **Container process** performs `pivot_root` and mounts `/proc`, `/sys`
8. **Container process** executes the entrypoint command inside isolated rootfs

Security Considerations

1. **Mount Flags**: Always use `nosuid,nodev,noexec` on non-essential mounts to prevent privilege escalation

2. **Read-Only Mounts**: Mount `/sys` and potentially other directories as read-only when possible
3. **Private Propagation**: Set mount propagation to `MS_PRIVATE` to prevent mount leaks between containers
4. **Capabilities**: The container process needs `CAP_SYS_ADMIN` for mount operations, which should be dropped after setup

Example Walkthrough: Alpine Container Startup

Let's trace through a concrete example: starting an Alpine Linux container with `/bin/sh` as the entrypoint:

1. **Image Extraction**: Alpine image layers are extracted to `/var/lib/container-runtime/containers/abc123/rootfs`
2. **Namespace Creation**: Container process created with `CLONE_NEWPID | CLONE_NEWNS | CLONE_NEWUTS`
3. **Mount Setup Inside Container**:
   - Mount new `proc` at `/var/lib/.../abc123/rootfs/proc`
   - Mount new `sysfs` at `/var/lib/.../abc123/rootfs/sys` (read-only)
   - Mount `tmpfs` at `/var/lib/.../abc123/rootfs/tmp`
4. **pivot_root**:
   - `pivot_root("/var/lib/.../abc123/rootfs", "/var/lib/.../abc123/rootfs/.pivot_root")`
   - `chdir("/")`
   - `umount2("/.pivot_root", MNT_DETACH)`
   - `rmdir("/.pivot_root")`
5. **Process Execution**: `execve("/bin/sh", ["/bin/sh"], container_env)`

The container now sees `/` as its Alpine root filesystem, `/proc` shows only container processes, and it cannot access any host files outside its rootfs.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
RootFS Preparation	Direct directory copy	OverlayFS union mount (Milestone 4)
pivot_root Wrapper	Direct syscall.Syscall	Use `github.com/moby/sys/mount` package
Mount Management	Manual mount/unmount calls	Use `golang.org/x/sys/unix` for mount constants
ProcFS Setup	Simple mount with default options	Fine-tuned options (hidepid, gid, etc.)

B. Recommended File/Module Structure

```
project-root/
  cmd/
    byod/                    # CLI entry point
      main.go
  internal/
    container/
      container.go           # Container struct and methods
      store.go               # ContainerStore implementation
    runtime/
      runtime.go             # ContainerRuntime main logic
    filesystem/
      manager.go             # FilesystemManager implementation
      pivot.go               # pivot_root logic
      mounts.go              # Mount/unmount helpers
      overlay.go             # OverlayFS (for Milestone 4)
    namespace/
      manager.go             # NamespaceManager
    cgroup/
      manager.go             # CgroupManager
    network/
      manager.go             # NetworkManager
    image/
      manager.go             # Image handling
  pkg/
    syscall/                 # System call wrappers
      pivot_root.go
      mount.go
```

C. Infrastructure Starter Code

Here's a complete, working `FilesystemManager` skeleton with helper functions:

```go
package filesystem

import (
	"fmt"
	"os"
	"path/filepath"
	"syscall"

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

// FilesystemManager handles container filesystem operations
type FilesystemManager struct {
	basePath   string      // Base directory for container storage
	imageStore ImageStore  // Reference to image storage
}

// NewFilesystemManager creates a new filesystem manager
func NewFilesystemManager(basePath string, imageStore ImageStore) *FilesystemManager {
	return &FilesystemManager{
		basePath:   basePath,
		imageStore: imageStore,
	}
}

// ContainerPath returns the path to a container's directory
func (fm *FilesystemManager) ContainerPath(containerID string) string {
	return filepath.Join(fm.basePath, "containers", containerID)
}

// RootfsPath returns the path to a container's root filesystem
func (fm *FilesystemManager) RootfsPath(containerID string) string {
	return filepath.Join(fm.ContainerPath(containerID), "rootfs")
}

// SetupRootfs extracts image layers and prepares root filesystem
func (fm *FilesystemManager) SetupRootfs(image Image, containerID string) (string, error) {
	containerPath := fm.ContainerPath(containerID)
	rootfsPath := fm.RootfsPath(containerID)

	// Create container directory structure
	if err := os.MkdirAll(rootfsPath, 0755); err != nil {
```

```
                return "", fmt.Errorf("creating container directory: %w", err)
        }

        // TODO 1: Extract image layers to rootfsPath
        // For Milestone 3, you can start with a simple rootfs directory
        // For Milestone 4, implement OverlayFS mounting here

        return rootfsPath, nil
}

// IsolateFilesystem performs pivot_root and mounts essential filesystems
// This function is called from inside the container process after clone()
func (fm *FilesystemManager) IsolateFilesystem(rootfsPath string) error {
        // TODO 2: Mount proc filesystem at rootfsPath/proc
        if err := MountProc(rootfsPath); err != nil {
                return fmt.Errorf("mounting proc: %w", err)
        }

        // TODO 3: Mount sysfs at rootfsPath/sys (read-only)
        if err := MountSys(rootfsPath); err != nil {
                return fmt.Errorf("mounting sys: %w", err)
        }

        // TODO 4: Create /dev directory and basic device nodes
        if err := SetupDev(rootfsPath); err != nil {
                return fmt.Errorf("setting up /dev: %w", err)
        }

        // TODO 5: Perform pivot_root to switch to the new root filesystem
        if err := PivotRoot(rootfsPath); err != nil {
                return fmt.Errorf("pivot_root: %w", err)
        }

        return nil
}

// MountProc mounts proc filesystem at the given path
func MountProc(rootfsPath string) error {
        procPath := filepath.Join(rootfsPath, "proc")
```

```
        if err := os.MkdirAll(procPath, 0755); err != nil {
                return err
        }

        // Mount proc with safe options: nosuid, nodev, noexec
        flags := uintptr(syscall.MS_NOSUID | syscall.MS_NODEV | syscall.MS_NOEXEC)
        return syscall.Mount("proc", procPath, "proc", flags, "")
}

// MountSys mounts sysfs at the given path (read-only)
func MountSys(rootfsPath string) error {
        sysPath := filepath.Join(rootfsPath, "sys")
        if err := os.MkdirAll(sysPath, 0755); err != nil {
                return err
        }

        // Mount sysfs read-only with safe options
        flags := uintptr(syscall.MS_NOSUID | syscall.MS_NODEV | syscall.MS_NOEXEC | syscall.MS_RDONLY)
        return syscall.Mount("sysfs", sysPath, "sysfs", flags, "")
}

// SetupDev creates /dev directory and essential device nodes
func SetupDev(rootfsPath string) error {
        devPath := filepath.Join(rootfsPath, "dev")
        if err := os.MkdirAll(devPath, 0755); err != nil {
                return err
        }

        // Create basic device nodes
        devices := []struct {
                path  string
                major uint32
                minor uint32
                mode  uint32
        }{
                {"/dev/null", 1, 3, syscall.S_IFCHR | 0666},
                {"/dev/zero", 1, 5, syscall.S_IFCHR | 0666},
                {"/dev/random", 1, 8, syscall.S_IFCHR | 0666},
                {"/dev/urandom", 1, 9, syscall.S_IFCHR | 0666},
```

```
        }

        for _, dev := range devices {
                fullPath := filepath.Join(rootfsPath, dev.path)
                if err := syscall.Mknod(fullPath, dev.mode, int(unix.Mkdev(dev.major, dev.minor))); err != nil && !os.IsExist(err) {
                        return fmt.Errorf("creating device node %s: %w", dev.path, err)
                }
        }

        return nil
}

// PivotRoot switches to a new root filesystem using pivot_root syscall
func PivotRoot(rootfs string) error {
        // pivot_root requires both paths to be absolute
        rootfs, err := filepath.Abs(rootfs)
        if err != nil {
                return fmt.Errorf("getting absolute path: %w", err)
        }

        // Create temporary directory for old root
        pivotDir := filepath.Join(rootfs, ".pivot_root")
        if err := os.Mkdir(pivotDir, 0755); err != nil {
                return fmt.Errorf("creating pivot directory: %w", err)
        }

        // Change to rootfs directory
        if err := syscall.Chdir(rootfs); err != nil {
                return fmt.Errorf("chdir to rootfs: %w", err)
        }

        // Call pivot_root system call
        // new_root = rootfs, put_old = pivotDir
        if err := syscall.PivotRoot(rootfs, pivotDir); err != nil {
                return fmt.Errorf("pivot_root system call: %w", err)
        }

        // Change directory to new root
        if err := syscall.Chdir("/"); err != nil {
```

```
                return fmt.Errorf("chdir to new root: %w", err)
        }

        // Unmount old root (now at /.pivot_root)
        oldRoot := "/.pivot_root"
        if err := syscall.Unmount(oldRoot, syscall.MNT_DETACH); err != nil {
                return fmt.Errorf("unmounting old root: %w", err)
        }

        // Remove temporary directory
        if err := os.Remove(oldRoot); err != nil {
                return fmt.Errorf("removing old root directory: %w", err)
        }

        return nil
}

// CleanupFilesystem unmounts and removes container filesystem
func (fm *FilesystemManager) CleanupFilesystem(containerID string) error {
        rootfsPath := fm.RootfsPath(containerID)

        // Try to unmount any mounted filesystems
        mounts := []string{
                filepath.Join(rootfsPath, "proc"),
                filepath.Join(rootfsPath, "sys"),
                filepath.Join(rootfsPath, "dev"),
        }

        for _, mountPoint := range mounts {
                if _, err := os.Stat(mountPoint); err == nil {
                        // Best-effort unmount, ignore errors if not mounted
                        syscall.Unmount(mountPoint, syscall.MNT_DETACH)
                }
        }

        // Remove container directory
        containerPath := fm.ContainerPath(containerID)
        return os.RemoveAll(containerPath)
```

```
}
```

D. Core Logic Skeleton Code

Here's the skeleton for the main container entry point that orchestrates filesystem isolation:

```go
package main


// ChildEntryPoint is the function executed inside the container namespaces

// This is called after clone() from the parent process

func ChildEntryPoint(childFunc ChildFunc) {

        // TODO 1: Parse container configuration from command-line arguments or pipe



        // TODO 2: Setup root filesystem isolation

        // fsManager.IsolateFilesystem(rootfsPath)



        // TODO 3: Set hostname if UTS namespace is isolated

        // syscall.Sethostname([]byte(config.Hostname))



        // TODO 4: Setup container networking if network namespace is isolated

        // networkManager.SetupContainerNetwork()



        // TODO 5: Drop capabilities if running as root

        // Apply security restrictions



        // TODO 6: Execute the container command

        // syscall.Exec(command, args, env)



        // TODO 7: If exec fails, exit with error code

}


// setupContainerFilesystem orchestrates all filesystem isolation steps

func setupContainerFilesystem(containerID string, config ContainerConfig) error {

        // TODO 1: Get FilesystemManager instance



        // TODO 2: Extract image and prepare rootfs if not already done

        // rootfsPath, err := fsManager.SetupRootfs(image, containerID)



        // TODO 3: Mount /proc inside container rootfs

        // fsManager.MountProc(rootfsPath)



        // TODO 4: Mount /sys inside container rootfs (read-only)

        // fsManager.MountSys(rootfsPath)



        // TODO 5: Setup /dev directory with essential device nodes
```

```
        // fsManager.SetupDev(rootfsPath)


        // TODO 6: Apply any bind mounts specified in config


        // TODO 7: Change working directory to container's WorkingDir


        return nil
}
```

E. Language-Specific Hints

1. **Syscall Package**: Use `golang.org/x/sys/unix` for system call constants and `syscall` package for actual calls. Note that Go's `syscall` package is frozen but still works for basic operations.
2. **Mount Flags**: Pay attention to mount flag differences between systems. Use `unix.MS_NOSUID | unix.MS_NODEV | unix.MS_NOEXEC` for safety.
3. **Error Handling**: Many filesystem operations can fail with `EBUSY` if something is still using the filesystem. Use `MNT_DETACH` flag when unmounting to force detachment.
4. **Path Handling**: Always use `filepath.Abs()` to convert to absolute paths before system calls, as many syscalls (including `pivot_root`) require absolute paths.
5. **Cleanup on Failure**: Implement proper cleanup in case of partial failure. Use `defer` statements strategically to clean up temporary directories and mounts.

F. Milestone Checkpoint

After implementing filesystem isolation, you should be able to:

1. **Test Basic Container**:

```
$ sudo ./byod run --rootfs /path/to/alpine-rootfs /bin/sh
```

Should start a shell inside an isolated filesystem.

2. **Verify Isolation**:

```
# Inside container
$ ls /           # Should show container rootfs, not host root
$ ps aux         # Should only show container processes (initially just sh)
$ hostname       # Should be container hostname, not host
```

BASH

3. **Test pivot_root Success**:

```
# From host, check mounts
$ mount | grep container-id
# Should show container's proc mount


# Try to access host files from container
$ sudo ./byod run --rootfs /path/to/alpine-rootfs /bin/sh -c "ls /host"
# Should fail: /host doesn't exist in container
```

BASH

4. **Common Failure Signs**:
   - **"Invalid argument" error**: Likely wrong paths to `pivot_root` (not absolute)

- **"Permission denied"**: Need root privileges for mount operations
- **"No such file or directory"**: Missing essential directories in rootfs
- **/proc shows host processes**: Forgot to mount new proc after pivot_root

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Container can't find `/bin/sh`	Incomplete rootfs or wrong path	Check `ls -la /path/to/rootfs/bin/` from host	Ensure rootfs contains all needed binaries
`pivot_root` returns EINVAL	Paths not absolute or rootfs not a mount point	Print paths before calling pivot_root	Use `filepath.Abs()` and ensure rootfs is a mount
Can still see host files in container	Old root not unmounted or mount propagation issues	Check `mount` output inside container	Ensure unmount old root with `MNT_DETACH`
`/proc` shows host PIDs	Proc not mounted or mounted incorrectly	Check `ls /proc` inside container	Mount proc after pivot_root with correct options
Container exits immediately	Entrypoint binary missing libraries	Run `ldd /path/to/rootfs/bin/sh` from host	Include all shared libraries in rootfs
"Operation not permitted"	Missing `CAP_SYS_ADMIN` capability	Check effective capabilities	Run as root or grant appropriate capabilities

> **Milestone(s):** Milestone 4: Layered Filesystem (OverlayFS)

Component Design: Layered Filesystem (Milestone 4)

This component is responsible for implementing copy-on-write layered filesystems, a cornerstone technology that enables efficient container image storage and sharing. While the previous filesystem isolation component (`FilesystemManager`) provides each container with its own private root filesystem, this component revolutionizes how those root filesystems are constructed by stacking multiple read-only layers and a single writable layer into a unified view, enabling the container to see a complete filesystem while writes are captured separately.

Mental Model: Transparent Overlays on a Projector

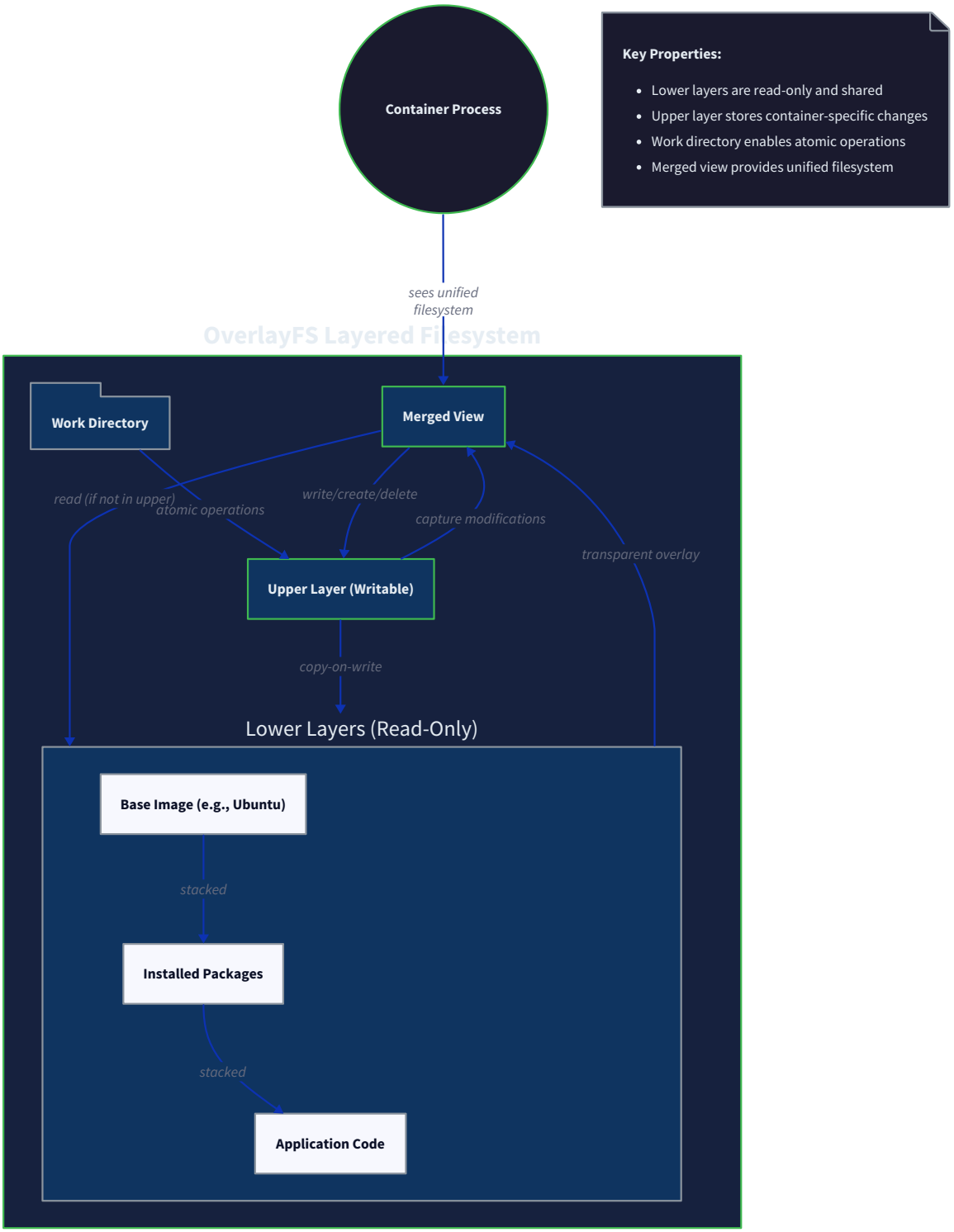
Think of layered filesystems as a **stack of transparent acetate sheets on an overhead projector**. Each sheet (layer) contains drawings (files and directories) that can overlap. When projected together, you see a unified image where upper sheets obscure portions of lower sheets. The crucial property is that you can only draw on the top sheet—any modifications go there while preserving the original drawings on lower sheets for everyone else.

In technical terms:

- **Lower layers** are read-only acetate sheets containing base operating system files, installed packages, and application code.
- **Upper layer** is the writable top sheet where all container modifications (created files, deleted files, modified files) are recorded.
- **Work directory** acts as a temporary scratch pad that the filesystem uses internally to manage atomic operations.
- **Merged view** is the projected image—what the container process actually sees when it looks at its root filesystem.

The critical insight is that **identical lower layers can be shared across hundreds of containers**, just as multiple classrooms can use the same set of base acetate sheets. Each container gets its own writable top sheet for modifications, but the vast majority of data (OS files, libraries) is stored once and referenced many times, achieving tremendous storage efficiency.

Architecture Decision Records: OverlayFS vs Alternatives



Decision: OverlayFS for Layered Filesystem Implementation

Context: We need a union filesystem that supports multiple read-only lower layers and a single writable upper layer with copy-on-write semantics. The solution must be kernel-based (for performance), widely available in modern Linux distributions, and support the OCI image standard's layer model. We must choose between kernel union filesystem implementations and userspace alternatives.

Options Considered:

1. **OverlayFS**: Kernel-based union filesystem merged into Linux mainline since kernel 3.18.
2. **AUFS (Another Union File System)**: Older kernel patchset popular in early Docker versions.
3. **Device Mapper + thin-provisioning**: Block-level snapshot mechanism used by Docker's devicemapper storage driver.
4. **FUSE-based union filesystems (mergerfs, unionfs-fuse)**: Userspace implementations that don't require kernel support.

Decision: We will implement OverlayFS as our layered filesystem technology.

Rationale:

- **Mainstream adoption**: OverlayFS is the default storage driver in Docker and recommended by the OCI community.
- **Kernel integration**: As a kernel filesystem, it offers superior performance over FUSE-based solutions with lower CPU overhead.
- **Minimal dependencies**: Available in stock Linux kernels since 3.18 (2015), requiring no additional kernel modules or patches.
- **Simplicity**: The mount options and directory structure are straightforward, with clear documentation.
- **Copy-on-write efficiency**: Provides true copy-on-write at the file level without block-level complexity.

Consequences:

- **Positive**: Excellent performance, widespread compatibility, and alignment with industry standards.
- **Negative**: Requires Linux kernel ≥3.18 with OverlayFS enabled (CONFIG_OVERLAY_FS). Some advanced features (like recursive directory operations) have edge cases.
- **Maintenance**: OverlayFS behavior is consistent across distributions, reducing platform-specific bugs.

Option	Pros	Cons	Why Not Chosen
OverlayFS	Kernel-native, excellent performance, default in Docker	Requires kernel ≥3.18, some edge cases with rename/delete	**CHOSEN** - Best balance of performance and availability
AUFS	Mature, feature-rich	Not in mainline kernel, requires patching, less maintained	Not in mainline kernel, distribution support inconsistent
Device Mapper	Block-level efficiency, snapshot support	Complex setup, requires LVM/thin-provisioning tools	Overkill for our educational container runtime, complex
FUSE unionfs	Works on any kernel, no special privileges	High CPU overhead, slower performance, userspace complexity	Performance penalty too high for container workloads

Decision: Multiple Lower Layer Support

Context: OCI images consist of multiple layers (often 10-20 for a typical application image). We need to decide how to represent these layers in our OverlayFS mount—either flattening them into a single directory or supporting true multi-layer stacking.

Options Considered:

1. **True multi-layer OverlayFS**: Pass all lower directories as a colon-separated list to the `lowerdir` mount option.
2. **Two-layer flattening**: Extract and merge all read-only layers into a single directory, then use it as the only lower layer.
3. **Incremental mounting**: Mount layers sequentially, with each layer becoming the lowerdir for the next.

Decision: We will implement true multi-layer OverlayFS with colon-separated `lowerdir`.

Rationale:

- **Preserves layer identity**: Each layer remains distinct, enabling efficient layer sharing across containers.
- **Optimized storage**: Identical layers aren't duplicated on disk when multiple images share them.
- **OCI compliance**: Matches the standard image format where layers are applied sequentially.
- **Efficient pull/update**: When pulling updated images, only changed layers need downloading; unchanged layers remain shared.

Consequences:

- **Implementation complexity**: Must manage layer ordering (lowest layer first in list) and handle potentially long `lowerdir` strings.
- **Path length limitations**: Linux has PATH_MAX (4096) limitations for mount options; many layers could exceed this.
- **Performance**: Kernel handles layer traversal efficiently without userspace merging overhead.

Common Pitfalls in OverlayFS Implementation

⚠️ Pitfall 1: Incorrect layer ordering in lowerdir

Description: Placing layers in the wrong order (newest first instead of oldest first) causes files from upper layers to be hidden by lower ones, breaking the image.

Why it's wrong: OverlayFS renders files from the **last** lowerdir entry as the "lowest" layer. The base OS layer must be first in the list, with application layers appended in the order they were added during image build.

How to fix: Always order layers chronologically from base to top. When parsing an OCI image manifest, layers are listed in application order—the first in the list is applied to the base, so it should be nearest to the upper layer in `lowerdir`.

```
# CORRECT: base layer first, then intermediate layers, then top read-only layer last
lowerdir=/layers/layer4:/layers/layer3:/layers/layer2:/layers/layer1

# WRONG: reversed order hides application files under OS files
lowerdir=/layers/layer1:/layers/layer2:/layers/layer3:/layers/layer4
```

⚠️ Pitfall 2: Forgetting the work directory

Description: Omitting the `workdir` mount option or pointing it to a non-empty directory.

Why it's wrong: OverlayFS requires a dedicated `workdir` for atomic rename operations during copy-up. If absent or non-empty, the mount fails with "invalid argument."

How to fix: Always create an empty `workdir` adjacent to the `upperdir`. Use a unique subdirectory per container to prevent collisions.

```go
// Create work directory alongside upper

workDir := filepath.Join(containerDir, "work")

if err := os.MkdirAll(workDir, 0755); err != nil { ... }
```

⚠️ Pitfall 3: Not handling whiteouts properly

Description: When a container deletes a file from a lower layer, OverlayFS creates a "whiteout" (character device with major/minor 0/0) in the upper layer. Userspace tools must interpret these when exporting layers.

Why it's wrong: If you later commit the container's upper layer to create a new image layer, you must convert whiteout devices back to OCI whiteout format (`.wh.<filename>` files) for compatibility with other tools.

How to fix: When traversing the upper layer for layer creation, check for device files with `stat.Rdev == 0`. Convert them to whiteout entries in the layer tar archive.

⚠️ Pitfall 4: Mount propagation leaking between containers

Description: Without proper mount namespace isolation, an OverlayFS mount in one container might become visible in another container's filesystem view.

Why it's wrong: This violates filesystem isolation—containers could see each other's OverlayFS mount points or, worse, modify them.

How to fix: Ensure the container process runs in its own mount namespace (`CLONE_NEWNS`). Set mount propagation to `MS_PRIVATE` before creating OverlayFS mounts so they don't propagate to other namespaces.

```go
// In the container setup, before mounting OverlayFS

syscall.Mount("none", "/", "", syscall.MS_PRIVATE|syscall.MS_REC, "")
```

⚠️ Pitfall 5: Open file handles surviving unmount

Description: If a process inside the container holds open file descriptors to files in the OverlayFS, unmounting will fail with "device busy."

Why it's wrong: During container cleanup, you need to unmount the OverlayFS to release resources. Open files prevent this, causing resource leaks.

How to fix: Before unmounting, ensure all processes in the container namespace have terminated. Use `MNT_DETACH` flag for lazy unmount as a fallback, but prefer proper process termination.

```go
// Force unmount with detach if regular unmount fails

if err := syscall.Unmount(mountPath, 0); err != nil {

    syscall.Unmount(mountPath, syscall.MNT_DETACH)

}
```

Implementation Guidance for Layered Filesystem

This section bridges the design concepts to concrete Go implementation, providing the scaffolding and key algorithms for the layered filesystem component.

Technology Recommendations Table

Component	Simple Option	Advanced Option
Union Filesystem	OverlayFS with `mount` syscall	Multiple backing stores (btrfs, zfs snapshots)
Layer Storage	Plain directories with extracted tar layers	Content-addressable storage with deduplication
Image Layer Cache	In-memory map of layer digests to paths	Database with reference counting and GC
Whiteout Handling	Basic device node detection	Full OCI whiteout spec with opaque directories

Recommended File/Module Structure

Extend the existing project structure with overlay-specific modules:

```
project-root/
├── cmd/
│   └── byod/                    # CLI entry point
├── internal/
│   ├── container/               # Container management
│   ├── cgroup/                  # Resource limits
│   ├── namespace/               # Namespace isolation
│   ├── network/                 # Networking
│   ├── image/                   # Image handling (from Milestone 6)
│   └── storage/                 # NEW: Storage and layered filesystems
│       ├── storage.go           # Storage interface definitions
│       ├── overlay.go           # OverlayFS implementation
│       ├── layer.go             # Layer management
│       ├── cache.go             # Layer caching
│       └── whiteout.go          # Whiteout handling utilities
└── pkg/
    └── oci/                     # OCI image spec types
```

Infrastructure Starter Code

Complete OverlayFS Mount Helper (ready to use):

```go
// internal/storage/overlay.go

package storage


import (

	"fmt"

	"os"

	"path/filepath"

	"strings"

	"syscall"


	"github.com/byod/pkg/oci"

)


// OverlayManager handles OverlayFS mounts for container root filesystems

type OverlayManager struct {

	storageBase string  // Base directory for layer storage (e.g., /var/lib/byod/storage)

}


// NewOverlayManager creates a new OverlayFS manager

func NewOverlayManager(storageBase string) (*OverlayManager, error) {

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

		return nil, fmt.Errorf("creating storage base: %w", err)

	}

	return &OverlayManager{storageBase: storageBase}, nil

}


// MountOverlay creates an OverlayFS mount for a container

// Returns the path to the merged directory (container's rootfs view)

func (om *OverlayManager) MountOverlay(containerID string, lowerDirs []string, upperDir, workDir string) (string, error) {

	// Create necessary directories

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

		return "", fmt.Errorf("creating upper dir: %w", err)

	}

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

		return "", fmt.Errorf("creating work dir: %w", err)

	}



	mergedDir := filepath.Join(filepath.Dir(upperDir), "merged")

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

		return "", fmt.Errorf("creating merged dir: %w", err)

	}
```

```
        // Build lowerdir string (oldest layer first)
        lowerdir := strings.Join(lowerDirs, ":")

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

        // Mount OverlayFS
        if err := syscall.Mount("overlay", mergedDir, "overlay", 0, options); err != nil {
                return "", fmt.Errorf("mounting overlayfs: %w", err)
        }

        return mergedDir, nil
}

// UnmountOverlay unmounts and cleans up OverlayFS mount
func (om *OverlayManager) UnmountOverlay(mergedDir string) error {
        // Try normal unmount first
        if err := syscall.Unmount(mergedDir, 0); err != nil {
                // Fallback to lazy unmount if busy
                syscall.Unmount(mergedDir, syscall.MNT_DETACH)
        }

        // Clean up merged directory (safe after unmount)
        os.RemoveAll(mergedDir)
        return nil
}

// GetLayerPath returns the path where a layer with given digest is stored
func (om *OverlayManager) GetLayerPath(digest string) string {
        // Convert digest to safe filesystem path
        // OCI digests are like "sha256:abc123..." - use just the hash part
        parts := strings.SplitN(digest, ":", 2)
        hash := parts[1] if len(parts) > 1 else digest
        return filepath.Join(om.storageBase, "layers", hash[:2], hash) // Sharding by first 2 chars
}
```

Layer Cache with Reference Counting (ready to use):

```go
// internal/storage/cache.go

package storage


import (

        "sync"

)


// LayerCache tracks which layers are in use by containers

type LayerCache struct {

        mu     sync.RWMutex

        layers map[string]int // digest -> reference count

}


// NewLayerCache creates a new layer cache

func NewLayerCache() *LayerCache {

        return &LayerCache{

                layers: make(map[string]int),

        }

}


// AddReference increments reference count for a layer

func (lc *LayerCache) AddReference(digest string) {

        lc.mu.Lock()

        defer lc.mu.Unlock()

        lc.layers[digest]++

}


// ReleaseReference decrements reference count, returns true if count reaches zero

func (lc *LayerCache) ReleaseReference(digest string) bool {

        lc.mu.Lock()

        defer lc.mu.Unlock()



        if count, exists := lc.layers[digest]; exists {

                if count <= 1 {

                        delete(lc.layers, digest)

                        return true // Layer is no longer referenced

                }

                lc.layers[digest] = count - 1

        }

        return false

}
```

```
// IsLayerCached checks if a layer exists in the cache
func (lc *LayerCache) IsLayerCached(digest string) bool {
        lc.mu.RLock()
        defer lc.mu.RUnlock()
        _, exists := lc.layers[digest]
        return exists
}
```

Core Logic Skeleton Code

Layer Extraction and Preparation (learner implements):

GO

```go
// internal/storage/layer.go

package storage


import (
        "archive/tar"
        "compress/gzip"
        "fmt"
        "io"
        "os"
        "path/filepath"

        "github.com/byod/pkg/oci"
)


// ExtractLayer extracts a compressed tar layer to the destination directory
// layerPath: path to the .tar.gz layer file
// destDir: directory to extract contents to
// Returns the digest of the extracted layer for verification
func ExtractLayer(layerPath, destDir string) (string, error) {
        // TODO 1: Open the layer file for reading
        // Hint: os.Open(layerPath)


        // TODO 2: Create a gzip reader to decompress the tar
        // Hint: gzip.NewReader(file)


        // TODO 3: Create a tar reader from the gzip reader
        // Hint: tar.NewReader(gzipReader)


        // TODO 4: Iterate through tar headers
        // for {
        //   header, err := tarReader.Next()
        //   if err == io.EOF { break }
        //   if err != nil { return "", err }


        // TODO 5: For each header, construct the full destination path
        // targetPath := filepath.Join(destDir, header.Name)


        // TODO 6: Handle different tar entry types:
        //   - TypeDir: Create directory with header.Mode permissions
        //   - TypeReg/TypeRegA: Create file, copy contents from tarReader
```

```
        //   - TypeSymlink: Create symbolic link
        //   - TypeLink: Create hard link
        //   - TypeChar/TypeBlock: Create device node with header.Devmajor/Devminor

        // TODO 7: Set proper ownership (header.Uid, header.Gid) using syscall.Chown
        // Note: This may require root privileges or user namespace setup

        // TODO 8: Set modification time from header.ModTime

        // TODO 9: Handle whiteout files (header.Name starts with ".wh.")
        //   - Remove the corresponding file/directory from destination
        //   - For ".wh..wh..opq" (opaque directory), mark directory as opaque

        // TODO 10: Close all readers and return nil if successful

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

// PrepareLayersForContainer prepares all layers for a container's rootfs
// image: The OCI image containing layer descriptors
// containerID: Unique container identifier for storage paths
// Returns: paths to lower directories (oldest first), upper dir, work dir, error
func (om *OverlayManager) PrepareLayersForContainer(image *oci.Image, containerID string) ([]string, string, string, error) {
        containerDir := filepath.Join(om.storageBase, "containers", containerID)

        // TODO 1: Create container directories: upper, work, layers
        // upperDir := filepath.Join(containerDir, "upper")
        // workDir := filepath.Join(containerDir, "work")
        // layersDir := filepath.Join(containerDir, "layers")

        // TODO 2: For each layer in image.Layers (in order they appear in manifest):
        //   a. Check if layer already extracted in cache (use GetLayerPath)
        //   b. If not cached, download layer (from image store) and extract to cache
        //   c. Add layer reference to LayerCache
        //   d. Add layer path to lowerDirs slice

        // TODO 3: IMPORTANT: Reverse the lowerDirs slice for OverlayFS
        // OverlayFS expects oldest layer first, but OCI lists layers in application order
        // So layer[0] is applied on top of base, should be last in lowerdir list
```

```
        // TODO 4: Return lowerDirs (reversed), upperDir, workDir, nil


        return nil, "", "", fmt.Errorf("not implemented")
}
```

Updated FilesystemManager Integration (learner implements):

GO

```go
// internal/container/filesystem.go

package container


import (

        "fmt"

        "path/filepath"


        "github.com/byod/internal/storage"

        "github.com/byod/pkg/oci"

)


// setupLayeredRootfs sets up a container rootfs using OverlayFS

func (fm *FilesystemManager) setupLayeredRootfs(image *oci.Image, containerID string) (string, error) {

        // TODO 1: Initialize OverlayManager if not already done

        // if fm.overlayManager == nil {

        //   fm.overlayManager = storage.NewOverlayManager(fm.basePath)

        // }



        // TODO 2: Prepare layers using OverlayManager.PrepareLayersForContainer

        // lowerDirs, upperDir, workDir, err := fm.overlayManager.PrepareLayersForContainer(image, containerID)



        // TODO 3: Mount OverlayFS to create merged view

        // mergedDir, err := fm.overlayManager.MountOverlay(containerID, lowerDirs, upperDir, workDir)



        // TODO 4: Mount essential filesystems inside the merged directory

        //   - Mount /proc at mergedDir/proc

        //   - Mount /sys at mergedDir/sys (read-only)

        //   - Create /dev entries in mergedDir/dev



        // TODO 5: Return path to merged directory (will become container's rootfs)



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

}


// cleanupLayeredRootfs cleans up OverlayFS mount and references

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

        // TODO 1: Unmount essential filesystems (/proc, /sys, /dev) inside container rootfs



        // TODO 2: Get path to merged directory

        // mergedDir := filepath.Join(fm.basePath, "containers", containerID, "merged")
```

```
        // TODO 3: Unmount OverlayFS using OverlayManager.UnmountOverlay


        // TODO 4: Release references to all layers in LayerCache


        // TODO 5: Clean up container directories (upper, work, layers)
        // Note: Don't delete cached layers - they might be used by other containers


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

Language-Specific Hints

1. **Mount Syscall**: Use `syscall.Mount("overlay", target, "overlay", 0, options)` where options is a comma-separated string like `"lowerdir=/a:/b,upperdir=/c,workdir=/d"`.
2. **Tar Extraction**: Use `archive/tar` and `compress/gzip` packages. Remember to handle symlinks and device files specially—regular file copying won't work for them.
3. **Whiteout Detection**: Check if tar header name starts with `.wh.`. The OCI spec uses `.wh.<filename>` for whiteouts and `.wh..wh..opq` for opaque directories.
4. **Path Length Limits**: Linux kernel has a 4096-byte limit for mount options. If your `lowerdir` string exceeds this, consider using the `volatile` lowerdir feature (kernel 4.13+) or workaround with intermediate mounts.
5. **Error Handling**: OverlayFS mount can fail with specific errno values:
   - `EINVAL`: Invalid options (check lowerdir/upperdir/workdir paths exist)
   - `ENODEV`: OverlayFS not enabled in kernel (`CONFIG_OVERLAY_FS`)
   - `EBUSY`: Mount point is busy (unmount previous mounts first)

Milestone Checkpoint

After implementing the layered filesystem component, verify it works correctly:

Test Command:

```bash
# Build and run a test that creates a container with layered rootfs
go test ./internal/storage/... -v -run TestOverlayFS
```

Expected Behavior:

1. An OCI image with multiple layers should be correctly assembled into a single rootfs.
2. Files from upper layers should override files from lower layers with the same path.
3. Creating a file in the container should appear only in the upper layer directory.
4. Deleting a file from a lower layer should create a whiteout in the upper layer.
5. After container cleanup, the merged directory should unmount successfully.

Manual Verification:

```bash
# 1. Create a container with an alpine image
sudo ./byod run --name test-layers alpine:latest /bin/sh

# 2. Inside container, create a file
echo "test" > /myfile.txt

# 3. Exit container and inspect layers
ls -la /var/lib/byod/storage/containers/test-layers/
# Should show: upper/, work/, merged/, layers/

# 4. Check upper layer contains the new file
ls -la /var/lib/byod/storage/containers/test-layers/upper/
# Should show myfile.txt

# 5. Check lower layers remain unchanged
```

Signs of Problems:

- Container fails to start with "invalid argument": Check OverlayFS mount options format.
- Files from base layer not visible: Lower layer order is reversed.
- "Device busy" on cleanup: Processes still running in container namespace.
- Permission errors: User namespace mapping incorrect or extraction ownership issues.

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Mount fails with "invalid argument"	Missing work directory or non-empty workdir	Check `dmesg | tail` for kernel messages	Create empty work directory before mount
Files from application layer not visible	Lowerdir order incorrect	Print lowerdir string, verify chronological order	Reverse the lowerdir list (oldest first)
"Operation not permitted" on mount	Missing capabilities or user namespace	Check `cat /proc/self/status | grep CapEff`	Run as root or grant CAP_SYS_ADMIN
Container sees wrong file contents	Upper layer not properly separated	Check if upperdir is shared between containers	Use unique upperdir per container
Unmount fails with "device busy"	Open file handles in container	`lsof +D /var/lib/byod/containers/<id>/merged`	Ensure all container processes terminated
Whiteouts appear as character devices	Upper layer not interpreting whiteouts	Check `ls -l` in upperdir for device files with 0,0	Implement whiteout handling in layer extraction
Copy-up causes high CPU	Many small files copying up	Monitor with `inotifywait -rm /merged`	Consider tuning overlay kernel module params

> **Milestone(s):** Milestone 5: Container Networking

Component Design: Container Networking (Milestone 5)

Container networking is the component that provides each container with an isolated network environment while enabling controlled connectivity to the host, other containers, and external networks. This involves creating a private network stack, establishing virtual network links, and configuring routing and firewall rules. The key architectural challenge is to provide useful network connectivity while maintaining the security and isolation guarantees of containers.

Mental Model: Private Phone Lines with a Switchboard

Imagine an office building with private phone lines. Each office (container) has its own private telephone network with a unique phone number and internal extensions. These private networks are connected to a central switchboard (bridge) in the building's telecom room (host). The switchboard can:

1. **Connect offices internally**: When an office calls another office's extension, the switchboard routes the call directly between their private lines.
2. **Route external calls**: When an office calls an outside number, the switchboard forwards the call through the building's main trunk line (host's physical interface), making it appear as if the call originated from the building's main number (host IP).
3. **Receive incoming calls**: The switchboard can route calls from the main trunk to specific offices by mapping the building's main number plus an extension (host port) to an office's internal line (container port).

The private phone lines are **virtual ethernet (veth) pairs**—twin cables where one end plugs into the office's phone system and the other into the switchboard. The switchboard is a **Linux bridge**—a virtual switch that connects multiple veth ends. The building's firewall rules (**iptables**) determine which calls are allowed and how they're translated.

This model illustrates the complete isolation (each office has its own phone system), controlled connectivity (via the switchboard), and network address translation (external calls appear from the building) that container networking provides.

Architecture Decision Records: Network Topology

Decision: Bridge Network with NAT for Default Connectivity

Context: Containers need network isolation but also practical connectivity. We must choose a default network topology that balances isolation, ease of use, and compatibility with existing Docker workflows.

Options Considered:

1. **Host network mode**: Containers share the host's network namespace, providing maximum performance but zero network isolation.
2. **Bridge network with NAT**: Each container gets its own network namespace, connected via veth pair to a Linux bridge, with NAT for external connectivity (Docker's default).
3. **Macvlan**: Containers get direct MAC addresses on the physical network, appearing as separate physical devices but requiring switch configuration.
4. **None**: No network connectivity at all—complete isolation.

Decision: Bridge network with NAT as the default topology.

Rationale:

- **Isolation**: Each container gets a full isolated network stack (own interfaces, routing, iptables).
- **Practicality**: NAT allows containers to access external networks without requiring public IPs for each container.
- **Port mapping**: Host port forwarding enables services inside containers to be accessible from outside the host.
- **Inter-container communication**: Containers on the same bridge can communicate directly via their private IPs.
- **Established pattern**: Matches Docker's default `bridge` driver, providing familiar behavior for users.

Consequences:

- **Performance overhead**: Extra hop through bridge and NAT layer introduces minor latency.
- **IP address management**: Requires IPAM (IP Address Management) to allocate unique IPs to containers.
- **NAT complexities**: Requires iptables rules for masquerading and port forwarding.
- **Bridge setup**: Requires creating and managing a Linux bridge device.

Option	Pros	Cons	Chosen?
Host network	Maximum performance, no setup required	Zero network isolation, port conflicts	No
Bridge with NAT	**Good isolation, external access via NAT, port mapping, inter-container communication**	**NAT overhead, IPAM required, bridge setup**	**Yes**
Macvlan	Direct physical network access, no NAT overhead	Requires network support, no port mapping, less isolation	No
None	Maximum isolation	No network connectivity	No

Decision: veth Pair for Container-Host Connectivity

Context: We need to connect the container's isolated network namespace to the host's network namespace to provide connectivity.

Options Considered:

1. **veth pair**: Virtual Ethernet device pair—one end in container, one end in host, connected like a virtual network cable.
2. **macvlan**: Assigns a unique MAC address to container, attaching directly to physical interface.
3. **ipvlan**: Similar to macvlan but shares MAC address, differing only in IP configuration.

Decision: veth pair.

Rationale:

- **Simplicity**: Well-understood, stable Linux primitive.
- **Bridge compatibility**: veth endpoints can be added to Linux bridges easily.
- **Isolation**: Complete layer 2 separation between container and host.
- **Debugging**: Easy to inspect with `ip link` on both ends.

Consequences:

- Requires creating and configuring two devices per container.
- Must manage lifecycle (create, move to namespace, delete).
- Bridge acts as a virtual switch connecting multiple veth ends.

Decision: iptables for NAT and Firewalling

Context: Containers with private IPs need to communicate with external networks, and we need to forward host ports to container ports.

Options Considered:

1. **iptables**: Traditional Linux firewall/NAT tool using netfilter hooks.
2. **nftables**: Newer replacement for iptables with unified syntax.
3. **Userspace proxy**: Proxy connections in userspace (like Docker's legacy mode).

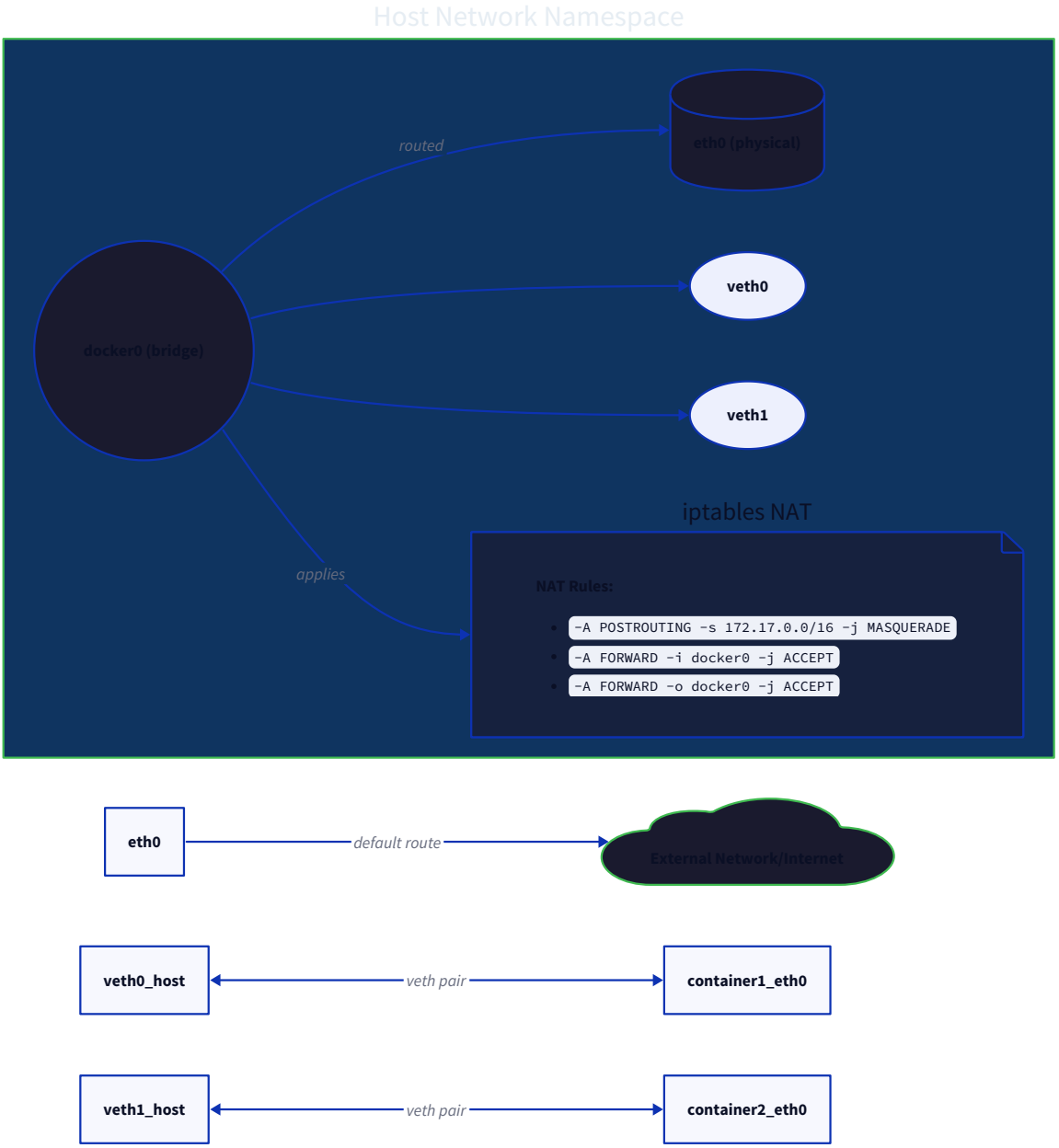
Decision: iptables.

Rationale:

- **Ubiquity**: Available on virtually all Linux systems.
- **Docker compatibility**: Docker uses iptables, ensuring consistent behavior.
- **Performance**: Kernel-space packet processing, minimal overhead.
- **Mature**: Well-documented with extensive community knowledge.

Consequences:

- Complex rule management (must add/remove rules per container).
- Potential conflicts with existing host iptables rules.
- Requires understanding of netfilter chains and traversal order.


Host Network Namespace
routed
eth0 (physical)
docker0 (bridge)
veth0
veth1
iptables NAT
applies
NAT Rules:
-A POSTROUTING -s 172.17.0.0/16 -j MASQUERADE
-A FORWARD -i docker0 -j ACCEPT
-A FORWARD -o docker0 -j ACCEPT
eth0
default route
External Network/Internet
veth0_host
veth pair
container1_eth0
veth1_host
veth pair
container2_eth0
Container 1 Network Namespace
eth0
network
Container Process
Container 2 Network Namespace
eth0
network
Container Process

Common Pitfalls in Container Networking

⚠️ Pitfall: DNS Resolution Failures Inside Container

Description: The container can ping IP addresses but cannot resolve hostnames (e.g., `curl https://google.com` fails). This occurs because the container's `/etc/resolv.conf` is either missing, empty, or contains invalid DNS servers.

Why it's wrong: Network connectivity without DNS is severely limited—most applications rely on hostname resolution.

How to fix:

1. Copy the host's `/etc/resolv.conf` into the container rootfs (but note this may leak host DNS configuration).
2. Use a known public DNS server like `8.8.8.8` (Google) or `1.1.1.1` (Cloudflare).
3. Better: Detect the host's DNS configuration via `systemd-resolve` or `/run/systemd/resolve/resolv.conf` and use those nameservers.

Prevention: Always configure `/etc/resolv.conf` as part of container filesystem setup. Use a template that includes fallback nameservers.

⚠️ Pitfall: iptables Rules Blocking Inter-Container Communication

Description: Containers on the same bridge cannot ping each other, though they have IP addresses on the same subnet. This often happens because the host's firewall rules (particularly in the `FORWARD` chain) are dropping packets between bridge interfaces.

Why it's wrong: The bridge device forwards at layer 2, but iptables filters at layer 3. Default policies may block forwarded traffic.

How to fix:

1. Ensure `net.ipv4.ip_forward=1` is set on the host.
2. Add iptables rules to allow forwarded traffic between bridge interfaces: `iptables -I FORWARD -i br0 -o br0 -j ACCEPT`.
3. Check that the bridge's `FORWARD` chain policy is `ACCEPT`.

Prevention: During bridge creation, configure iptables rules to allow traffic between containers on the same bridge and set appropriate forward policies.

⚠️ Pitfall: Host Port Conflict on Container Start

Description: Starting a container with port mapping fails with "bind: address already in use" because the host port is already occupied by another process.

Why it's wrong: The container runtime should detect this before attempting to start the container and provide a clear error message.

How to fix:

1. Before setting up port forwarding, check if the host port is available by attempting to listen on it (with `SO_REUSEADDR`).
2. If occupied, either fail with a clear error or automatically select an alternative port (if configured to do so).

Prevention: Implement port availability checking in the `NetworkManager.SetupNetwork` method before adding iptables rules.

⚠️ Pitfall: Orphaned Network Resources After Container Crash

Description: When a container crashes or is killed, its network namespace and veth pair may remain on the host, accumulating over time and causing resource leaks.

Why it's wrong: Orphaned network namespaces consume kernel resources and may interfere with subsequent containers (e.g., IP address conflicts).

How to fix:

1. Always implement cleanup in a `defer` statement or signal handler.
2. On container start, record the network namespace handle and veth pair names for later cleanup.
3. During container removal, forcefully delete the network namespace if it still exists (`ip netns delete <ns>`).

Prevention: Design the `NetworkManager` to track created resources and provide a `CleanupNetwork` method that is called during container removal, even after failures.

⚠️ Pitfall: MTU Mismatch Causing Packet Fragmentation

Description: Network performance is poor, with high latency and dropped packets. This can occur when the MTU (Maximum Transmission Unit) of the veth interface differs from the host's physical interface or bridge.

Why it's wrong: Packets exceeding the MTU will be fragmented (reducing performance) or dropped.

How to fix:

1. Set the MTU on veth interfaces to match the host's physical interface (typically 1500 for Ethernet).
2. Consider lowering MTU if using overlay networks or VPNs.

Prevention: During veth creation, explicitly set the MTU based on the host's default route interface MTU.

Implementation Guidance for Network Setup

This section provides concrete implementation guidance for the networking component. The primary challenge is orchestrating multiple Linux networking primitives while handling errors and cleanup properly.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Network Namespace Creation	`unshare(CLONE_NEWNET)` via Go's `syscall` package	`ip netns add` via exec for better tool compatibility
Interface Management	`netlink` library (e.g., `github.com/vishvananda/netlink`)	Raw syscalls via `syscall` package
Bridge Creation	`netlink` library or `ip link add br0 type bridge` via exec	Direct ioctl calls
iptables Rules	`exec.Command("iptables", ...)`	`github.com/coreos/go-iptables/iptables` library
Port Forwarding	iptables DNAT rules	Userspace proxy (like socat) for complex scenarios
IP Address Management	Static allocation from a predefined subnet	DHCP server or IPAM library

B. Recommended File/Module Structure

Add networking-specific files to the project structure:

```
project-root/
├── cmd/
│   └── byod/                       # CLI entry point
├── internal/
│   ├── container/                  # Container management
│   ├── cgroup/                     # cgroup management
│   ├── image/                      # Image handling
│   ├── filesystem/                 # Filesystem isolation
│   ├── network/                    # NEW: Network component
│   │   ├── manager.go              # NetworkManager implementation
│   │   ├── bridge.go               # Bridge creation and management
│   │   ├── veth.go                 # veth pair operations
│   │   ├── iptables.go             # iptables rule management
│   │   ├── ipam.go                 # IP Address Management
│   │   └── network_test.go         # Network tests
│   └── runtime/                    # Main runtime coordination
└── pkg/
    └── netutil/                    # Reusable networking utilities
```

C. Infrastructure Starter Code

Complete `netlink` wrapper for basic operations (import and use as-is):

GO

```go
// internal/network/netlink_helper.go

package network


import (

        "fmt"

        "net"

        "runtime"


        "github.com/vishvananda/netlink"

        "github.com/vishvananda/netns"

)


// NetlinkHelper provides convenience methods for common network operations

type NetlinkHelper struct{}


// NewNetlinkHelper creates a new helper instance

func NewNetlinkHelper() *NetlinkHelper {

        return &NetlinkHelper{}

}


// CreateBridge creates a Linux bridge with the given name and IP address

func (h *NetlinkHelper) CreateBridge(name string, ipNet *net.IPNet) error {

        // Check if bridge already exists

        _, err := netlink.LinkByName(name)

        if err == nil {

                return fmt.Errorf("bridge %s already exists", name)

        }


        // Create bridge attributes

        la := netlink.NewLinkAttrs()

        la.Name = name

        bridge := &netlink.Bridge{LinkAttrs: la}


        // Add bridge

        if err := netlink.LinkAdd(bridge); err != nil {

                return fmt.Errorf("failed to create bridge %s: %v", name, err)

        }


        // Set bridge UP

        if err := netlink.LinkSetUp(bridge); err != nil {

                // Clean up on failure

                netlink.LinkDel(bridge)
```

```
                return fmt.Errorf("failed to set bridge %s up: %v", name, err)
        }

        // Assign IP address if provided
        if ipNet != nil {
                addr := &netlink.Addr{IPNet: ipNet}
                if err := netlink.AddrAdd(bridge, addr); err != nil {
                        netlink.LinkDel(bridge)
                        return fmt.Errorf("failed to add IP %s to bridge %s: %v", ipNet.String(), name, err)
                }
        }

        return nil
}

// CreateVethPair creates a veth pair with the given names
func (h *NetlinkHelper) CreateVethPair(hostVethName, containerVethName string, mtu int) error {
        // Create veth pair attributes
        veth := &netlink.Veth{
                LinkAttrs: netlink.LinkAttrs{
                        Name: hostVethName,
                        MTU:  mtu,
                },
                PeerName: containerVethName,
        }

        // Add veth pair
        if err := netlink.LinkAdd(veth); err != nil {
                return fmt.Errorf("failed to create veth pair %s-%s: %v", hostVethName, containerVethName, err)
        }

        // Set host veth UP
        hostVeth, err := netlink.LinkByName(hostVethName)
        if err != nil {
                netlink.LinkDel(veth)
                return fmt.Errorf("failed to find host veth %s: %v", hostVethName, err)
        }
        if err := netlink.LinkSetUp(hostVeth); err != nil {
                netlink.LinkDel(veth)
                return fmt.Errorf("failed to set host veth %s up: %v", hostVethName, err)
        }
```

```
        return nil
}

// MoveInterfaceToNamespace moves a network interface to the specified network namespace
func (h *NetlinkHelper) MoveInterfaceToNamespace(ifaceName string, nsPath string) error {
        // Get the interface
        iface, err := netlink.LinkByName(ifaceName)
        if err != nil {
                return fmt.Errorf("failed to find interface %s: %v", ifaceName, err)
        }

        // Get the target namespace
        nsHandle, err := netns.GetFromPath(nsPath)
        if err != nil {
                return fmt.Errorf("failed to get namespace from path %s: %v", nsPath, err)
        }
        defer nsHandle.Close()

        // Move the interface
        if err := netlink.LinkSetNsFd(iface, int(nsHandle)); err != nil {
                return fmt.Errorf("failed to move interface %s to namespace: %v", ifaceName, err)
        }

        return nil
}

// ConfigureInterfaceInNamespace configures an interface inside a network namespace
// Must be called from within the namespace context
func (h *NetlinkHelper) ConfigureInterfaceInNamespace(ifaceName string, ipNet *net.IPNet, gateway net.IP) error {
        // Get the interface
        iface, err := netlink.LinkByName(ifaceName)
        if err != nil {
                return fmt.Errorf("failed to find interface %s in namespace: %v", ifaceName, err)
        }

        // Set interface UP
        if err := netlink.LinkSetUp(iface); err != nil {
                return fmt.Errorf("failed to set interface %s up: %v", ifaceName, err)
        }

        // Add IP address
```

```
        if ipNet != nil {
                addr := &netlink.Addr{IPNet: ipNet}
                if err := netlink.AddrAdd(iface, addr); err != nil {
                        return fmt.Errorf("failed to add IP %s to interface %s: %v", ipNet.String(), ifaceName, err)
                }
        }

        // Add default route if gateway provided
        if gateway != nil {
                defaultRoute := &netlink.Route{
                        Dst: nil, // Default route
                        Gw:  gateway,
                }
                if err := netlink.RouteAdd(defaultRoute); err != nil {
                        return fmt.Errorf("failed to add default route via %s: %v", gateway.String(), err)
                }
        }

        return nil
}

// LockOSThreadAndExecute locks the OS thread and executes fn in the specified namespace
func (h *NetlinkHelper) LockOSThreadAndExecute(nsPath string, fn func() error) error {
        // Lock the goroutine to its current OS thread
        runtime.LockOSThread()
        defer runtime.UnlockOSThread()

        // Get current namespace
        originalNS, err := netns.Get()
        if err != nil {
                return fmt.Errorf("failed to get current namespace: %v", err)
        }
        defer originalNS.Close()

        // Get target namespace
        targetNS, err := netns.GetFromPath(nsPath)
        if err != nil {
                return fmt.Errorf("failed to get target namespace %s: %v", nsPath, err)
        }
        defer targetNS.Close()
```

```
        // Switch to target namespace
        if err := netns.Set(targetNS); err != nil {
                return fmt.Errorf("failed to enter namespace %s: %v", nsPath, err)
        }

        // Execute function
        err = fn()

        // Switch back to original namespace
        if setErr := netns.Set(originalNS); setErr != nil {
                // Log but return original error if any
                if err == nil {
                        err = fmt.Errorf("failed to return to original namespace: %v", setErr)
                }
        }

        return err
}
```

D. Core Logic Skeleton Code

NetworkManager implementation (to be completed by the learner):

GO

```go
// internal/network/manager.go

package network


import (

	"fmt"

	"net"

	"os"

	"path/filepath"

	"strconv"


	"your-project/internal/container"

)


// NetworkManager handles container network setup and teardown

type NetworkManager struct {

	baseBridgeName string

	bridgeIPNet    *net.IPNet

	ipam           *IPAM

	helper         *NetlinkHelper

}


// NewNetworkManager creates a new NetworkManager with default bridge configuration

func NewNetworkManager() (*NetworkManager, error) {

	// Default bridge subnet: 172.17.0.0/16 (Docker-compatible)

	_, ipNet, err := net.ParseCIDR("172.17.0.1/16")

	if err != nil {

		return nil, fmt.Errorf("failed to parse default subnet: %v", err)

	}


	return &NetworkManager{

		baseBridgeName: "byod0",

		bridgeIPNet:    ipNet,

		ipam:           NewIPAM(ipNet),

		helper:         NewNetlinkHelper(),

	}, nil

}


// SetupNetwork configures the network namespace for a container

func (nm *NetworkManager) SetupNetwork(nsPath string, config container.NetworkConfig) error {

	// TODO 1: Validate network configuration

	//   - Check NetworkConfig.Mode is supported (e.g., "bridge", "none", "host")
```

```
        //   - For bridge mode, ensure BridgeName is set or use default
        //   - Validate IPAddress if provided (must be within bridge subnet)
        //   - Validate PortMappings (host port > 0, container port > 0)

        // TODO 2: Create or ensure bridge exists
        //   - Use nm.helper.CreateBridge() to create bridge if it doesn't exist
        //   - Set bridge IP address (first IP in subnet, e.g., 172.17.0.1/16)
        //   - Enable IP forwarding on host: sysctl net.ipv4.ip_forward=1

        // TODO 3: Allocate IP address for container
        //   - Use nm.ipam.Allocate() to get an available IP
        //   - If config.IPAddress is provided, try to allocate that specific IP
        //   - Store allocation for later cleanup

        // TODO 4: Create veth pair
        //   - Generate unique names: host veth "veth<containerID[:8]>", container veth "eth0"
        //   - Use nm.helper.CreateVethPair() with MTU 1500
        //   - Move container veth end to container's network namespace using nm.helper.MoveInterfaceToNamespace()

        // TODO 5: Configure container veth inside namespace
        //   - Use nm.helper.LockOSThreadAndExecute() to enter container namespace
        //   - Inside namespace:
        //     a. Rename container veth to "eth0"
        //     b. Configure IP address and default gateway (bridge IP)
        //     c. Set interface UP
        //     d. Configure loopback interface (set UP)

        // TODO 6: Attach host veth to bridge
        //   - Use netlink to attach host veth end to bridge

        // TODO 7: Setup iptables rules
        //   - Masquerading: Allow container outbound traffic via NAT
        //   - Port forwarding: DNAT rules for each PortMapping
        //   - Inter-container communication: Allow FORWARD traffic between bridge interfaces

        // TODO 8: Configure DNS in container
        //   - Create /etc/resolv.conf in container rootfs with nameservers (e.g., 8.8.8.8)

        // TODO 9: Record network configuration for later cleanup
        //   - Store veth names, IP allocation, iptables rule identifiers

        return nil
}
```

```go
// CleanupNetwork removes all network resources for a container
func (nm *NetworkManager) CleanupNetwork(containerID string, config container.NetworkConfig) error {
	// TODO 1: Retrieve saved network configuration for this container

	// TODO 2: Remove iptables rules
	//   - Delete masquerading rule
	//   - Delete port forwarding rules
	//   - Delete inter-container communication rules if this is the last container

	// TODO 3: Remove veth pair from bridge and delete it

	// TODO 4: Release IP address back to IPAM pool

	// TODO 5: If this is the last container using the bridge, optionally remove bridge

	return nil
}

// IPAM manages IP address allocation within a subnet
type IPAM struct {
	subnet    *net.IPNet
	allocated map[string]bool
}

// NewIPAM creates a new IPAM for the given subnet
func NewIPAM(subnet *net.IPNet) *IPAM {
	return &IPAM{
		subnet:    subnet,
		allocated: make(map[string]bool),
	}
}

// Allocate allocates an available IP address from the subnet
func (i *IPAM) Allocate(requestedIP string) (net.IP, error) {
	// TODO 1: If requestedIP is provided, validate it's within subnet and not allocated
	// TODO 2: If no requestedIP, find first available IP in subnet (skip network and broadcast addresses)
	// TODO 3: Mark IP as allocated in the map
	// TODO 4: Return allocated IP
	return nil, nil
}

// Release returns an IP address to the pool
```

```go
func (i *IPAM) Release(ip net.IP) error {

        // TODO 1: Validate IP is within subnet

        // TODO 2: Check if IP is actually allocated

        // TODO 3: Remove from allocated map

        return nil

}
```

iptables rule management (skeleton):

GO

```go
// internal/network/iptables.go

package network


import (

	"fmt"

	"os/exec"

	"strconv"

)


// IPTablesManager manages iptables rules for container networking

type IPTablesManager struct{}


// SetupMasquerade sets up NAT masquerading for containers to reach external networks

func (im *IPTablesManager) SetupMasquerade(bridgeName string) error {

	// TODO 1: Check if masquerade rule already exists in nat table POSTROUTING chain

	// TODO 2: If not, add rule: iptables -t nat -A POSTROUTING -s <bridgeSubnet> ! -o <bridgeName> -j MASQUERADE

	// TODO 3: Also ensure FORWARD chain accepts traffic from and to bridge

	return nil

}


// AddPortForwarding adds DNAT rule to forward host port to container port

func (im *IPTablesManager) AddPortForwarding(hostPort, containerPort int, protocol, containerIP string) error {

	// TODO 1: Validate protocol is "tcp" or "udp"

	// TODO 2: Add rule: iptables -t nat -A PREROUTING -p <protocol> --dport <hostPort> -j DNAT --to-destination <containerIP>:
<containerPort>

	// TODO 3: Add rule: iptables -A FORWARD -p <protocol> -d <containerIP> --dport <containerPort> -j ACCEPT

	return nil

}


// RemovePortForwarding removes the DNAT rule

func (im *IPTablesManager) RemovePortForwarding(hostPort, containerPort int, protocol, containerIP string) error {

	// TODO 1: Construct the exact rule that was added

	// TODO 2: Use iptables -t nat -D PREROUTING ... to delete the rule

	// TODO 3: Delete the corresponding FORWARD rule

	return nil

}


// runIptablesCommand is a helper to execute iptables commands

func runIptablesCommand(args ...string) error {

	cmd := exec.Command("iptables", args...)

	output, err := cmd.CombinedOutput()

	if err != nil {
```

```
                return fmt.Errorf("iptables %v failed: %v, output: %s", args, err, output)
        }
        return nil
}
```

E. Language-Specific Hints

1. **Namespace Handling**: Use `runtime.LockOSThread()` when switching network namespaces with `netns.Set()`. The Go scheduler may move goroutines between threads, causing namespace switches to affect other goroutines.
2. **Netlink Library**: The `github.com/vishvananda/netlink` and `github.com/vishvananda/netns` packages are de facto standards for Go container networking. They wrap complex netlink socket operations.
3. **Error Cleanup**: Use defer statements strategically. When creating multiple resources (bridge, veth, iptables rules), create a cleanup function that can be called on failure.
4. **Concurrent Access**: The `NetworkManager` may be called concurrently for multiple containers. Use a mutex to protect the IPAM allocation map and bridge creation.
5. **File Descriptors**: When working with network namespaces via file descriptors (`/proc/<pid>/ns/net`), ensure they're closed properly to avoid resource leaks.

F. Milestone Checkpoint

Verification Steps:

1. **Basic Network Isolation**:

```bash
# Run a container with network isolation
sudo ./byod run --network=bridge --hostname=testcontainer alpine ip addr
```

Expected: The container should show only `lo` and `eth0` interfaces, not host interfaces.

2. **External Connectivity**:

```bash
# Run a container and test external connectivity
sudo ./byod run --network=bridge alpine ping -c 3 8.8.8.8
```

Expected: Ping succeeds with 3 packets transmitted and received.

3. **DNS Resolution**:

```bash
# Test DNS inside container
sudo ./byod run --network=bridge alpine nslookup google.com
```

Expected: Returns IP addresses for google.com.

4. **Port Forwarding**:

```bash
# In one terminal, run a web server in container
sudo ./byod run --network=bridge -p 8080:80 alpine nc -l -p 80


# In another terminal, connect via host port
echo "test" | nc localhost 8080
```

Expected: The connection succeeds; the first terminal receives "test".

5. **Inter-Container Communication**:

```bash
# Start first container with custom hostname
sudo ./byod run --network=bridge --name=container1 alpine sleep 30 &
CONTAINER1_PID=$!

# Get its IP address (you'll need to implement `byod inspect` or check bridge)
CONTAINER1_IP=172.17.0.2  # Adjust based on actual allocation

# Start second container and ping first
sudo ./byod run --network=bridge alpine ping -c 2 $CONTAINER1_IP
```

Expected: Ping succeeds between containers.

Signs of Problems:

- `ping: bad address 'google.com'` → DNS misconfiguration
- `connect: Network is unreachable` → Default route missing or bridge not configured
- `bind: address already in use` → Host port conflict
- `ping: sendmsg: operation not permitted` → Missing NAT masquerading rule

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Container has no network interfaces	Network namespace not created or veth not moved	Check `/proc/<pid>/ns/net` symlink target; run `ip netns list`	Ensure `CLONE_NEWNET` flag is used and veth is moved to namespace
Container cannot ping external IPs	Missing NAT masquerading or IP forwarding disabled	Check `sysctl net.ipv4.ip_forward`; check iptables `POSTROUTING` chain	Enable IP forwarding; add masquerade rule
Container cannot resolve DNS	Missing/incorrect `/etc/resolv.conf`	Check container's `/etc/resolv.conf` contents	Copy host's resolv.conf or use public DNS (8.8.8.8)
Host cannot connect to container port	Missing port forwarding rule or wrong iptables chain	Check `iptables -t nat -L PREROUTING -n` for DNAT rule	Add correct DNAT rule and ensure FORWARD chain allows traffic
Containers on same bridge cannot ping each other	FORWARD chain dropping packets between bridge interfaces	Check `iptables -L FORWARD -n`; check bridge STP settings	Add `iptables -I FORWARD -i br0 -o br0 -j ACCEPT`
veth interface disappears after container exit	Not properly cleaning up network namespace	Check if network namespace still exists: `ip netns list`	Ensure cleanup deletes namespace or moves veth back before deleting

Diagnostic Commands:

```bash
# Inspect network namespaces

sudo ip netns list

sudo ls -la /proc/<container-pid>/ns/


# Check bridge configuration

sudo brctl show  # or ip link show type bridge

sudo ip addr show br0


# Check iptables rules

sudo iptables -t nat -L -n -v

sudo iptables -L FORWARD -n -v


# Check routing inside container namespace

sudo nsenter -t <pid> -n ip route show

sudo nsenter -t <pid> -n ip addr show


# Check DNS configuration

sudo nsenter -t <pid> -n cat /etc/resolv.conf
```

> **Milestone(s):** Milestone 6: Image Format and CLI

Component Design: Image Format and CLI (Milestone 6)

This component is the user-facing interface and the image management engine of the container runtime. It transforms user commands into containerized processes by orchestrating all the underlying isolation primitives and managing the lifecycle of container images according to the OCI (Open Container Initiative) specification. While the previous components (`NamespaceManager`, `CgroupManager`, `FilesystemManager`, `NetworkManager`) provide the raw isolation capabilities, this component provides the cohesive user experience and image distribution model that make containers usable and portable.

Mental Model: Recipe Cards and Ingredient Boxes

Think of a container image as a **recipe card** for creating a standardized environment. The card (`ImageConfig`) lists the ingredients (`ImageLayers` - tar archives of filesystem diffs) and the instructions (`Entrypoint`, `Cmd`, `Env`) for what to run. The **ingredient boxes** are stored in a shared pantry (`image store`), where each box is labeled with a unique hash (`Digest`) of its contents. When you want to run the recipe, you don't copy all the boxes to your kitchen; instead, you create a thin, writable overlay on top of the stacked boxes (`OverlayFS merged view`) to cook in. If multiple people are making the same dish, they can all reference the same read-only ingredient boxes, saving immense space and time.

The **CLI (Command Line Interface)** is like the head chef who reads the recipe card, gathers the ingredients from the pantry (or orders them online from a `registry`), sets up a new, private kitchen station (`namespaces`, `cgroups`, `network`), and then hands over control to the recipe's instructions to cook the meal (run the `Entrypoint`). The chef also keeps a logbook (`ContainerStore`) of all the kitchen stations currently set up, their status, and who is cooking what.

This mental model clarifies the separation of concerns: **images are immutable, content-addressed templates**, and **containers are runtime instances** of those templates with added writable state, resource limits, and network connectivity. The CLI is the orchestration layer that glues these concepts together into a coherent workflow.

Architecture Decision Records: OCI Compatibility

> **Decision: Implement OCI Image and Runtime Specs for Maximum Compatibility**
>
> - **Context**: Our runtime must be able to run standard container images (like those from Docker Hub) and interoperate with existing ecosystem tools. We need to choose an image format and runtime interface.
> - **Options Considered**:
>   1. **Proprietary, Simple Format**: Define a custom, minimal tarball and JSON config format.
>   2. **Docker v1 Image Format**: Implement Docker's older, legacy image manifest format.
>   3. **OCI Image & Runtime Specifications**: Implement the open standards defined by the Open Container Initiative.
> - **Decision**: Implement a subset of the **OCI Image Specification (v1.0.2)** and adhere to the **OCI Runtime Specification** for the low-level container execution.
> - **Rationale**: The OCI specs are industry standards, ensuring our runtime can pull and run any OCI-compliant image (including those built by Docker, Podman, and buildah). This maximizes utility and educational value by connecting to the real-world container ecosystem. While more complex than a proprietary format, the specs are well-documented and designed for interoperability.
> - **Consequences**: We must parse multi-layer manifests, handle content-addressable storage via layer digests, and structure our container creation process to match the OCI runtime lifecycle. This adds implementation complexity but yields a tool with real-world relevance.

Option	Pros	Cons	Chosen?
Proprietary Format	Simple to implement and understand; total control.	Cannot run existing images; not useful beyond the project.	❌
Docker v1 Format	Wide historical image base; well-understood.	Legacy, deprecated format; less future-proof.	❌
OCI Specifications	Industry standard; interoperable; future-proof; well-documented.	More complex; larger spec surface area.	✅

A second critical decision involves the client-registry protocol:

> **Decision: Implement Basic OCI Distribution API for Image Pull**
>
> - **Context**: To get images, we need to communicate with a registry (like Docker Hub). The protocol must support authentication, manifest fetching, and layer downloads.
> - **Options Considered**:
>   1. **Local Images Only**: Only support loading images from tar files on disk.
>   2. **Implement Docker Registry HTTP API V2**: The protocol Docker uses.
>   3. **Implement OCI Distribution Spec**: The standardized protocol for distributing OCI images.
> - **Decision**: Implement the core of the **OCI Distribution Specification** (which is largely compatible with Docker Registry API v2) for pulling images.
> - **Rationale**: The OCI Distribution Spec is the standardized, non-proprietary way to fetch OCI images. Its overlap with Docker's API means we can pull from `docker.io` and other compliant registries. Supporting only local archives would severely limit the tool's usefulness.
> - **Consequences**: We must handle HTTP `GET` requests for manifests and blob (layer) downloads, parse authentication realms from `WWW-Authenticate` headers for public images (and optionally support token-based auth), and respect layer `Content-Type` headers. We will not initially support pushing (`PUT`) images.

Common Pitfalls in Image Handling and CLI

⚠️ Pitfall: Assuming Manifest is a Single JSON File

- **Description**: An OCI image index or manifest is not a simple JSON file sitting alone. It's part of a **content-addressable store** where the digest (hash) of the manifest's content is used as its address. The `index.json` points to a platform-specific manifest, which then points to a config and layers.
- **Why it's wrong**: Trying to fetch `index.json` as a static file will fail. You must use the digest (or tag) to request the manifest blob from the `/manifests/` endpoint. The registry returns the manifest's content *and* a `Docker-Content-Digest` header, which you must verify.
- **How to fix**: Always treat manifests as blobs. Use the `Accept: application/vnd.oci.image.manifest.v1+json` header. After downloading, compute the SHA256 of the received content and verify it matches the digest in the response header or the one you requested.

⚠️ Pitfall: Not Managing Layer Cache References

- **Description**: When multiple containers use the same base image layer, the layer should exist only once on disk. If you simply extract layers to a container-specific directory each time, you duplicate storage.
- **Why it's wrong**: This defeats the primary efficiency of container images. It wastes disk space and makes container creation slower.

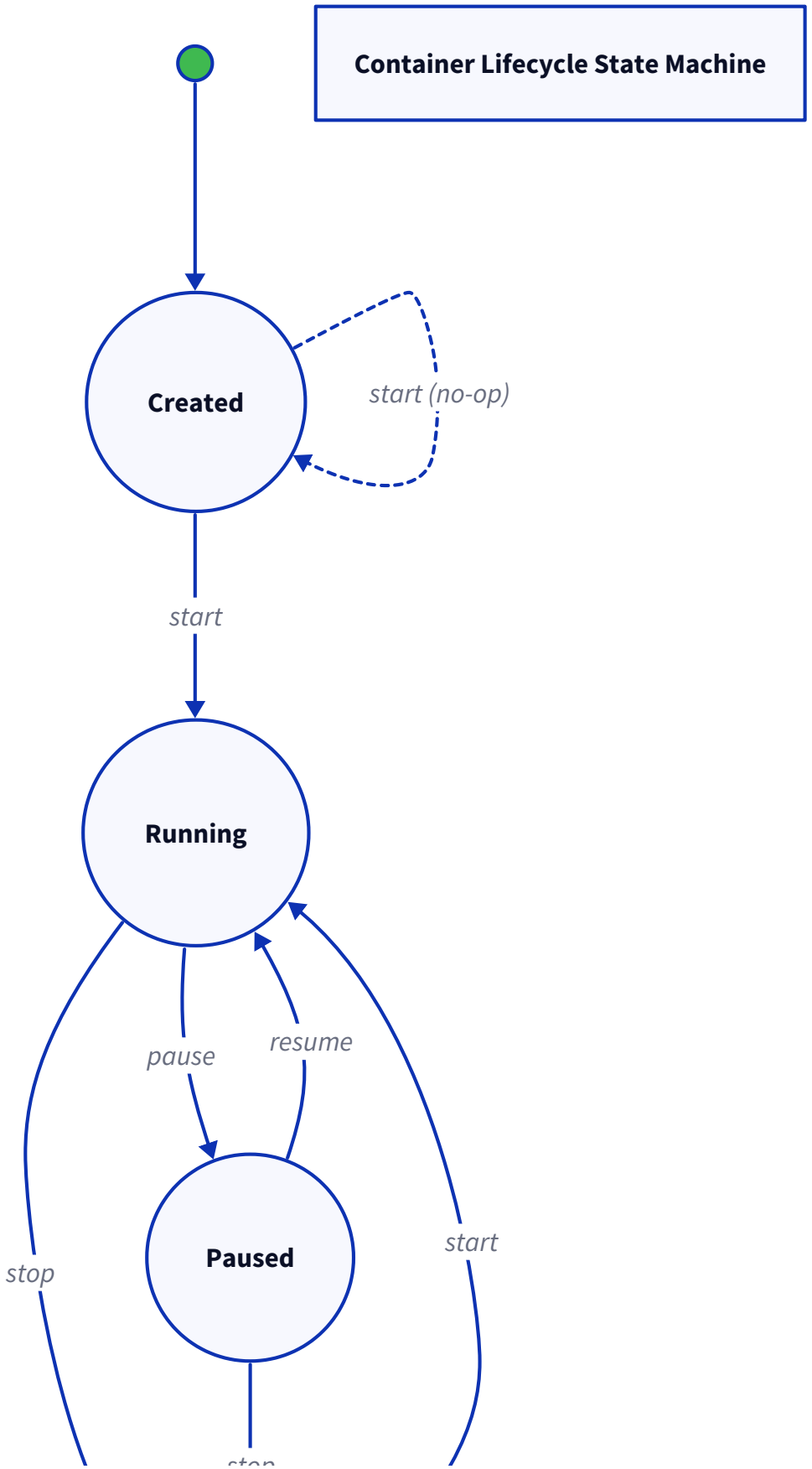
- **How to fix**: Implement a **content-addressable layer cache** (using the `LayerCache` type). Store layers keyed by their digest. When setting up a container rootfs, create hard links or copy-on-write references from the cache to the container's OverlayFS lower directories. Use a reference count to know when a layer can be safely garbage-collected.

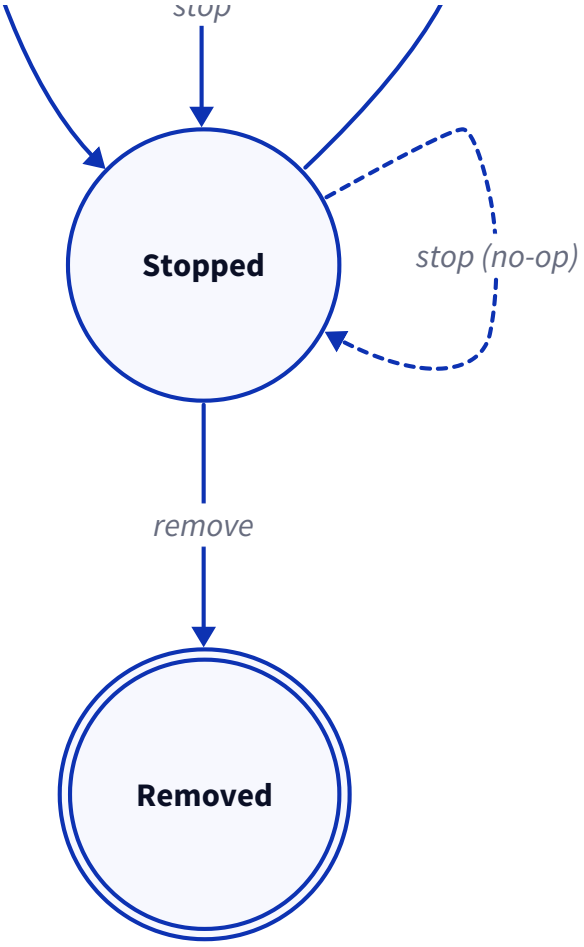
⚠️ Pitfall: Ignoring Image Configuration

- **Description**: The `ImageConfig` (a separate JSON blob referenced by the manifest) contains critical runtime information: the default `Entrypoint`, `Cmd`, `Env`, `WorkingDir`, and `Volumes`. Ignoring it and just extracting layers results in a container that may not start correctly.
- **Why it's wrong**: The container's intended command and environment won't be set. The user would have to specify the full command every time, breaking compatibility with standard images.
- **How to fix**: Parse the `ImageConfig` and use its fields to populate the `ContainerConfig`. The `Entrypoint` and `Cmd` from the image should be combined (following OCI rules) to form the final `argv` for the container process. The `Env` should be merged with any user-provided environment variables.

⚠️ Pitfall: Poor CLI State Management

- **Description**: The CLI's `start`, `stop`, `remove` commands must track the container's state precisely. For example, trying to `start` an already `running` container should be an error, and `remove` should only work on `stopped` containers.
- **Why it's wrong**: Without a clear state machine, you can get into inconsistent states (e.g., a "removed" container whose cgroups and network are still active). This leads to resource leaks and confusing user experience.
- **How to fix**: Model the container lifecycle as a **state machine** (see diagram below). The `ContainerStore` should persist the `ContainerState`. Every state transition (`start`, `stop`, `remove`) must validate the current state, perform the transition, and update the persisted state atomically. Use file locks or a simple database to prevent concurrent modifications.


Container Lifecycle State Machine
Created
start (no-op)
start
Running
pause
resume
Paused
stop
start



Implementation Guidance for Image Format and CLI

This section bridges the high-level design to concrete implementation in Go. We'll structure the code into clear modules for image handling and CLI commands.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Image Registry Client	HTTP GET with `net/http`, basic token auth	Full OCI distribution client with auth challenges, retries, progress bars
Manifest/Layer Storage	Filesystem in a directory tree (`<store>/blobs/sha256/<digest>`)	Content-addressable storage with integrity verification and garbage collection
CLI Framework	Standard library `flag` package for parsing	`cobra` or `urfave/cli` for structured subcommands, help generation
State Persistence	JSON files in a directory (`<store>/containers/<id>/config.json`)	Embedded SQLite database for atomic transactions and queries
Image Cache	In-memory map of digest to path with file-based refcount	Deduplicated storage with reflinks (if filesystem supports)

B. Recommended File/Module Structure

The Image and CLI components integrate with the existing runtime components. Create a new `internal` directory for them.

```
project-root/
├── cmd/
│   └── byod/                       # CLI entry point
│       └── main.go                 # Parses flags, calls runtime
├── internal/
│   ├── cli/                        # CLI command implementations
│   │   ├── commands.go             # run, start, stop, ps, rm commands
│   │   └── utils.go                # Table output, formatting helpers
│   ├── image/                      # OCI image handling
│   │   ├── image.go                # Image, ImageConfig, ImageLayer types
│   │   ├── store.go                # ImageStore (local cache of images)
│   │   ├── registry/               # Client for remote registries
│   │   │   ├── client.go           # RegistryClient, pull logic
│   │   │   ├── auth.go             # Token authentication
│   │   │   └── types.go            # Manifest, Descriptor types
│   │   └── oci/                    # OCI spec structs (can be generated)
│   │       ├── manifest.go         # OCI Manifest and Index structs
│   │       └── config.go           # OCI Image Config structs
│   └── runtime/                    # Existing components (from prev. milestones)
│       ├── container.go            # Container, ContainerConfig, etc.
│       ├── store.go                # ContainerStore
│       ├── manager.go              # ContainerRuntime (orchestrates all managers)
│       ├── namespace/
│       ├── cgroup/
│       ├── filesystem/
│       ├── network/
│       └── overlay/
└── storage/                        # Default storage directory
    ├── containers/
    │   └── <id>/
    │       ├── config.json
    │       └── rootfs/             # OverlayFS merged directory
    └── images/
        ├── <image-name>/
        │   └── manifest.json       # Cached manifest
        └── blobs/
            └── sha256/
                ├── <digest1>       # Layer tar.gz
                ├── <digest2>       # Config json
                └── ...
```

C. Infrastructure Starter Code

Here is complete, working starter code for the **ImageStore** (local cache) and the **RegistryClient**'s core HTTP fetching logic. These are prerequisites that learners can copy and use directly.

File: `internal/image/store.go`

```go
package image

import (
	"crypto/sha256"
	"encoding/hex"
	"encoding/json"
	"fmt"
	"io"
	"os"
	"path/filepath"

	"yourproject/internal/image/oci"
)

// ImageStore manages the local cache of OCI images.
type ImageStore struct {
	basePath string // e.g., "/var/lib/byod/images"
}

// NewImageStore creates a new store at the given base directory.
func NewImageStore(basePath string) (*ImageStore, error) {
	if err := os.MkdirAll(basePath, 0755); err != nil {
		return nil, fmt.Errorf("creating image store directory: %w", err)
	}
	// Create blob directory structure
	blobsPath := filepath.Join(basePath, "blobs", "sha256")
	if err := os.MkdirAll(blobsPath, 0755); err != nil {
		return nil, fmt.Errorf("creating blobs directory: %w", err)
	}
	return &ImageStore{basePath: basePath}, nil
}

// blobPath returns the filesystem path for a blob with the given digest.
// Digest must be of the form "sha256:abcdef..."
func (s *ImageStore) blobPath(digest string) (string, error) {
	if len(digest) < 9 || digest[:7] != "sha256:" {
		return "", fmt.Errorf("invalid digest format: %s", digest)
	}
	hash := digest[7:]
	return filepath.Join(s.basePath, "blobs", "sha256", hash), nil
}
```

```
// StoreBlob saves a blob (layer tar, config json, manifest) to the store.
// It computes the SHA256 digest of the content as it's written and verifies it matches the provided digest.
// Returns the digest (sha256:...) on success.
func (s *ImageStore) StoreBlob(src io.Reader, expectedDigest string) (string, error) {
        path, err := s.blobPath(expectedDigest)
        if err != nil {
                return "", err
        }
        // If already exists, verify? For simplicity, assume it's correct.
        if _, err := os.Stat(path); err == nil {
                return expectedDigest, nil
        }

        // Write to temporary file first, then rename atomically.
        tmpPath := path + ".tmp"
        tmpFile, err := os.Create(tmpPath)
        if err != nil {
                return "", fmt.Errorf("creating temp blob file: %w", err)
        }
        defer tmpFile.Close()
        defer os.Remove(tmpPath)

        hash := sha256.New()
        multiWriter := io.MultiWriter(tmpFile, hash)

        if _, err := io.Copy(multiWriter, src); err != nil {
                return "", fmt.Errorf("copying blob data: %w", err)
        }
        if err := tmpFile.Close(); err != nil {
                return "", err
        }

        computedDigest := "sha256:" + hex.EncodeToString(hash.Sum(nil))
        if computedDigest != expectedDigest {
                return "", fmt.Errorf("digest mismatch: expected %s, got %s", expectedDigest, computedDigest)
        }

        if err := os.Rename(tmpPath, path); err != nil {
                return "", fmt.Errorf("renaming blob to final location: %w", err)
        }
```

```
        return computedDigest, nil
}

// GetBlob returns a ReadCloser for the blob with the given digest.
func (s *ImageStore) GetBlob(digest string) (io.ReadCloser, error) {
        path, err := s.blobPath(digest)
        if err != nil {
                return nil, err
        }
        return os.Open(path)
}

// GetManifest reads and parses an OCI manifest for a given image reference (tag or digest).
func (s *ImageStore) GetManifest(ref string) (*oci.Manifest, error) {
        // For simplicity, assume ref is a digest. In a real implementation,
        // you'd resolve tags to digests via an index.
        path, err := s.blobPath(ref)
        if err != nil {
                return nil, err
        }
        data, err := os.ReadFile(path)
        if err != nil {
                return nil, fmt.Errorf("reading manifest blob: %w", err)
        }
        var manifest oci.Manifest
        if err := json.Unmarshal(data, &manifest); err != nil {
                return nil, fmt.Errorf("unmarshaling manifest: %w", err)
        }
        return &manifest, nil
}

// StoreManifest saves a manifest and associates it with a tag (image reference).
func (s *ImageStore) StoreManifest(ref string, manifest *oci.Manifest) error {
        data, err := json.Marshal(manifest)
        if err != nil {
                return fmt.Errorf("marshaling manifest: %w", err)
        }
        // Store as a blob first.
        digest, err := s.StoreBlob(bytes.NewReader(data), "")
        if err != nil {
```

```
                return err
        }
        // Then create a tag file pointing to the digest.
        // In OCI, tags are usually managed by an index. Simplified here.
        tagPath := filepath.Join(s.basePath, "refs", ref)
        if err := os.MkdirAll(filepath.Dir(tagPath), 0755); err != nil {
                return err
        }
        return os.WriteFile(tagPath, []byte(digest+"\n"), 0644)
}
```

File: `internal/image/registry/client.go`

```go
package registry


import (

	"encoding/json"

	"fmt"

	"io"

	"net/http"

	"net/url"

	"strings"


	"yourproject/internal/image/oci"

)


// RegistryClient can pull images from an OCI-compliant registry.

type RegistryClient struct {

	baseURL    string // e.g., "https://registry-1.docker.io"

	httpClient *http.Client

}


// NewRegistryClient creates a client for a given registry host.

func NewRegistryClient(host string) *RegistryClient {

	return &RegistryClient{

		baseURL:    "https://" + host,

		httpClient: &http.Client{},

	}

}


// getManifest fetches the manifest for a given image reference (e.g., "library/alpine:latest").

// It returns the raw manifest bytes and its digest (from Docker-Content-Digest header).

func (c *RegistryClient) getManifest(imageRef, tag string) ([]byte, string, error) {

	// URL path: /v2/<image>/manifests/<tag>

	path := fmt.Sprintf("/v2/%s/manifests/%s", imageRef, tag)

	req, err := http.NewRequest("GET", c.baseURL+path, nil)

	if err != nil {

		return nil, "", err

	}

	// Accept OCI manifest format

	req.Header.Set("Accept", "application/vnd.oci.image.manifest.v1+json")

	// For Docker Hub, also accept Docker's manifest schema.

	req.Header.Add("Accept", "application/vnd.docker.distribution.manifest.v2+json")
```

```
        resp, err := c.httpClient.Do(req)
        if err != nil {
                return nil, "", fmt.Errorf("HTTP request failed: %w", err)
        }
        defer resp.Body.Close()

        if resp.StatusCode != http.StatusOK {
                return nil, "", fmt.Errorf("unexpected status %d for manifest", resp.StatusCode)
        }

        data, err := io.ReadAll(resp.Body)
        if err != nil {
                return nil, "", err
        }

        digest := resp.Header.Get("Docker-Content-Digest")
        if digest == "" {
                // Fallback: compute digest locally if header missing.
                digest = computeSHA256Digest(data)
        }
        return data, digest, nil
}

// PullManifest fetches and parses the manifest for an image.
func (c *RegistryClient) PullManifest(imageRef, tag string) (*oci.Manifest, string, error) {
        data, digest, err := c.getManifest(imageRef, tag)
        if err != nil {
                return nil, "", err
        }
        var manifest oci.Manifest
        if err := json.Unmarshal(data, &manifest); err != nil {
                return nil, "", fmt.Errorf("unmarshaling manifest: %w", err)
        }
        return &manifest, digest, nil
}

// PullLayer downloads a layer blob identified by digest and writes it to the provided writer.
func (c *RegistryClient) PullLayer(imageRef, digest string, w io.Writer) error {
        // URL path: /v2/<image>/blobs/<digest>
        path := fmt.Sprintf("/v2/%s/blobs/%s", imageRef, digest)
        req, err := http.NewRequest("GET", c.baseURL+path, nil)
```

```
        if err != nil {
                return err
        }
        resp, err := c.httpClient.Do(req)
        if err != nil {
                return fmt.Errorf("HTTP request failed: %w", err)
        }
        defer resp.Body.Close()
        if resp.StatusCode != http.StatusOK {
                return fmt.Errorf("unexpected status %d for layer %s", resp.StatusCode, digest)
        }
        _, err = io.Copy(w, resp.Body)
        return err
}

// Helper function to compute SHA256 digest.
func computeSHA256Digest(data []byte) string {
        // Implementation omitted for brevity; use crypto/sha256.
        return "sha256:..."
}
```

D. Core Logic Skeleton Code

Now, the core logic that learners should implement themselves: the `ContainerRuntime` methods that orchestrate image pull and container creation, and the CLI command handlers.

File: `internal/runtime/manager.go` (additions)

```go
// PullImage downloads an image from a registry, stores it locally, and returns an Image object.
// imageRef format: [registry/]repository[:tag|@digest] (e.g., "alpine:latest", "docker.io/library/nginx:alpine")
func (r *ContainerRuntime) PullImage(imageRef string) (*image.Image, error) {
	// TODO 1: Parse the image reference into registry host, repository, and tag/digest.
	//         Default registry to "docker.io" and tag to "latest" if not specified.
	// TODO 2: Create a RegistryClient for the registry host.
	// TODO 3: Call PullManifest to fetch the manifest and its digest.
	// TODO 4: Download the config blob (manifest.Config.Digest) using PullLayer and store it via ImageStore.StoreBlob.
	// TODO 5: For each layer descriptor in manifest.Layers:
	//   a. Check if the layer already exists in the local ImageStore (by digest).
	//   b. If not, download it via PullLayer, writing to a temporary file or pipe.
	//   c. Store the downloaded layer blob using ImageStore.StoreBlob.
	//   d. Optionally, update a progress bar for the user.
	// TODO 6: Create and return an image.Image struct populated with the manifest data and local layer paths.
	return nil, nil
}

// CreateContainerFromImage creates a new container from a locally stored image.
// It performs all isolation setup but does not start the process.
func (r *ContainerRuntime) CreateContainerFromImage(imageName string, userConfig ContainerConfig) (*Container, error) {
	// TODO 1: Look up the image by name in the ImageStore. If not found, return error (or pull?).
	// TODO 2: Merge the image's configuration (ImageConfig) with the user's ContainerConfig.
	//         Rules: user-specified Cmd overrides image's Entrypoint+Cmd; user Env extends image Env.
	// TODO 3: Generate a unique container ID (e.g., using uuid.Generate()).
	// TODO 4: Create a Container struct with the merged config, ID, and state StateCreated.
	// TODO 5: Call FilesystemManager.SetupRootfs (which internally uses OverlayManager) to prepare the container's root filesystem.
	// TODO 6: Create namespaces for the container (using NamespaceManager.CreateNamespaces). This may involve forking a child process that pauses.
	// TODO 7: Create cgroups for the container and apply resource limits (CgroupManager.CreateGroup, SetLimits).
	// TODO 8: Setup network namespace (NetworkManager.SetupNetwork) if network mode is not "none".
	// TODO 9: Save the container state to ContainerStore (ContainerStore.Save).
	// TODO 10: Return the created Container object.
	return nil, nil
}
```

File: `internal/cli/commands.go`

```go
package cli

import (
	"fmt"
	"os"

	"yourproject/internal/runtime"
)

// RunCommand handles `byod run <image> [command...]`
func RunCommand(r *runtime.ContainerRuntime, imageRef string, userCmd []string) error {
	// TODO 1: Check if image exists locally; if not, call runtime.PullImage to download it.
	// TODO 2: Create a ContainerConfig, populating Cmd with userCmd (if provided), and other defaults.
	// TODO 3: Call runtime.CreateContainerFromImage to create a container in StateCreated.
	// TODO 4: Call runtime.StartContainer to start the container process (which will transition to StateRunning).
	// TODO 5: If the run is detached (--detach flag), print the container ID and return.
	// TODO 6: If attached (foreground), wait for the container process to exit (by monitoring its PID).
	// TODO 7: After the process exits, call runtime.StopContainer to transition to StateStopped.
	// TODO 8: Optionally, remove the container automatically if --rm flag is set.
	return nil
}

// StartCommand handles `byod start <container-id>`
func StartCommand(r *runtime.ContainerRuntime, containerID string) error {
	// TODO 1: Load the container from ContainerStore by ID.
	// TODO 2: Validate that the container state is StateCreated (or StateStopped if we support restart).
	// TODO 3: Call runtime.StartContainer.
	// TODO 4: If attached, wait for the process and then update state to StateStopped.
	return nil
}

// StopCommand handles `byod stop <container-id>`
func StopCommand(r *runtime.ContainerRuntime, containerID string, force bool) error {
	// TODO 1: Load the container from ContainerStore.
	// TODO 2: Validate state is StateRunning (or StatePaused).
	// TODO 3: If force is true, send SIGKILL to the container's init process. Otherwise, send SIGTERM, wait, then SIGKILL.
	// TODO 4: Call runtime.StopContainer to update state to StateStopped.
	return nil
}

// RemoveCommand handles `byod rm <container-id>`
```

```
func RemoveCommand(r *runtime.ContainerRuntime, containerID string) error {

        // TODO 1: Load the container.

        // TODO 2: Validate state is StateStopped (cannot remove running containers).

        // TODO 3: Call runtime.RemoveContainer which will:

        //   a. Clean up network resources (NetworkManager.CleanupNetwork).

        //   b. Remove cgroups (CgroupManager.RemoveCgroup).

        //   c. Unmount and clean up the root filesystem (FilesystemManager.CleanupFilesystem).

        //   d. Delete the container from ContainerStore.

        // TODO 4: If successful, print confirmation.

        return nil

}
```

E. Language-Specific Hints

1. **Go's `net/http` for Registry Client**: Use `http.Client` with a custom `Transport` if you need to add authentication headers globally. Remember to close response bodies (`defer resp.Body.Close()`). For downloading large layers, consider using `io.Copy` with a buffer to provide progress updates.
2. **JSON Parsing**: Use `encoding/json` with struct tags that match the OCI spec field names. You can generate Go structs from the OCI JSON schema using tools like `quicktype`. For flexibility with unknown fields, use `json.RawMessage` for parts you don't need to parse immediately.
3. **Concurrent Layer Downloads**: To speed up image pull, you can download multiple layers concurrently using goroutines and a `sync.WaitGroup`. Be mindful of registry rate limits. Use a semaphore pattern (buffered channel) to limit concurrency.
4. **CLI Flag Parsing**: The standard `flag` package supports subcommands via `flag.NewFlagSet`. For a more structured CLI, consider using `cobra`. Remember to define persistent flags (like `--storage-path`) and local flags for each command.
5. **State Machine Implementation**: Implement the container state transitions as methods on `Container` or `ContainerRuntime`. Use a `sync.Mutex` to protect state changes if the CLI could be called concurrently (e.g., from multiple terminals). The state should be persisted after each transition.
6. **Error Handling**: Use Go's error wrapping with `fmt.Errorf("... %w", err)` . Provide meaningful error messages to the user (e.g., "image 'foo:latest' not found locally and pulling from registry failed: connection refused").

F. Milestone Checkpoint

After implementing Milestone 6, you should be able to perform the following workflow:

1. **Pull an image**: `sudo ./byod pull alpine:latest`
   - **Expected**: Output showing layer download progress (e.g., "Layer sha256:abc... downloaded"). The image should be stored in the configured storage directory (e.g., `/var/lib/byod/images`).
   - **Verification**: Check `ls -la /var/lib/byod/images/blobs/sha256/` to see downloaded layer files.
2. **Run a container**: `sudo ./byod run alpine:latest echo "hello world"`
   - **Expected**: The command outputs "hello world" and exits. The container should be created, started, execute the command, stop, and (if `--rm` is used) be removed.
   - **Verification**: Use `./byod ps -a` to see the container (if not removed) in `Stopped` state.
3. **Run a detached container**: `sudo ./byod run -d --name myapp nginx:alpine`
   - **Expected**: Prints a container ID. The container stays running in the background.
   - **Verification**: `./byod ps` shows the container in `Running` state. `curl http://<container-ip>` should return the nginx welcome page (if networking is set up).
4. **Container lifecycle**: `sudo ./byod stop myapp` , then `sudo ./byod rm myapp`
   - **Expected**: `stop` terminates the container, `rm` removes it.
   - **Verification**: `./byod ps -a` no longer lists the container. Check that the container's rootfs directory and cgroup are cleaned up.

Signs something is wrong:

- **"manifest unknown" error**: Likely an incorrect image reference format or registry authentication issue. Verify the image exists on Docker Hub.
- **Container exits immediately with code 127**: The `Entrypoint` or `Cmd` binary not found in the image. Check that the image config was parsed correctly and the rootfs was set up properly.
- **Resource leaks (cgroups, mounts) after `rm`**: The `RemoveContainer` method is not cleaning up all resources. Add logging to each cleanup step and verify they are called.

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Pull fails with "unauthorized"	Registry requires authentication, even for public images.	Check the `WWW-Authenticate` header in the HTTP response.	Implement token authentication: request a token from the auth service URL, then retry the request with `Authorization: Bearer <token>`.
Container runs but cannot resolve DNS	`/etc/resolv.conf` is not set up inside the container.	`cat` the container's rootfs `/etc/resolv.conf`; it may be empty or missing.	Copy the host's `/etc/resolv.conf` into the container rootfs during filesystem setup, or bind-mount it.
`byod ps` shows empty list	Container state is not being persisted to disk.	Check the storage directory for container config JSON files.	Ensure `ContainerStore.Save` is called after creating/updating a container, and that it writes valid JSON.
Image layers are re-downloaded every time	Layer cache is not being checked or used.	Check if `ImageStore.StoreBlob` is skipping already-existing blobs.	Implement a layer cache with reference counting. Before downloading, check if the blob file exists and matches the digest.
CLI command hangs after container start	The parent process is waiting for the child but the child might have died.	Check the container's init process status with `ps aux | grep <container-pid>`.	Ensure the child process is correctly reparented to PID 1 after the parent exits. Use double-fork or proper signal handling.

> **Milestone(s):** All milestones (this section describes the dynamic interactions between components across the entire system)

Interactions and Data Flow

This section describes the dynamic orchestration between the architectural components defined in previous sections. While component design explains *what* each part does, interactions and data flow explain *how* and *when* they work together to achieve container lifecycle operations. Understanding these sequences is critical for implementing a cohesive runtime where isolation primitives are applied in the correct order with proper cleanup on failure.

Container Creation Sequence

The container creation sequence transforms a user's request (`container run`) into an isolated, resource-limited process with its own filesystem and network. Think of this as a **restaurant kitchen preparing a complex dish**: the chef (Container Manager) coordinates specialized stations (Namespace, Cgroup, Filesystem managers) in a precise sequence, where timing matters (network must be set up before the process starts, cgroups must be applied before the process forks). If any station fails, the entire order must be rolled back cleanly.

The complete sequence involves 8 major phases, with error handling and rollback at each step:

1. **Request Validation and Preparation** (CLI → Container Manager)
    - The CLI parses user command (image name, command, flags) into a `ContainerConfig`
    - Container Manager validates the configuration (e.g., image exists locally, resource limits are sane)
    - A unique `containerID` is generated (via `uuid.Generate()`)
    - Container entry is created in `ContainerStore` with state `StateCreated`
2. **Image Preparation** (Container Manager → Image Store → Filesystem Manager)
    - If the specified image isn't locally available, trigger the image pull flow (described separately)
    - `ImageStore` loads the `Image` object with its layers and configuration
    - `FilesystemManager.SetupRootfs()` is called with the image and containerID
        - This delegates to `OverlayManager` to create layered rootfs (Milestone 4)
        - Returns path to the prepared root filesystem
3. **Namespace Creation** (Container Manager → Namespace Manager)

- `NamespaceManager.CreateNamespaces()` is called with flags for `CLONE_NEWPID|CLONE_NEWUTS|CLONE_NEWNS|CLONE_NEWNET|CLONE_NEWIPC`
- The manager creates a new child process via `clone()` or `unshare()` with these namespace flags
- This child will eventually become the container's PID 1
- The function returns the host PID of this child process (still in a suspended state)

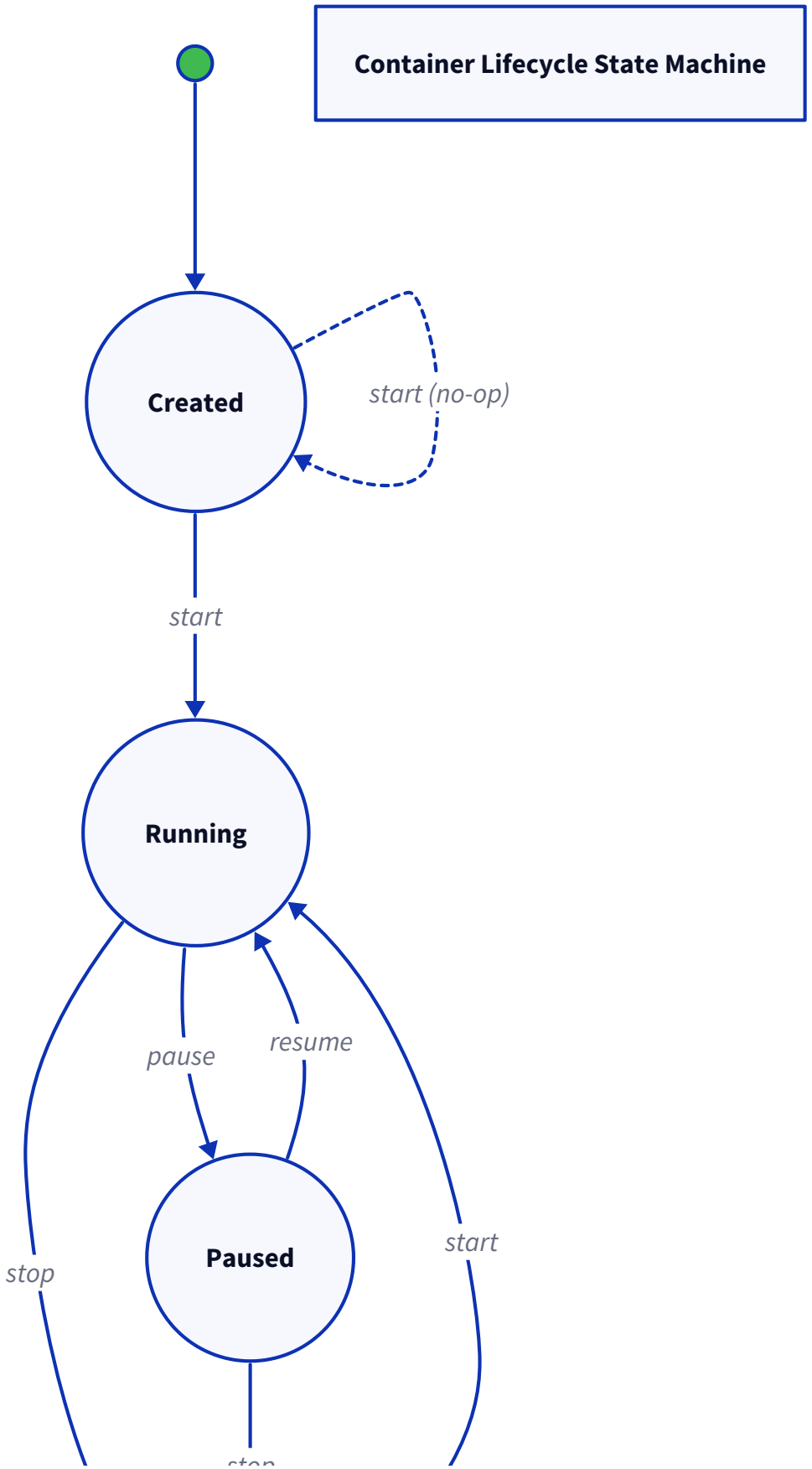
4. **Cgroup Configuration** (Container Manager → Cgroup Manager)
   - `CgroupManager.CreateGroup(containerID)` creates the cgroup hierarchy
   - `CgroupManager.SetLimits(containerID, config.Limits)` writes memory, CPU, and PID limits to control files
   - `CgroupManager.AddProcess(containerID, pid)` moves the child process (from step 3) into the cgroup
   - This ensures resource limits are enforced from the moment the child process starts executing
5. **Network Namespace Setup** (Container Manager → Network Manager)
   - `NetworkManager.SetupNetwork()` is called with the path to the container's network namespace (`/proc/<pid>/ns/net`)
   - Network Manager creates veth pair, attaches one end to bridge, configures IP addresses, sets up iptables rules
   - This happens *outside* the container namespace but affects the namespace by moving interfaces into it
   - Port forwarding rules are established if specified in `NetworkConfig`
6. **Filesystem Isolation Finalization** (Container Manager → Filesystem Manager, inside the child process)
   - The child process (still suspended) executes `ChildEntryPoint()` which calls:
     - `FilesystemManager.IsolateFilesystem(rootfsPath)` to perform `pivot_root()` and mount `/proc` , `/sys`
     - `syscall.Sethostname(config.Hostname)` to set UTS namespace hostname
     - Environment variable setup and working directory change
   - This happens *inside* the container's namespaces, so mount operations only affect the container
7. **Process Execution** (Inside the child process)
   - The child process performs `exec()` to replace itself with the container's entry point command
   - Command is taken from `ImageConfig.Entrypoint` and `ImageConfig.Cmd` , overridden by user's `config.Cmd`
   - The process becomes PID 1 in the container's PID namespace
8. **State Transition and Monitoring** (Container Manager)
   - Container state transitions from `StateCreated` to `StateRunning`
   - `ContainerStore.Save()` persists the updated state
   - Container Manager begins monitoring the container process (via `waitpid()` or cgroup notifications)
   - On process exit, state transitions to `StateStopped` and cleanup is scheduled

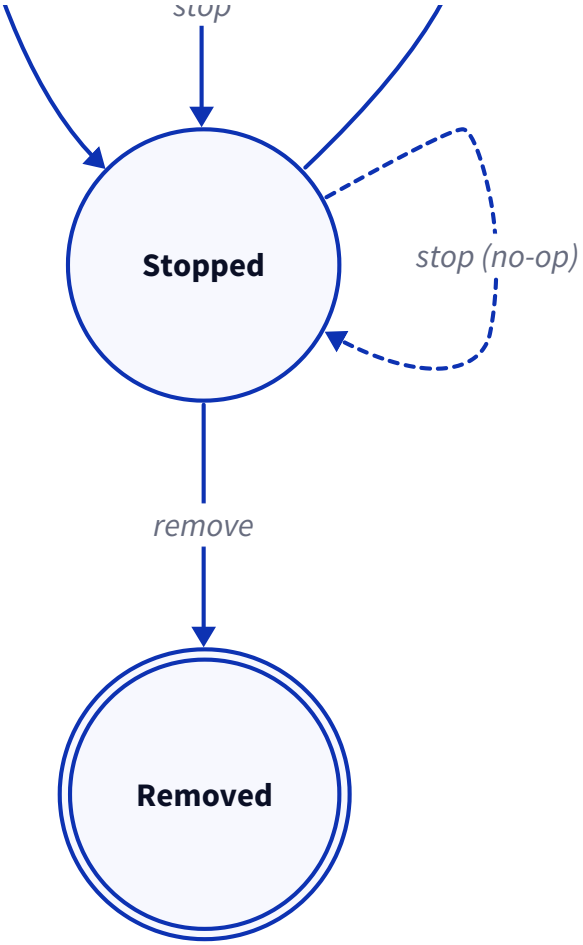
The following table details the key interactions between components during container creation:

Step	Triggering Component	Target Component	Data Passed	Expected Outcome	Error Rollback
1. Validation	CLI	Container Manager	`ContainerConfig`	Validated config, generated containerID	Return error to user
2. Image Prep	Container Manager	Filesystem Manager	`Image` , containerID	Prepared rootfs path	Cleanup any extracted layers
3. Namespace	Container Manager	Namespace Manager	namespace flags	Child process PID	Kill child process if created
4. Cgroup	Container Manager	Cgroup Manager	containerID, `ResourceLimits`	Process in cgroup with limits	Remove cgroup hierarchy
5. Network	Container Manager	Network Manager	nsPath, `NetworkConfig`	Network namespace configured	Remove veth, iptables rules, release IP
6. Filesystem	Container Manager	Filesystem Manager	rootfsPath (inside child)	`pivot_root()` completed, mounts setup	Cannot rollback child - it exits with error
7. Execution	Child process	OS kernel	command, args, env	Container process running	Child exits with error code
8. Monitoring	Container Manager	Container Store	updated `Container` state	State saved as `StateRunning`	Revert to previous state

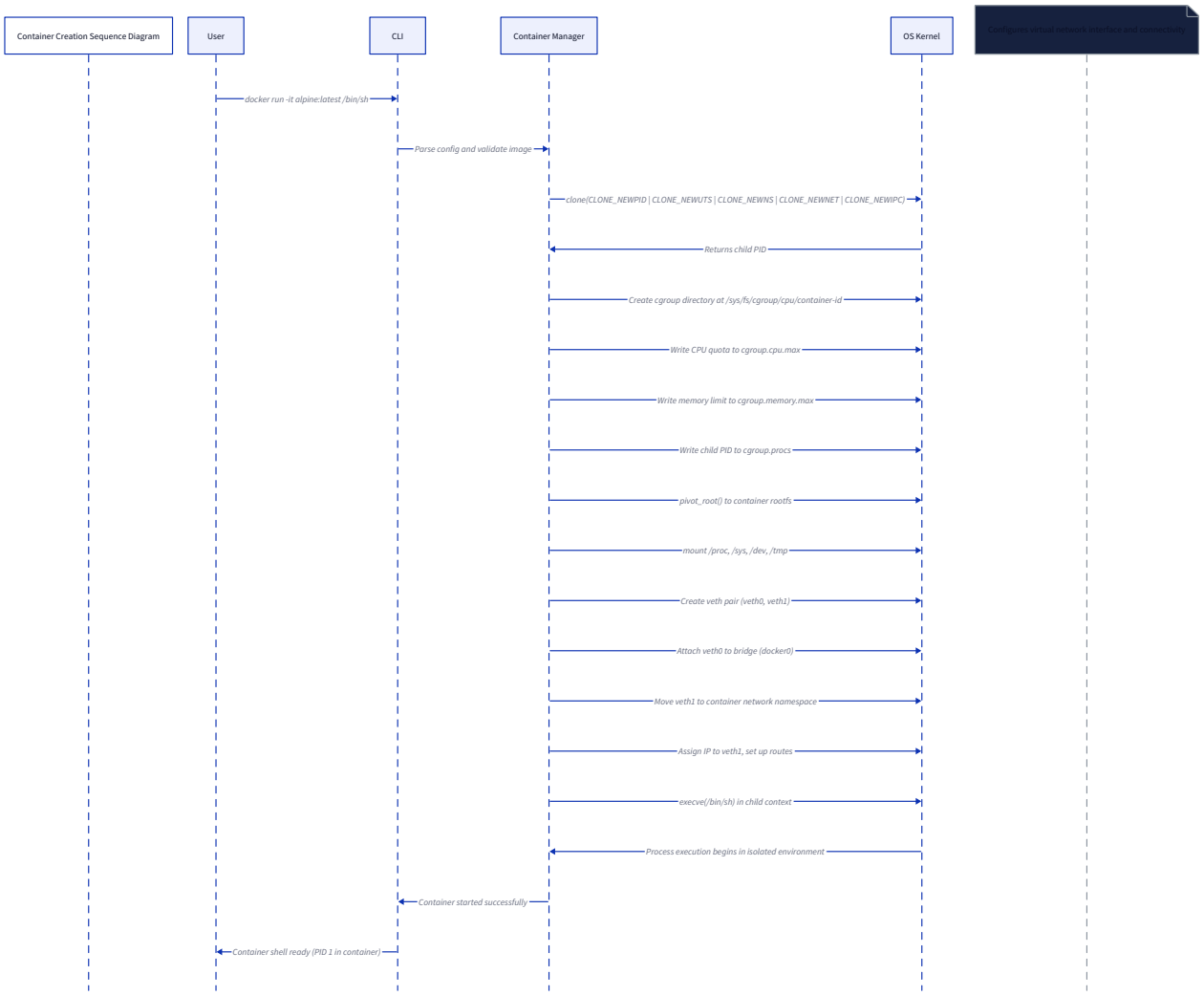
> **Key Insight:** The sequence must preserve the **principle of least privilege escalation** - once the child process enters the container's namespaces and drops privileges (if user namespaces are implemented), it should not perform operations requiring host root privileges. That's why network setup and cgroup configuration happen *outside* the container namespace before the child starts.

The state transitions during this sequence follow a strict state machine:


Container Lifecycle State Machine
Created
start (no-op)
start
Running
pause
resume
Paused
stop
start



For a visual representation of the component interactions, refer to the sequence diagram:



Critical Error Handling Considerations:

- **Rollback must be comprehensive:** If network setup fails after cgroups are created, both must be cleaned up
- **Orphan prevention:** If the Container Manager crashes mid-creation, restart should detect and clean up partially created resources (via containerID scanning)
- **Atomic state transitions:** The `ContainerStore.Save()` operation should be atomic; partial writes could leave the container in an undefined state

Image Pull and Preparation Flow

Image preparation is the process of transforming a named image reference (e.g., `ubuntu:latest`) into a ready-to-use root filesystem. Think of this as a **warehouse fulfillment system**: the clerk (Image Handler) receives an order, checks local inventory (layer cache), requests missing items from suppliers (registry), verifies shipments (digest validation), and assembles the complete package (layered rootfs) for the kitchen (container creation).

The flow involves both remote registry interaction and local filesystem operations:

1. **Reference Parsing** (CLI → Registry Client)
    - Parse `imageRef` (e.g., `docker.io/library/ubuntu:latest`) into registry URL, repository, and tag
    - Default to `latest` tag if not specified
    - Check local `ImageStore` for existing image with same digest
2. **Manifest Fetching** (Registry Client → OCI Registry)
    - `RegistryClient.PullManifest()` makes HTTP request to registry API
    - For Docker Hub: `GET /v2/library/ubuntu/manifests/latest`
    - Registry returns OCI manifest (JSON) with media type and layer digests
    - Client verifies manifest structure and extracts layer descriptors
3. **Layer Download and Verification** (Registry Client → Image Store, per layer)
    - For each layer descriptor in the manifest:
        - Check `LayerCache.IsLayerCached(digest)` - if present, skip download
        - `RegistryClient.PullLayer()` downloads layer blob with progress reporting
        - `ImageStore.StoreBlob()` saves with content-addressable naming (digest as filename)
        - SHA256 verification ensures downloaded content matches digest
        - Layer is marked as cached in `LayerCache.AddReference(digest)`
4. **Image Configuration** (Registry Client → Image Store)
    - Download image configuration blob (referenced in manifest)
    - Parse into `ImageConfig` (entrypoint, env, working directory, etc.)
    - Create and store complete `Image` object with references to all layers
5. **Root Filesystem Preparation** (Container Manager → Filesystem Manager, on demand)
    - When container creation requires the image, `FilesystemManager.SetupRootfs()` is called
    - `OverlayManager.PrepareLayersForContainer()` assembles layers:
        - Lower directories: all read-only layers in order (base → top)
        - Upper directory: writable layer for this container
        - Work directory: internal OverlayFS work area
    - `OverlayManager.MountOverlay()` creates the merged view
    - Returns path to merged directory ready for `pivot_root()`

The decision flow for image handling can be visualized:

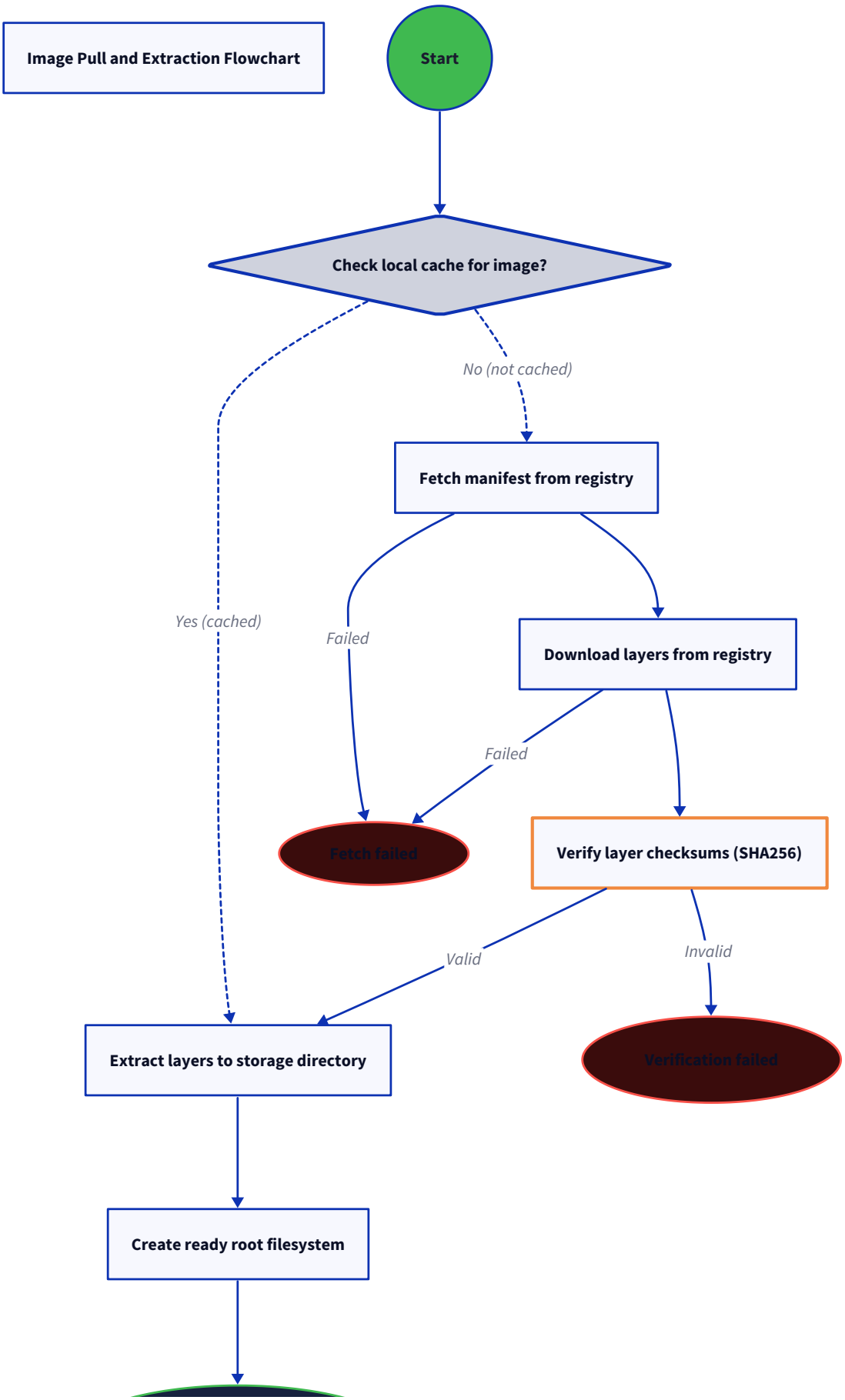
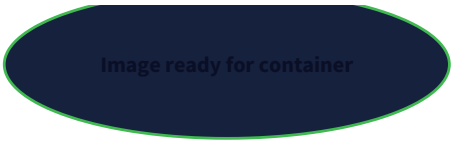


Image Pull and Extraction Flowchart
Start
Check local cache for image?
No (not cached)
Yes (cached)
Fetch manifest from registry
Failed
Download layers from registry
Failed
Fetch failed
Verify layer checksums (SHA256)
Valid
Invalid
Verification failed
Extract layers to storage directory
Create ready root filesystem



The following table details the data transformations at each stage:

Stage	Input	Processing	Output	Storage Location
Reference Parsing	`"ubuntu:latest"`	Parse registry, repo, tag	`{Registry: "docker.io", Repository: "library/ubuntu", Tag: "latest"}`	In-memory struct
Manifest Fetch	Repository, tag	HTTP GET to registry, parse JSON	`Manifest` with layer digests, config digest	`ImageStore` as blob (digest)
Layer Download	Layer digest	HTTP GET blob, SHA256 verify	Compressed tar blob	`ImageStore/base/blobs/sha256/<digest>`
Image Assembly	Manifest, all layers	Create `Image` with layer paths, config	Complete `Image` object	`ImageStore/index/<name>/image.json`
Rootfs Prep	`Image`, containerID	OverlayFS mount with layers	Merged directory path	`OverlayManager` temporary mount

> **Critical Insight:** Image layers are **content-addressable** and **immutable**. The same layer digest (SHA256 of its content) will always produce identical files. This enables global deduplication: if two images share a base layer (e.g., `alpine` base), it's stored once and referenced by both images, significantly saving disk space.

Common Failure Modes and Recovery:

Failure Mode	Detection	Recovery Strategy
Network timeout during layer pull	HTTP client timeout	Retry with exponential backoff (max 3 attempts)
Digest mismatch	SHA256 comparison fails	Delete corrupted blob, retry download
Disk full during extraction	`io.ErrShortWrite` or `ENOSPC`	Clean up partial extraction, return error
Registry authentication required	HTTP 401 Unauthorized	Prompt for credentials or use configured auth
Manifest not found	HTTP 404	Check tag exists, suggest alternative tags

Optimization: Layer Caching Strategy

- **Reference counting:** `LayerCache` tracks how many images/containers use each layer
- **LRU eviction:** When cache exceeds size limit, remove least recently used layers with zero references
- **Shared layers:** Multiple containers from same image share the same read-only lower layers, with separate upper writable layers
- **Concurrent pulls:** Download different layers in parallel, but serialize writes to same layer

Message and Configuration Formats

The container runtime uses three primary formats for communication: 1) **Internal Go structs** for component interaction, 2) **Persistent JSON files** for state and configuration storage, and 3) **External protocols** (OCI registry API) for image distribution. Understanding these formats is essential for implementing serialization, deserialization, and interoperability.

Internal Component Communication

Components communicate through method calls with the structured types defined in the Data Model section. The following table summarizes key method signatures and their data flows:

Method	Calling Component	Target Component	Input Data	Output Data	Purpose
`CreateContainer(config)`	CLI/User	Container Manager	`ContainerConfig`	`Container`	Initiate container creation
`SetupRootfs(image, id)`	Container Manager	Filesystem Manager	`Image`, containerID	rootfs path (string)	Prepare container filesystem
`CreateNamespaces(flags)`	Container Manager	Namespace Manager	namespace flags (int)	child PID (int)	Create isolated namespaces
`SetupNetwork(nsPath, config)`	Container Manager	Network Manager	nsPath (string), `NetworkConfig`	error	Configure container networking
`SetLimits(path, limits)`	Container Manager	Cgroup Manager	cgroup path (string), `ResourceLimits`	error	Apply resource constraints
`PullManifest(imageRef, tag)`	Image Handler	Registry Client	image reference, tag	`Manifest`, digest	Fetch image metadata
`StoreBlob(reader, digest)`	Registry Client	Image Store	io.Reader, expected digest	actual digest	Store and verify layer blob

Method Invocation Patterns:

- **Synchronous blocking:** Most methods block until completion (network setup, filesystem mount)
- **Error propagation:** All methods return `error`; callers must handle failures appropriately
- **Context passing:** Some long operations (image pull) accept `context.Context` for cancellation

Persistent Storage Formats

The runtime persists state in JSON files within the `CONTAINER_RUNTIME_ROOT` directory (typically `/var/lib/byd`). These files survive process restarts and enable recovery.

1. Container State File (`/containers/<id>/state.json`):

```json
{
  "ID": "a1b2c3d4e5f6",
  "Name": "my-ubuntu",
  "State": "running",
  "Config": {
    "Image": "ubuntu:latest",
    "Cmd": ["/bin/bash"],
    "Env": ["PATH=/usr/bin"],
    "WorkingDir": "/",
    "Hostname": "mycontainer",
    "Limits": {
      "MemoryMB": 100,
      "CPUShares": 512,
      "PidsLimit": 100
    },
    "Network": {
      "Mode": "bridge",
      "BridgeName": "byd0",
      "IPAddress": "10.0.0.2",
      "PortMappings": [
        {"HostPort": 8080, "ContainerPort": 80, "Protocol": "tcp"}
      ]
    }
  },
  "Pid": 12345,
  "CreatedAt": "2023-10-01T12:00:00Z"
}
```

2. Image Manifest File (`/images/<repo>/<tag>/manifest.json`): This follows the OCI Image Manifest Specification v1.0.2:

Field	Type	Description	Example
`schemaVersion`	int	OCI spec version	`2`
`mediaType`	string	MIME type of manifest	`application/vnd.oci.image.manifest.v1+json`
`config`	object	Descriptor for image config	`{"mediaType": "...", "digest": "sha256:...", "size": 1234}`
`layers`	array	Array of layer descriptors	`[{"mediaType": "...", "digest": "...", "size": ...}]`
`annotations`	object	Optional metadata	`{"org.opencontainers.image.created": "2023-10-01"}`

3. Image Configuration (`/blobs/sha256/<digest>`): This is the OCI Image Configuration JSON, parsed into our `ImageConfig` struct:

Field	Type	Description	Maps to `ImageConfig` Field
`architecture`	string	Target CPU architecture	(Not directly used)
`os`	string	Target OS	(Not directly used)
`config.Entrypoint`	array	Default executable	`Entrypoint`
`config.Cmd`	array	Default arguments	`Cmd`
`config.Env`	array	Environment variables	`Env`
`config.WorkingDir`	string	Working directory	`WorkingDir`
`rootfs.diff_ids`	array	Layer digests (uncompressed)	Used for layer verification

External Protocol: OCI Registry API

The runtime communicates with OCI/Docker registries using the Distribution Specification API:

Registry Endpoints:

- `GET /v2/` - Registry version check
- `GET /v2/<name>/manifests/<reference>` - Fetch manifest (reference can be tag or digest)
- `GET /v2/<name>/blobs/<digest>` - Fetch layer/blob content
- `HEAD /v2/<name>/blobs/<digest>` - Check if blob exists (layer cache validation)

HTTP Headers for Content Negotiation:

- `Accept: application/vnd.oci.image.manifest.v1+json, application/vnd.docker.distribution.manifest.v2+json`
- `Authorization: Bearer <token>` (for authenticated registries)
- `Docker-Content-Digest: sha256:...` (in responses, for verification)

Authentication Flow:

1. Initial request often returns `401 Unauthorized` with `WWW-Authenticate` header
2. Parse header for realm, service, scope
3. Request bearer token from authentication server
4. Retry original request with `Authorization: Bearer <token>`

Message Flow Example (Pull Manifest):

```
Client → GET /v2/library/ubuntu/manifests/latest
Registry → 200 OK
           Content-Type: application/vnd.oci.image.manifest.v1+json
           Docker-Content-Digest: sha256:abc123...

           {
             "schemaVersion": 2,
             "mediaType": "application/vnd.oci.image.manifest.v1+json",
             "config": {"digest": "sha256:def456...", "size": 1234},
             "layers": [
               {"digest": "sha256:layer1...", "size": 567890}
             ]
           }
```

Configuration Inheritance and Overrides

The runtime merges configuration from multiple sources with clear precedence:

1. **Base Image Configuration** (`ImageConfig` from OCI image) - Lowest priority
2. **Container Defaults** (runtime defaults for security, resources) - Medium priority
3. **User Configuration** (`ContainerConfig` from CLI/flags) - Highest priority
4. **Runtime Enforcement** (security hardening, required mounts) - Overrides all

Example merge for command execution:

- Image defines: `Entrypoint: ["/bin/sh"]` , `Cmd: ["-c", "echo hello"]`
- User provides: `Cmd: ["/bin/bash"]`

- Result: `["/bin/sh", "/bin/bash"]` (Docker behavior: user `Cmd` replaces image `Cmd`, not `Entrypoint`)

The configuration resolution is implemented in `ContainerRuntime.CreateContainerFromImage()`:

```go
// Pseudo-code for configuration merging
func mergeConfig(imageConfig ImageConfig, userConfig ContainerConfig) ContainerConfig {
    result := ContainerConfig{}

    // Command: user Cmd overrides image Cmd, but Entrypoint from image unless user specifies
    if len(userConfig.Cmd) > 0 {
        result.Cmd = userConfig.Cmd
        // Keep image Entrypoint unless user provided both
    } else {
        result.Cmd = imageConfig.Cmd
    }

    // Environment: merge with user values overriding image values
    result.Env = mergeEnv(imageConfig.Env, userConfig.Env)

    // Working directory: user overrides image
    if userConfig.WorkingDir != "" {
        result.WorkingDir = userConfig.WorkingDir
    } else {
        result.WorkingDir = imageConfig.WorkingDir
    }

    return result
}
```

Inter-Process Communication

For the parent-child coordination during container creation, the runtime uses:

1. **`/proc/self/exe` re-execution:** The parent re-executes itself with a `--child` flag and the containerID
2. **Environment variables:** Parent passes configuration via `BYD_CONTAINER_ID`, `BYD_ROOTFS_PATH`
3. **File descriptors:** Network namespace path, cgroup directories can be passed via inherited FDs
4. **Signals:** Parent monitors child via `SIGCHLD`, sends `SIGTERM` / `SIGKILL` for stop operations

This IPC approach avoids complex serialization and maintains simplicity while crossing privilege boundaries.

Implementation Guidance

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Component Communication	Direct method calls (in-process)	gRPC with Protocol Buffers (for daemon mode)
Persistent Storage	JSON files with `encoding/json`	SQLite database with migrations
Registry Protocol	HTTP/1.1 with net/http	HTTP/2 with connection pooling, parallel downloads
Configuration Merging	Manual field-by-field merging	JSON Merge Patch (RFC 7396) or strategic merge

B. Recommended File/Module Structure:

```
byd/                          # Project root
├── cmd/
│   └── byd/                  # CLI entry point
│       └── main.go
├── internal/
│   ├── runtime/              # Container Manager component
│   │   ├── runtime.go        # ContainerRuntime type and main logic
│   │   ├── state_manager.go  # Container state transitions
│   │   └── sequence.go       # Container creation sequence (this section)
│   ├── store/                # Persistent storage
│   │   ├── container_store.go # ContainerStore implementation
│   │   ├── image_store.go    # ImageStore implementation
│   │   └── layer_cache.go    # LayerCache implementation
│   ├── images/               # Image handling (Milestone 6)
│   │   ├── registry.go       # RegistryClient
│   │   ├── oci_parser.go     # OCI manifest/config parsing
│   │   └── pull_sequence.go  # Image pull flow
│   └── utils/
│       └── uuid.go           # ID generation helpers
└── pkg/                      # Public API (if needed)
    └── types/
        └── types.go          # All data model structs
```

C. Infrastructure Starter Code:

Complete UUID Generator (`pkg/utils/uuid.go`):

```go
package utils

import (
	"crypto/rand"
	"encoding/hex"
	"fmt"
	"sync"
)

var (
	mu        sync.Mutex
	shortID   string
	shortOnce sync.Once
)

// Generate creates a random 128-bit hex ID (32 characters)
func Generate() (string, error) {
	bytes := make([]byte, 16)
	if _, err := rand.Read(bytes); err != nil {
		return "", fmt.Errorf("generate uuid: %w", err)
	}
	return hex.EncodeToString(bytes), nil
}

// Short generates a random 64-bit hex ID (16 characters) with caching
func Short() string {
	shortOnce.Do(func() {
		bytes := make([]byte, 8)
		if _, err := rand.Read(bytes); err != nil {
			// Fallback to timestamp-based ID if crypto fails
			shortID = fmt.Sprintf("%x", time.Now().UnixNano())
			return
		}
		shortID = hex.EncodeToString(bytes)
	})
	return shortID
}
```

Container State Manager (`internal/runtime/state_manager.go`):

```go
package runtime

import (
	"fmt"
	"sync"
	"time"

	"byd/pkg/types"
)

type StateManager struct {
	mu         sync.RWMutex
	containers map[string]*types.Container
	store      ContainerStore
}

func NewStateManager(storePath string) (*StateManager, error) {
	store, err := NewContainerStore(storePath)
	if err != nil {
		return nil, fmt.Errorf("create state manager: %w", err)
	}

	// Load existing containers from disk
	containers := make(map[string]*types.Container)
	// Implementation: scan store directory, load each container

	return &StateManager{
		containers: containers,
		store:      store,
	}, nil
}

// Transition validates and executes state changes
func (sm *StateManager) Transition(containerID string,
		fromState, toState types.ContainerState) error {
	sm.mu.Lock()
	defer sm.mu.Unlock()

	container, exists := sm.containers[containerID]
	if !exists {
```

```
                return fmt.Errorf("container %s not found", containerID)
        }

        if container.State != fromState {
                return fmt.Errorf("invalid state transition: current %s, expected %s",
                        container.State, fromState)
        }

        // Validate transition
        if !isValidTransition(fromState, toState) {
                return fmt.Errorf("invalid transition from %s to %s", fromState, toState)
        }

        // Update state
        container.State = toState
        if toState == types.StateRunning {
                // Set started timestamp if needed
        }

        // Persist
        if err := sm.store.Save(container); err != nil {
                // Revert in-memory state on persistence failure
                container.State = fromState
                return fmt.Errorf("save container state: %w", err)
        }

        return nil
}

func isValidTransition(from, to types.ContainerState) bool {
        // Define valid transitions based on state machine diagram
        transitions := map[types.ContainerState][]types.ContainerState{
                types.StateCreated:  {types.StateRunning, types.StateStopped},
                types.StateRunning:  {types.StatePaused, types.StateStopped},
                types.StatePaused:   {types.StateRunning, types.StateStopped},
                types.StateStopped:  {types.StateRemoved},
                types.StateRemoved:  {}, // Terminal state
        }
```

```
        for _, validTo := range transitions[from] {
                if validTo == to {
                        return true
                }
        }
        return false
}
```

D. Core Logic Skeleton Code:

Container Creation Sequence (`internal/runtime/sequence.go`):

```go
package runtime

import (
	"fmt"
	"os"
	"path/filepath"

	"byd/pkg/types"
)

// createContainerSequence orchestrates the 8-phase container creation
// Returns the created Container and any error encountered
func (cr *ContainerRuntime) createContainerSequence(
	config types.ContainerConfig,
	name string) (*types.Container, error) {

	// Phase 1: Validation and Preparation
	// TODO 1: Validate config fields (image exists, limits positive, etc.)
	// TODO 2: Generate containerID using utils.Generate()
	// TODO 3: Create container entry with StateCreated
	// TODO 4: Save to ContainerStore

	// Phase 2: Image Preparation
	// TODO 5: Check if image exists locally via ImageStore
	// TODO 6: If not, call cr.PullImage() (this may be long-running)
	// TODO 7: Load Image object from store
	// TODO 8: Call FilesystemManager.SetupRootfs() to prepare root filesystem

	// Phase 3: Namespace Creation
	// TODO 9: Determine namespace flags from config (PID, UTS, mount, network, IPC)
	// TODO 10: Call NamespaceManager.CreateNamespaces() with flags
	// TODO 11: Capture returned child PID

	// Phase 4: Cgroup Configuration
	// TODO 12: Call CgroupManager.CreateGroup() with containerID
	// TODO 13: Call CgroupManager.SetLimits() with config.Limits
	// TODO 14: Call CgroupManager.AddProcess() to move child PID into cgroup

	// Phase 5: Network Setup
```

```
        // TODO 15: Construct network namespace path: fmt.Sprintf("/proc/%d/ns/net", childPID)
        // TODO 16: Call NetworkManager.SetupNetwork() with nsPath and config.Network
        // TODO 17: Handle error - if fails, roll back previous phases

        // Phase 6: Filesystem Isolation (inside child)
        // This is handled by the child process via ChildEntryPoint
        // The parent just waits for child to signal ready or error

        // Phase 7: Process Execution (inside child)
        // Handled by ChildEntryPoint which calls exec()

        // Phase 8: State Transition
        // TODO 18: Update container state to StateRunning
        // TODO 19: Set container.Pid = childPID
        // TODO 20: Save updated container to store
        // TODO 21: Start monitoring goroutine for container exit

        // Error handling: implement rollback for each phase
        // Use defer with named return values to track if sequence succeeded

        return nil, fmt.Errorf("not implemented")
}

// ChildEntryPoint is the function executed inside the new namespaces
// It's called via re-execution of the binary with --child flag
func ChildEntryPoint() int {
        // TODO 1: Parse containerID from environment (BYD_CONTAINER_ID)
        // TODO 2: Load container from store
        // TODO 3: Call FilesystemManager.IsolateFilesystem() with container's rootfs
        // TODO 4: Set hostname using syscall.Sethostname()
        // TODO 5: Set up environment variables from container.Config.Env
        // TODO 6: Change working directory to container.Config.WorkingDir
        // TODO 7: Determine command to execute (merge image and user config)
        // TODO 8: Perform syscall.Exec() to replace this process with container command
        // TODO 9: If exec fails, exit with error code; otherwise never returns

        return 1 // Error exit if we reach here
}
```

Image Pull Sequence (`internal/images/pull_sequence.go`):

```go
package images


// PullImage orchestrates downloading an image from registry
func (ih *ImageHandler) PullImage(imageRef string) (*types.Image, error) {
        // TODO 1: Parse image reference into registry, repository, tag
        // TODO 2: Check local cache for existing image with same digest
        // TODO 3: Fetch manifest from registry using RegistryClient.PullManifest()
        // TODO 4: Verify manifest structure and media type
        // TODO 5: For each layer in manifest.Layers:
        //   TODO 5a: Check LayerCache for existing layer with same digest
        //   TODO 5b: If not cached, download using RegistryClient.PullLayer()
        //   TODO 5c: Verify SHA256 digest matches
        //   TODO 5d: Store blob in ImageStore.StoreBlob()
        //   TODO 5e: Add to LayerCache with AddReference()
        // TODO 6: Download image configuration blob
        // TODO 7: Parse into ImageConfig struct
        // TODO 8: Create Image object with all layers and config
        // TODO 9: Store image in ImageStore with tag reference
        // TODO 10: Return complete Image object


        return nil, fmt.Errorf("not implemented")
}
```

E. Language-Specific Hints:

- **Go concurrency:** Use goroutines for parallel layer downloads but limit concurrency with a worker pool
- **Error wrapping:** Use `fmt.Errorf("step: %w", err)` for error chains that show the sequence of failures
- **JSON serialization:** Use struct tags: `` `json:"fieldName,omitempty"` `` for proper serialization
- **File locking:** Use `flock` or directory-based locking for atomic operations on container state
- **Context propagation:** Pass `context.Context` through long operations (network downloads) for cancellation
- **Cleanup with defer:** Structure functions with `defer cleanup()` pattern, but be careful with early returns

F. Milestone Checkpoint: After implementing the container creation sequence, verify with:

```bash
# Build and test
go build ./cmd/byd
sudo ./byd run --rm alpine:latest echo "Hello from container"

# Expected behavior:
# 1. Downloads alpine image (if first time)
# 2. Creates container with isolated namespaces
# 3. Runs echo command
# 4. Prints "Hello from container"
# 5. Container exits and is cleaned up (due to --rm)

# Verification commands:
ps aux | grep byd          # Should show only the byd process, not container processes
ls /sys/fs/cgroup/         # Should see byd-* cgroups created and cleaned up
ip link show               # Should see veth interfaces created and removed
```

G. Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Container exits immediately with code 1	`exec()` failure in child	Check child logs, verify binary exists in rootfs	Ensure rootfs has the executable at correct path
Network setup fails with "device busy"	veth interface already exists	`ip link show` to see conflicting names	Use unique interface names, clean up old interfaces
Cgroup limits not enforced	Process not in cgroup	Check `/proc/<pid>/cgroup` file	Ensure `AddProcess()` is called before process starts
Rootfs appears empty after mount	OverlayFS mount failed	Check `dmesg` for overlay errors	Verify lowerdir paths exist, work dir is empty
Registry pull returns 404	Invalid image reference	Check registry API response	Use full image name: `docker.io/library/alpine:latest`
State file corruption	Concurrent writes to same file	Check file timestamps, use file locking	Implement atomic writes (write to temp, then rename)

Error Handling and Edge Cases

> **Milestone(s):** All milestones (error handling and edge cases are relevant throughout the entire system, from low-level system calls to high-level user operations)

This section defines the comprehensive error handling philosophy and edge case management strategy for the container runtime. Unlike application-level software that can often rely on the operating system to clean up after failures, container runtimes operate at the system level where partial failures can leave resources (namespaces, cgroups, mounts, network interfaces) orphaned and system state inconsistent. The key architectural challenge is designing **idempotent, atomic, and recoverable operations** despite the inherent statefulness of Linux kernel primitives.

Common Failure Modes and Detection

Container runtime failures can be categorized by their **origin layer**: kernel/system call failures, resource exhaustion, user configuration errors, race conditions, and external dependencies. Each category requires distinct detection strategies.

> **Design Insight**: The runtime must assume that any operation can fail at any point, and the failure detection must be **immediate and precise**. Delayed or vague error detection leads to resource leaks and security vulnerabilities.

System Call and Kernel Interface Failures

These are the most fundamental failures, occurring when Linux kernel interfaces reject our requests due to permissions, invalid arguments, or resource constraints.

Failure Mode	Detection Strategy	Impact
Namespace creation failure (`clone()`, `unshare()` returns -1)	Check `errno` after syscall: `EPERM` (no privileges), `EINVAL` (invalid flags), `ENOMEM` (kernel memory)	Container cannot start; partial isolation may leave process in host namespace
cgroup filesystem operations fail (`mkdir`, `write` to control files)	Check file operation errors: `EACCES` (permissions), `ENOSPC` (device full), `EROFS` (read-only)	Resource limits not applied; container may consume unbounded resources
Mount operations fail (`mount()`, `pivot_root()`, `umount()`)	Check mount return value and `errno`: `EINVAL` (invalid option), `EPERM` (no CAP_SYS_ADMIN), `EBUSY` (busy mount)	Filesystem isolation incomplete; container may access host filesystem
Network namespace operations fail (`socket()`, `ioctl()`, netlink errors)	Check network syscall returns and netlink error codes	Network connectivity broken; container may be unreachable or have incorrect routing
Process execution fails (`execve()` returns -1)	Check `errno`: `ENOENT` (binary not found), `EACCES` (permission denied), `ENOTDIR` (path component not directory)	Container starts but immediately exits; init process fails

Detection Implementation: Each component manager (`NamespaceManager`, `CgroupManager`, `FilesystemManager`, `NetworkManager`) must wrap system calls with explicit error checking that preserves the original `errno` value. The Go standard library's `syscall` package returns error types that include the raw errno, which should be inspected for precise diagnosis.

Resource Exhaustion Failures

The runtime itself, or containers it manages, can exhaust system resources. These failures often manifest as "soft" failures where operations succeed but with degraded performance, eventually leading to "hard" failures.

Failure Mode	Detection Strategy	Impact
Memory exhaustion during image extraction	Monitor `extractLayer` for `ENOSPC` on disk write or `ENOMEM` when decompressing	Partial layer extraction; incomplete root filesystem
Inode or file descriptor exhaustion	Check for `EMFILE` (process FD limit) or `ENFILE` (system FD limit)	Cannot create pipes, sockets, or open files; runtime becomes inoperable
PID exhaustion in cgroup	Monitor `fork()`/`clone()` for `EAGAIN` when cgroup pids.max reached	Cannot create new processes in container; fork bombs partially contained
IP address pool exhaustion	`IPAM.Allocate()` returns error when subnet exhausted	Container cannot get network connectivity; start fails
Disk space for upper layer (OverlayFS)	Write to upper layer fails with `ENOSPC`	Container filesystem becomes read-only; writes fail

Detection Implementation: Proactive monitoring is better than reactive failure. The runtime should check resource availability before operations: `CgroupManager` should read cgroup limits and current usage before container start; `IPAM` should track free addresses; `FilesystemManager` should check available disk space in storage directories.

User Configuration and Input Errors

These failures stem from invalid user input, misconfigured images, or incompatible system configurations.

Failure Mode	Detection Strategy	Impact
Invalid image reference format	Parse image reference with OCI distribution spec regex	Cannot pull image; operation fails early
Missing or malformed OCI manifest	Validate JSON schema, required fields, and layer digests	Image cannot be used; rootfs cannot be constructed
Invalid resource limits (negative memory, zero CPU shares)	Validate `ResourceLimits` fields with range checks	cgroup setup fails or applies incorrect limits
Conflicting port mappings	Check `PortMappings` for duplicate `HostPort` values	Network setup fails or creates conflicting iptables rules
Missing capabilities for requested isolation	Check `CAP_SYS_ADMIN`, `CAP_NET_ADMIN`, etc. at runtime start	Namespace creation fails with `EPERM`

Detection Implementation: Comprehensive validation at API boundaries. The `ContainerRuntime.CreateContainer` method should validate all fields of `ContainerConfig` before any side effects. Image validation should occur during `ImageStore.StoreManifest` and layer extraction.

Race Conditions and State Inconsistencies

Concurrent operations or unexpected external state changes can cause failures even when individual operations succeed.

Failure Mode	Detection Strategy	Impact
Container already running when start attempted	Check `Container.State` in `StateManager.Transition()`	State transition rejected; prevents double execution
Container removed while being stopped	Check container exists after acquiring lock	Orphaned resources; potential use-after-free
Network namespace deleted by OS before cleanup	Check `/proc/self/ns/net` existence before join	Runtime crash when trying to operate on non-existent namespace
Image layer garbage collected while container running	Reference counting in `LayerCache` prevents deletion	Container rootfs becomes broken; read operations fail
cgroup directory removed externally	Stat cgroup path before writing control files	Resource limits lost; process escapes constraints

Detection Implementation: State must be protected by synchronization primitives. The `StateManager` uses `sync.RWMutex` to serialize state transitions. Resource managers should verify resource existence before operations (defensive programming). Reference counting (`LayerCache.AddReference/ReleaseReference`) prevents premature deletion of shared resources.

External Dependency Failures

The runtime depends on external systems (Docker registry, DNS, internet connectivity) that may be unavailable.

Failure Mode	Detection Strategy	Impact
Registry authentication failure	HTTP 401/403 from registry API	Cannot pull private images
Registry network timeout	HTTP client timeout with context deadline	Image pull hangs or fails after retries
DNS resolution failure during network setup	`resolv.conf` creation fails if nameservers unreachable	Container has network but cannot resolve hostnames
iptables lock contention or missing modules	Check `iptables` command exit code	NAT or port forwarding not configured
OverlayFS kernel module not loaded	Check `/proc/filesystems` for overlay entry	Layered filesystem unavailable; must fall back to plain directory

Detection Implementation: Network operations should use contexts with timeouts. Registry client should implement exponential backoff for transient failures. System dependency checks should occur at runtime initialization (`NewContainerRuntime` should validate kernel features and available commands).

Recovery and Cleanup Strategies

The container runtime must implement **compensating transactions**—for every operation that modifies system state, there must be a corresponding cleanup operation that can be executed even after partial failures. The strategy follows a **cleanup stack** pattern where operations push cleanup functions onto a stack

as they succeed, and failures unwind the stack.

State-Driven Cleanup Orchestration

Each container has a well-defined state machine (`ContainerState` : `Created` , `Running` , `Paused` , `Stopped` , `Removed`). The cleanup strategy differs by which state transition fails:

Failed Transition	Cleanup Strategy	Rollback Actions
Nothing → Created (container creation fails)	Delete all allocated resources	1. Delete cgroup directory 2. Delete container metadata from `ContainerStore` 3. Release IP address from `IPAM`
Created → Running (start fails)	Roll back to `Created` state, keep resources allocated	1. Kill any partially started process 2. Leave namespaces, cgroups, network intact for retry 3. Unmount any temporary mounts (e.g., /proc in wrong namespace)
Running → Stopped (stop fails or timeout)	Force kill with SIGKILL, then cleanup	1. Send SIGKILL to container process tree 2. Wait for process termination 3. Perform full cleanup as if stop succeeded
Stopped → Removed (remove fails)	Retry with more aggressive cleanup	1. Force unmount lingering mounts (MNT_DETACH) 2. Force delete cgroup directory (write `cgroup.kill` for cgroup v2) 3. Force remove network namespace by killing processes inside

Implementation Pattern: The `createContainerSequence` function (from Milestone 6) should implement the cleanup stack:

```go
// Pseudo-code structure (actual code in Implementation Guidance)
var cleanupTasks []func()
defer func() {
    if err != nil {
        // Execute cleanup in reverse order (LIFO)
        for i := len(cleanupTasks)-1; i >= 0; i-- {
            cleanupTasks[i]()
        }
    }
}()

// Each successful operation pushes its cleanup
cleanupTasks = append(cleanupTasks, func() { cgroupManager.RemoveCgroup(containerID) })
cleanupTasks = append(cleanupTasks, func() { networkManager.CleanupNetwork(containerID, config) })
// ... etc
```

Idempotent Cleanup Operations

All cleanup operations must be **idempotent**—calling them multiple times or calling them when resources no longer exist should not error. This property is crucial for recovery after crashes where the runtime doesn't know which cleanup operations have already completed.

Operation	Idempotent Implementation
cgroup deletion	Check if cgroup directory exists before removal; ignore `ENOENT` errors
network namespace cleanup	Check if network namespace path exists; skip if already gone
mount unmounting	Try unmount, ignore `EINVAL` (not mounted) and `ENOENT` (mount point gone)
IP address release	`IPAM.Release` should check if IP was allocated; no-op if not in allocated map
filesystem cleanup	`RemoveAll` on container directory, tolerate missing directories

Rationale: After a runtime crash and restart, the recovery routine will attempt to clean up all containers that were in progress. Idempotent cleanup ensures the system converges to a clean state regardless of how many times cleanup is attempted.

Orphan Detection and Garbage Collection

Despite best efforts, some resources may become orphaned (e.g., if runtime is SIGKILLed during namespace creation). The runtime must include a **garbage collection** system that runs at startup and periodically to reclaim lost resources.

Resource Type	Orphan Detection Method	Cleanup Action
cgroups	Scan cgroup filesystem for directories matching container ID pattern	Remove directory if no processes inside
network namespaces	Scan `/var/run/netns` for namespaces not referenced by any container	Unlink namespace file (only removes reference, not namespace with processes)
mounts	Parse `/proc/self/mountinfo` for mounts under runtime storage path	Unmount with `MNT_DETACH` flag
OverlayFS layers	Scan layer cache for unreferenced layers	Delete layer directories after ensuring no containers reference them
veth interfaces	Scan network interfaces for veth* interfaces not in any bridge	Delete interface with `ip link delete`

Implementation Approach: The `ContainerRuntime` should maintain a **reaper** goroutine that periodically scans for orphans. At startup, before any new container operations, the runtime should run a full garbage collection to clean up from previous crashes.

Transaction Logging for Crash Recovery

For critical multi-step operations (like container creation), the runtime can write intentions to a **transaction log** before performing operations. If the runtime crashes mid-operation, upon restart it reads the log to know which operations were in progress and can complete or roll them back.

Operation Phase	Log Entry	Recovery Action
Start container creation	`{"op": "create", "id": "abc", "state": "allocating"}`	If found at recovery, container was being created; run cleanup
After cgroup created	`{"op": "create", "id": "abc", "state": "cgroup_created"}`	Cleanup must remove cgroup
After network setup	`{"op": "create", "id": "abc", "state": "network_created"}`	Cleanup must remove network resources
After rootfs prepared	`{"op": "create", "id": "abc", "state": "rootfs_ready"}`	Cleanup must unmount rootfs
Completion	Remove log entry	No recovery needed

Trade-off: While transaction logging provides robust recovery, it adds complexity. For this educational implementation, we opt for the simpler cleanup stack approach, noting that production systems would need transaction logging.

Edge Case Scenarios

Edge cases represent unusual but possible situations that the runtime must handle gracefully. These often involve boundary conditions, unusual user behavior, or unexpected system configurations.

Container Process Behavior Edge Cases

Scenario	Behavior	Handling Strategy
Container init process exits immediately (echo "hello")	Container starts and stops within milliseconds	Runtime must still capture exit code, trigger cleanup, transition to `Stopped` state
Container process forks daemon children and exits	PID 1 exits but other processes remain in namespace	Runtime should kill entire process tree when stopping container; use cgroup process tracking
Container process calls `unshare()` itself	Nested namespaces inside container	Runtime's namespaces remain outer layer; child namespaces are container's business
Container process mounts /proc over container /proc	Overwrites runtime's /proc mount	Runtime should mount /proc with `MS_PRIVATE` to prevent propagation, but cannot prevent container from remounting
Container process writes to /sys files that affect host	e.g., writing to /sys/class/backlight	Runtime should mount /sys as read-only (`MS_RDONLY`) where possible
Container runs setuid binary that escapes namespace	Without user namespace, setuid binaries run with host privileges	Consider implementing user namespace mapping (advanced) or warn users

Implementation Guidance: The runtime should place the container process in a cgroup and use the cgroup's process tracking to ensure all descendant processes are accounted for. When stopping, signal should be sent to the entire process tree (using negative PID to process group).

Filesystem Edge Cases

Scenario	Behavior	Handling Strategy
Base image has broken symlinks	Symlinks pointing outside rootfs or to non-existent files	Preserve symlinks as-is; container will get ENOENT when following
OverlayFS copy-up on directory rename	Renaming a directory requires copying entire directory tree	Accept performance hit; OverlayFS handles this automatically
Multiple containers sharing same image layer	Concurrent write to same file in shared layer	OverlayFS ensures each container gets its own copy in upper layer
Container writes to lower-layer whiteout file	Writing to `.wh..wh..opq` or `.wh.<filename>`	OverlayFS prevents this; runtime doesn't need special handling
Rootfs contains mount points (bind mounts in image)	These mounts propagate to container if not properly isolated	Runtime should ensure mount namespace is created before rootfs setup
/proc mounted but shows host processes	Forgot to mount /proc after `CLONE_NEWPID`	Runtime must mount new /proc inside container after PID namespace creation

Critical Insight: The order of filesystem operations matters tremendously. The sequence must be: 1) Create mount namespace, 2) Mount rootfs, 3) Mount /proc, /sys, /dev, 4) `pivot_root()` , 5) Unmount old root. Deviations cause visibility of host processes or files.

Networking Edge Cases

Scenario	Behavior	Handling Strategy
Container tries to configure its own IP address	Process inside calls `ip addr add` on veth interface	Allowed; container owns its network namespace
Host network namespace disappears (unlikely)	Bridge or veth host-side deleted by admin	Container loses connectivity; runtime cannot repair external changes
Container uses raw sockets to send spoofed packets	Without dropping capabilities, container can send arbitrary packets	Drop `CAP_NET_RAW` capability from container by default
DNS resolution inside container fails but host DNS works	/etc/resolv.conf not properly configured	Runtime should copy host's resolv.conf or use well-known DNS servers
Port mapping conflict with host service	Container port 80 maps to host port 80 already in use	Check host port availability before setting up iptables rule; fail fast
Multiple containers requesting same static IP	Two containers configured with same `IPAddress`	`IPAM.Allocate` should detect conflict and return error

Implementation Note: Network isolation is only as strong as the capability dropping. The runtime should drop dangerous capabilities (`CAP_NET_RAW`, `CAP_NET_ADMIN`) from the container unless explicitly requested by the user.

Image and Storage Edge Cases

Scenario	Behavior	Handling Strategy
Image layer tar has same file in multiple layers	Last layer wins; lower layer files are obscured	OverlayFS correctly handles this; file from upper layer (later layer) appears
Image manifest lists multiple architectures	Manifest list contains amd64, arm64, etc. entries	Runtime should select based on host architecture or fail with clear error
Layer download interrupted and partially written	Partial .tar file in cache	Validate SHA256 digest after download; redownload if mismatch
Image config specifies invalid entrypoint (non-existent binary)	`execve` fails at container start	Container fails to start; return meaningful error about missing binary
Two images share layers with different digests (impossible by definition)	Content-addressable storage ensures same content = same digest	If digests differ, content differs; treat as different layers
Image with :latest tag updates while container running	New image pulled doesn't affect running containers	Running containers use extracted layers; new containers use new image

Content Integrity: The runtime must verify layer digests after download and after extraction. A corrupted layer should be re-downloaded automatically.

Security and Permission Edge Cases

Scenario	Behavior	Handling Strategy
Non-root user runs runtime	Most operations require `CAP_SYS_ADMIN`	Fail early with clear message about needing root/sudo
Container process gains root via setuid binary	Root inside container ≠ root on host (without user ns)	Without user namespace, container root can do damage; warn users
Host filesystem mounted inside container (bind mount)	Container can modify host files if mounted rw	Only allow bind mounts from explicit host paths with user consent
Container escapes via /proc/self/ns/ join	Process joins host namespace via leaked fd	Keep namespace fds private to runtime; don't expose to container
Resource limit bypass via fork bomb	Process creates many children quickly	cgroup pids.max prevents this; ensure limit is set before process starts

Security Principle: The runtime should follow the principle of least privilege. Drop all capabilities not explicitly needed, mount filesystems as read-only where possible, and use user namespaces if implemented.

Cross-Milestone Integration Edge Cases

These edge cases emerge when components from different milestones interact in unexpected ways:

Scenario	Affected Milestones	Handling Strategy
Network namespace created but bridge not set up	Milestone 1 + 5	Container has isolated network but no connectivity; runtime should ensure bridge exists or create it
cgroup limits set but process not moved into cgroup	Milestone 2 + 1	Process escapes limits; must call `cgroupManager.AddProcess` after fork but before exec
Rootfs mounted but /proc not mounted in PID ns	Milestone 3 + 1	ps, top show host processes; must mount /proc after entering PID namespace
OverlayFS mounted but workdir not empty	Milestone 4	mount fails with `ENOTEMPTY`; runtime should create fresh workdir or clear it
Image pulled but layers extracted to wrong location	Milestone 6 + 4	OverlayFS can't find lowerdirs; use consistent paths: `/var/lib/runtime/layers/<digest>`
Container stopped but network namespace not cleaned up	Milestone 5 + 6	veth interface orphaned; cleanup must be part of container removal, not just stop

Integration Testing: The runtime needs comprehensive integration tests that verify all milestones work together correctly. Each edge case should have a corresponding test case.

Implementation Guidance

This section provides concrete implementation patterns for error handling and cleanup in Go, with complete starter code for the cleanup stack pattern.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Error Propagation	Return `error` with context using `fmt.Errorf` and `%w`	Structured errors with custom error types implementing `Is()` and `As()`
Cleanup Orchestration	Defer-based cleanup stack	Transaction log with recovery scanner
Resource Tracking	In-memory maps with mutexes	Database (SQLite) for persistent state
Orphan Detection	Periodic scanner goroutine	inotify watches on resource directories
State Recovery	Rebuild state from container directories	Write-ahead log of all state changes

B. Recommended File/Module Structure

```
project-root/
  internal/
    runtime/
      errors.go              # Custom error types and error handling utilities
      cleanup.go             # Cleanup stack implementation
      recovery.go            # Orphan detection and garbage collection
      state_manager.go       # State transitions with error handling
    cgroup/
      manager.go             # cgroup operations with idempotent cleanup
    network/
      manager.go             # Network setup/cleanup with rollback
    filesystem/
      manager.go             # Mount operations with proper unmount on error
    image/
      store.go               # Image validation and integrity checking
  cmd/
    runtime/
      main.go               # Runtime initialization with startup recovery
```

C. Infrastructure Starter Code

Here's complete, working code for the cleanup stack pattern that can be used throughout the runtime:

```go
// internal/runtime/cleanup.go

package runtime


import (

	"fmt"

	"sync"

)


// CleanupFunc is a function that cleans up resources.

// It should be idempotent (safe to call multiple times).

type CleanupFunc func()


// CleanupStack manages a stack of cleanup functions to be executed in reverse order.

type CleanupStack struct {

	funcs []CleanupFunc

	mu    sync.Mutex

	done  bool

}


// NewCleanupStack creates a new CleanupStack.

func NewCleanupStack() *CleanupStack {

	return &CleanupStack{

		funcs: make([]CleanupFunc, 0),

	}

}


// Push adds a cleanup function to the stack.

func (cs *CleanupStack) Push(f CleanupFunc) {

	cs.mu.Lock()

	defer cs.mu.Unlock()

	if cs.done {

		panic("cleanup stack already executed")

	}

	cs.funcs = append(cs.funcs, f)

}


// Execute runs all cleanup functions in reverse order (LIFO).

// After execution, the stack is marked as done and cannot be used again.

func (cs *CleanupStack) Execute() {

	cs.mu.Lock()

	defer cs.mu.Unlock()
```

```
        if cs.done {
                return
        }
        cs.done = true

        // Execute in reverse order (LIFO)
        for i := len(cs.funcs) - 1; i >= 0; i-- {
                cs.funcs[i]()
        }
        cs.funcs = nil // Allow garbage collection
}

// DeferOnError returns a function that executes the cleanup stack if the error is non-nil.
// Usage: defer cleanup.DeferOnError(&err)()
func (cs *CleanupStack) DeferOnError(err *error) func() {
        return func() {
                if *err != nil {
                        cs.Execute()
                }
        }
}

// CleanupManager orchestrates cleanup across the entire runtime.
type CleanupManager struct {
        containerStacks map[string]*CleanupStack
        globalStack     *CleanupStack
        mu              sync.RWMutex
}

// NewCleanupManager creates a new CleanupManager.
func NewCleanupManager() *CleanupManager {
        return &CleanupManager{
                containerStacks: make(map[string]*CleanupStack),
                globalStack:     NewCleanupStack(),
        }
}

// RegisterContainerCleanup creates a cleanup stack for a container.
func (cm *CleanupManager) RegisterContainerCleanup(containerID string) *CleanupStack {
        cm.mu.Lock()
```

```
        defer cm.mu.Unlock()

        stack := NewCleanupStack()

        cm.containerStacks[containerID] = stack

        return stack

}


// ExecuteContainerCleanup runs cleanup for a specific container.

func (cm *CleanupManager) ExecuteContainerCleanup(containerID string) {

        cm.mu.Lock()

        defer cm.mu.Unlock()

        if stack, ok := cm.containerStacks[containerID]; ok {

                stack.Execute()

                delete(cm.containerStacks, containerID)

        }

}


// ExecuteAllCleanup runs cleanup for all containers and global resources.

func (cm *CleanupManager) ExecuteAllCleanup() {

        cm.mu.Lock()

        defer cm.mu.Unlock()



        // Clean up all containers

        for containerID, stack := range cm.containerStacks {

                stack.Execute()

                delete(cm.containerStacks, containerID)

        }



        // Clean up global resources

        cm.globalStack.Execute()

}
```

D. Core Logic Skeleton Code

Here's the skeleton for the container creation sequence with integrated error handling and cleanup:

```go
// internal/runtime/container_creation.go

package runtime


import (

        "fmt"



        "github.com/your-username/container-runtime/internal/cgroup"

        "github.com/your-username/container-runtime/internal/network"

        "github.com/your-username/container-runtime/internal/filesystem"

        "github.com/your-username/container-runtime/internal/types"

)


func (r *ContainerRuntime) createContainerSequence(config types.ContainerConfig, name string) (*types.Container, error) {

        var err error

        containerID := uuid.Generate()



        // Create cleanup stack for this container

        cleanupStack := r.cleanupManager.RegisterContainerCleanup(containerID)

        defer cleanupStack.DeferOnError(&err)()



        // Step 1: Validate configuration

        // TODO 1: Check all fields of config for validity (positive memory, valid image ref, etc.)

        // TODO 2: Check if container name is unique

        // TODO 3: Check if image exists locally; if not, return error (caller should pull first)



        // Step 2: Create container metadata

        container := &types.Container{

                ID:   containerID,

                Name: name,

                State: types.StateCreated,

                Config: config,

                CreatedAt: time.Now(),

        }



        // Step 3: Save container to persistent storage (so we know about it even if crash)

        // TODO 4: Call r.containerStore.Save(container)

        cleanupStack.Push(func() {

                // If creation fails, delete the container record

                r.containerStore.Delete(containerID)
```

```
        })

        // Step 4: Create cgroup for container
        // TODO 5: Call r.cgroupManager.CreateCgroup(containerID)
        // TODO 6: Apply resource limits from config.Limits using r.cgroupManager.ApplyLimits
        cleanupStack.Push(func() {
                // Idempotent cgroup cleanup
                r.cgroupManager.RemoveCgroup(containerID)
        })

        // Step 5: Setup network namespace and interfaces
        // TODO 7: Call r.networkManager.SetupNetwork with containerID and config.Network
        cleanupStack.Push(func() {
                // Idempotent network cleanup
                r.networkManager.CleanupNetwork(containerID, config.Network)
        })

        // Step 6: Prepare root filesystem
        // TODO 8: Get image from image store
        // TODO 9: Call r.filesystemManager.SetupRootfs to extract layers and prepare rootfs
        cleanupStack.Push(func() {
                // Idempotent filesystem cleanup
                r.filesystemManager.CleanupFilesystem(containerID)
        })

        // Step 7: All additional preparations (mounts, etc.)
        // TODO 10: Setup additional mounts, /proc, /sys, /dev inside container rootfs

        // Step 8: Container is ready to start
        // TODO 11: Update container state to StateCreated (if not already)

        // Clear cleanup stack since creation succeeded
        // We'll create a new stack for the start operation
        r.cleanupManager.ExecuteContainerCleanup(containerID)

        return container, nil
}
```

E. Language-Specific Hints

1. **Error wrapping in Go**: Use `fmt.Errorf("context: %w", err)` to wrap errors while preserving the original error for inspection with `errors.Is()` and `errors.As()`.
2. **Checking syscall errors**: Use the `os` and `syscall` packages:

```go
if err := syscall.Mount("proc", "/proc", "proc", 0, ""); err != nil {
    if err == syscall.EPERM {
        return fmt.Errorf("need CAP_SYS_ADMIN: %w", err)
    }
    return fmt.Errorf("mount proc failed: %w", err)
}
```

3. **Idempotent cleanup functions**:

```go
func idempotentUnmount(path string) {
    if err := syscall.Unmount(path, syscall.MNT_DETACH); err != nil {
        // Ignore "not mounted" errors
        if err != syscall.EINVAL && err != syscall.ENOENT {
            log.Printf("Warning: unmount %s failed: %v", path, err)
        }
    }
}
```

4. **Context for timeouts**: Use `context.WithTimeout` for network operations:

```go
ctx, cancel := context.WithTimeout(context.Background(), 30*time.Second)
defer cancel()
manifest, err := r.registryClient.PullManifest(ctx, imageRef, tag)
```

5. **Atomic file writes**: To avoid partial writes during image layer download:

```go
// Write to temp file first, then rename atomically
tmpPath := destPath + ".tmp"
defer os.Remove(tmpPath)
// ... download to tmpPath ...
if err := os.Rename(tmpPath, destPath); err != nil {
    return fmt.Errorf("atomic rename failed: %w", err)
}
```

F. Milestone Checkpoint

After implementing error handling, you should be able to test:

1. **Container creation failure recovery**:

```bash
# Run with invalid image reference
sudo ./runtime run --memory 100m nonexistent/image:latest echo "hello"
# Should fail cleanly with error message, no orphaned resources
```

2. **Resource cleanup verification**:

```bash
# Start container, then kill runtime process (Ctrl+Z, kill -9)
sudo ./runtime run --memory 50m alpine:latest sleep 60
# Kill runtime in another terminal
# Restart runtime - it should detect and clean up orphaned container
sudo ./runtime cleanup --all
# Check no cgroups, mounts, or network interfaces remain
```

3. **Edge case testing**:

```bash
# Test immediate exit container
sudo ./runtime run alpine:latest echo "done"
# Container should stop cleanly and be removable


# Test fork bomb with pids limit
sudo ./runtime run --pids-limit 50 alpine:latest /bin/sh -c "forkbomb() { forkbomb | forkbomb & }; forkbomb"
# Should be contained and not affect host
```

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Container starts but immediately exits with code 127	Entrypoint binary not found in rootfs	Check container logs, verify image layers extracted correctly	Ensure image has correct entrypoint, extract layers fully
Mount operation fails with "invalid argument"	OverlayFS not supported or missing kernel module	Check `/proc/filesystems` for overlay entry	Load overlay module: `sudo modprobe overlay`
cgroup creation fails with "permission denied"	cgroup filesystem mounted read-only or wrong permissions	Check `/sys/fs/cgroup` mount options	Remount as rw: `sudo mount -o remount,rw /sys/fs/cgroup`
Network connectivity broken inside container	veth interface not added to bridge or iptables rules missing	Check `ip link`, `brctl show`, `iptables -L -n -t nat`	Ensure network setup completes all steps
Container can see host processes in /proc	/proc not mounted after entering PID namespace	Check mount namespace and order of operations	Mount /proc inside container after `CLONE_NEWPID`
Orphaned resources after runtime crash	Cleanup not executed due to crash	Run runtime with `cleanup --all` flag	Implement startup garbage collection
Image layer download fails with 404	Wrong digest or registry authentication issue	Check manifest for correct layer digests	Ensure authentication tokens are valid

Testing Strategy

> **Milestone(s):** All milestones (testing is essential for validating each component and the integrated system)

This section outlines a comprehensive testing strategy for the container runtime. Given the complexity of interacting with low-level Linux kernel features, testing must be approached at multiple levels—from unit tests of individual components to full system integration tests. The strategy prioritizes **safety** (preventing host system contamination), **reproducibility** (consistent test environments), and **progressive validation** (each milestone builds on tested foundations).

Verification Approaches and Properties

Testing a container runtime presents unique challenges because it directly manipulates kernel primitives that affect system state. We employ a layered testing pyramid with specific verification approaches for each level:

1. Unit Testing: Component Isolation and Mocking

Mental Model: Testing Individual Machine Parts Before Assembly Think of unit testing as testing each component of an engine (fuel pump, spark plugs, pistons) on a workbench before assembling the entire engine. Each component is tested in isolation with simulated inputs and dependencies.

Unit tests focus on the business logic of each component, mocking all external system interactions. This allows rapid iteration and validation of control flow, error handling, and data transformations without requiring root privileges or creating actual kernel resources.

Core Verification Properties for Unit Tests:

- **Data Structure Integrity:** All fields are correctly initialized, serialized, and deserialized
- **State Machine Transitions:** Container state changes follow allowed paths with proper validation
- **Configuration Processing:** User input is validated and merged with defaults correctly
- **Error Propagation:** Errors from lower layers are properly wrapped and returned
- **Resource Management:** Reference counting and cleanup logic work correctly

Mocking Strategy Table:

Component	Mocked Dependencies	Verification Focus
`ContainerStore`	Filesystem I/O	Persistence operations, JSON marshaling/unmarshaling
`CgroupManager`	`/sys/fs/cgroup` files	Limit validation, path construction, controller detection
`NamespaceManager`	`syscall.Clone`, `syscall.Unshare`	Flag composition, error condition simulation
`FilesystemManager`	`syscall.Mount`, `syscall.PivotRoot`	Path validation, mount option construction
`NetworkManager`	`netlink`, `iptables`	IP allocation, bridge setup, rule generation
`ImageStore`	Filesystem, HTTP client	Digest verification, layer extraction logic
`RegistryClient`	HTTP transport	Manifest parsing, authentication flow, retry logic

Example Unit Test Scenarios:

- `ContainerStore.Save()` with invalid container state should return an error
- `CgroupManager.SetLimits()` with negative memory value should validate and reject
- `StateManager.Transition()` from `StateRunning` to `StateCreated` should be blocked
- `CleanupStack.Execute()` should run functions in reverse order and handle panics

2. Integration Testing: Component Interaction with Real Dependencies

Mental Model: Assembling Engine Subsystems on a Test Stand Integration tests assemble related components and test them against real (but isolated) kernel interfaces. This is like testing the fuel system with real gasoline but in a controlled test chamber.

These tests require root privileges and create actual kernel resources but within isolated scopes (temporary directories, test-specific cgroup hierarchies). Each test is responsible for complete cleanup to avoid leaving system state.

Key Integration Testing Patterns:

Test Isolation Table:

Isolation Technique	Implementation	Purpose
Temporary Directory	`os.MkdirTemp()`	Provides scratch space for root filesystems, layer storage
Test-specific Cgroup	Create under `/sys/fs/cgroup/test-*`	Contains cgroup operations to test hierarchy
Private Mount Namespace	`unshare(CLONE_NEWNS)`	Isolate mount operations from host filesystem
Network Namespace Sandbox	`netns.New()` at test start	Test networking without affecting host interfaces
Process Namespace Tracking	Track PIDs created during test	Ensure all child processes are terminated

Integration Test Categories:

1. **Filesystem Integration Tests:** Mount operations, OverlayFS setup, pivot_root behavior
2. **Cgroup Integration Tests:** Limit enforcement, process migration, statistic collection
3. **Namespace Integration Tests:** Process isolation, hostname setting, mount propagation
4. **Network Integration Tests:** veth pair creation, bridge attachment, iptables rules
5. **Image Management Tests:** Layer extraction, manifest parsing, registry communication

Verification Properties for Integration Tests:

- **Kernel Interface Correctness:** System calls behave as expected with given parameters
- **Resource Cleanup:** All created resources (mounts, cgroups, network interfaces) are removed
- **Isolation Integrity:** Operations in test environment don't leak to host
- **Error Recovery:** Partial failures are cleaned up properly (no orphan resources)
- **Concurrent Safety:** Multiple test runs don't interfere with each other

3. System Testing: End-to-End Container Execution

Mental Model: Test Driving the Complete Vehicle System tests run actual containers with realistic configurations and verify the entire stack works together correctly. This is like taking the fully assembled car for a test drive on a closed track.

These tests require full root privileges and create complete container environments. They validate the user-facing CLI commands and the complete container lifecycle.

System Test Architecture:

```
Test Runner (as root)
    ├── Creates temporary runtime root directory
    ├── Initializes all managers with test paths
    ├── Executes container operations via CLI
    └── Verifies container behavior and isolation
```

System Test Scenarios Table:

Test Scenario	Verification Points	Expected Outcome
Simple Container Run	Process starts, runs, exits cleanly	Exit code 0, logs captured
Isolation Verification	Check PID, hostname, filesystem view	Cannot see host processes, has own hostname, limited filesystem
Resource Limit Enforcement	Exceed memory limit, fork bomb with PID limit	OOM kill, fork failure
Network Connectivity	Ping gateway, external IP, other containers	Network works with NAT, inter-container communication
Filesystem Persistence	Write to filesystem, restart container	Data persists in writable layer
Image Pull and Run	Pull from registry, create container	Image layers extracted, container runs with correct entrypoint
Cleanup After Crash	Kill runtime process mid-operation	No orphaned mounts, cgroups, or network interfaces

System Test Properties:

- **End-to-End Correctness:** The complete user workflow works as documented
- **Idempotent Cleanup:** Repeated cleanup operations don't fail
- **State Persistence:** Container state survives runtime restarts
- **Performance Baselines:** Operations complete within expected time bounds

- **Security Boundaries:** Container cannot escape isolation or affect host

4. Property-Based and Fuzz Testing

Mental Model: Stress Testing with Random Inputs Property-based testing generates random but valid inputs to test edge cases that manual test cases might miss. This is like using a vibration table to find weak points in the assembly.

Areas for Property-Based Testing:

- **Configuration Generation:** Random but valid `ContainerConfig` instances
- **Path Manipulation:** Edge cases in filesystem path handling
- **Network Configuration:** Various IP addresses, subnet masks, port combinations
- **Image Layer Combinations:** Different layer counts, sizes, extraction scenarios

Fuzz Testing Targets:

- **Image Manifest Parsers:** Malformed JSON, unexpected fields, large inputs
- **CLI Argument Processing:** Special characters, extremely long strings
- **Environment Variable Handling:** Injection attempts, null bytes, Unicode

5. Negative Testing: Error Paths and Edge Cases

Mental Model: Deliberately Pushing the System to Failure Negative tests verify the system handles errors gracefully rather than crashing or leaving the system in an inconsistent state. This is like testing safety systems by intentionally creating failure conditions.

Critical Negative Test Cases:

1. **Insufficient Privileges:** Run operations as non-root user
2. **Resource Exhaustion:** Fill disk, memory, PID space during operations
3. **Concurrent Modification:** Multiple processes manipulating same container
4. **Partial Failures:** Kill processes mid-operation, disconnect network during pull
5. **Malicious Inputs:** Attempt path traversal, symlink attacks, privilege escalation

Milestone Checkpoints

Each milestone has specific acceptance criteria that must be verified through testing. These checkpoints serve as gates—the implementation should pass all tests for a milestone before proceeding to the next.

Milestone 1: Process Isolation (Namespaces) Checkpoints

Verification Tests:

Unit Tests:

- `NamespaceManager.CreateNamespaces()` correctly composes clone flags from requested namespace types
- State machine prevents starting containers without required namespaces
- Configuration validation rejects invalid namespace combinations

Integration Tests:

1. **PID Namespace Isolation:**

```bash
# Run container with /bin/sh and check PID
$ sudo ./byod run --pid-isolation alpine /bin/sh -c "echo \$\$"
Expected: Outputs "1" (not the host PID)
```

2. **UTS Namespace Hostname:**

```bash
# Set custom hostname in container
$ sudo ./byod run --hostname mycontainer alpine hostname
Expected: Outputs "mycontainer" (host hostname unchanged)
```

3. **Mount Namespace Isolation:**

```bash
# Mount tmpfs inside container, check host doesn't see it

$ sudo ./byod run alpine sh -c "mount -t tmpfs none /tmp && mount | grep tmpfs"

Expected: Shows tmpfs mount inside container

$ mount | grep tmpfs  # On host

Expected: No output (or different tmpfs mounts)
```

System Test Checklist:

- [ ] Process in container cannot see host processes via `ps aux`
- [ ] `unshare` and `nsenter` can attach to container namespaces
- [ ] Container processes are properly reaped when container exits
- [ ] Orphaned namespaces are cleaned up after container removal

Milestone 2: Resource Limits (cgroups) Checkpoints

Verification Tests:

Unit Tests:

- `CgroupManager.DetectCgroupInfo()` correctly identifies cgroup v1 vs v2
- `CgroupManager.SetLimits()` validates and converts units correctly
- Error handling for missing controllers or permission denied

Integration Tests:

1. **Memory Limit Enforcement:**

```bash
# Run container that allocates more than limit

$ sudo ./byod run --memory 10m alpine sh -c "tail /dev/zero"

Expected: Process killed by OOM killer within seconds
```

2. **CPU Limit Throttling:**

```bash
# Run CPU-intensive task, check throttling statistics

$ sudo ./byod run --cpu-shares 512 alpine md5sum /dev/urandom

# Monitor via: cat /sys/fs/cgroup/byod-<id>/cpu.stat

Expected: `nr_throttled` count increases
```

3. **PID Limit Prevention:**

```bash
# Attempt fork bomb with PID limit

$ sudo ./byod run --pids-limit 10 alpine sh -c ":(){ :|:& };:"

Expected: Fork fails after 10 processes, container remains stable
```

System Test Checklist:

- [ ] Cgroup directory created with correct hierarchy
- [ ] Control files contain specified limit values
- [ ] Process moved to cgroup before execution starts
- [ ] Cgroup removed when container is removed
- [ ] Statistics collection works (`GetStats()` returns valid data)

Milestone 3: Filesystem Isolation (chroot/pivot_root) Checkpoints

Verification Tests:

Unit Tests:

- `FilesystemManager.SetupRootfs()` validates rootfs directory structure
- Mount option construction for proc, sys, dev
- Error handling for missing essential directories

Integration Tests:

1. **Root Filesystem Isolation:**

```bash
# Try to access host files from container
$ sudo ./byod run --rootfs /path/to/rootfs alpine ls /host
Expected: No such file or directory (host / not visible)
```

2. **Proc Filesystem Mount:**

```bash
# Check /proc shows container processes only
$ sudo ./byod run alpine sh -c "ls /proc | grep -E '^[0-9]+$' | wc -l"
Expected: Small number (1-2), not host process count
```

3. **Basic Filesystem Operations:**

```bash
# Create, read, delete files in container
$ sudo ./byod run alpine sh -c "echo test > /tmp/file && cat /tmp/file && rm /tmp/file"
Expected: Successfully creates, reads, and deletes file
```

System Test Checklist:

- [ ] Container cannot access host files via path traversal
- [ ] `/proc`, `/sys`, `/dev` mounted with correct options
- [ ] Essential device nodes (`/dev/null`, `/dev/zero`, `/dev/random`) exist
- [ ] Working directory is set correctly per image configuration
- [ ] Old root is properly unmounted after `pivot_root`

Milestone 4: Layered Filesystem (OverlayFS) Checkpoints

Verification Tests:

Unit Tests:

- `OverlayManager.MountOverlay()` validates layer ordering and directories
- `LayerCache` reference counting maintains correct counts
- Error handling for incompatible lower layer combinations

Integration Tests:

1. **Copy-on-Write Behavior:**

```bash
# Modify file from lower layer
$ sudo ./byod run alpine sh -c "echo modified > /etc/hostname && cat /etc/hostname"
Expected: Shows "modified" (change in upper layer)
# Check lower layer unchanged
$ cat /path/to/lower-layer/etc/hostname
Expected: Original content
```

2. **Layer Sharing Between Containers:**

```bash
# Run two containers from same image

$ sudo ./byod run alpine sh -c "touch /shared-test"

$ sudo ./byod run alpine ls -la /shared-test

Expected: File not found in second container (isolated upper layers)
```

3. **Whiteout File Deletion:**

```bash
# Delete file from lower layer

$ sudo ./byod run alpine sh -c "rm /etc/hostname && ls /etc/hostname"

Expected: No such file or directory

# Check whiteout created in upper layer

$ ls -la /path/to/upper-layer/etc/

Expected: Character device with major/minor 0/0 (whiteout)
```

System Test Checklist:

- ☐ Multiple lower layers merged correctly (order preserved)
- ☐ Upper layer captures all writes
- ☐ Work directory used correctly for atomic operations
- ☐ Unmount cleans up all mount points
- ☐ Layer cache prevents duplicate extraction of same layer

Milestone 5: Container Networking Checkpoints

Verification Tests:

Unit Tests:

- `IPAM.Allocate()` prevents duplicate IP allocation
- `NetlinkHelper` command generation for bridge, veth, iptables
- Network configuration validation (CIDR, port ranges)

Integration Tests:

1. **Basic Network Connectivity:**

```bash
# Ping external address

$ sudo ./byod run --network bridge alpine ping -c 1 8.8.8.8

Expected: Successful ping response
```

2. **Inter-Container Communication:**

```bash
# Start two containers on same bridge

$ container1=$(sudo ./byod run --network bridge alpine sleep 60)

$ ip1=$(sudo ./byod inspect $container1 | jq .network.ip_address)

$ sudo ./byod run --network bridge alpine ping -c 1 $ip1

Expected: Successful ping between containers
```

3. **Port Forwarding:**

```bash
# Forward host port to container

$ sudo ./byod run --port 8080:80 alpine nc -l -p 80 &

$ curl http://localhost:8080

Expected: Connection succeeds (or times out waiting for response)
```

System Test Checklist:

- ☐ Network namespace created and isolated
- ☐ veth pair connects container to bridge
- ☐ IP address assigned from configured subnet
- ☐ NAT masquerading allows internet access
- ☐ iptables rules cleaned up on container removal
- ☐ Loopback interface up and working inside container

Milestone 6: Image Format and CLI Checkpoints

Verification Tests:

Unit Tests:

- `RegistryClient` handles authentication errors and retries
- OCI manifest parsing with various schema versions
- CLI command flag parsing and validation

Integration Tests:

1. **Image Pull and Run:**

```bash
# Pull image and run it

$ sudo ./byod pull alpine:latest

Expected: Downloads layers, verifies digests, stores locally

$ sudo ./byod run alpine:latest echo "Hello from container"

Expected: Runs container with correct entrypoint, outputs message
```

2. **Container Lifecycle Management:**

```bash
# Full lifecycle test

$ id=$(sudo ./byod create alpine:latest)

$ sudo ./byod start $id

$ sudo ./byod stop $id

$ sudo ./byod remove $id

Expected: Each command succeeds, state transitions correctly
```

3. **Configuration Merging:**

```bash
# Override image entrypoint with CLI command

$ sudo ./byod run alpine:latest /bin/ls -la /

Expected: Runs ls instead of default shell, shows root directory
```

System Test Checklist:

- ☐ Image layers stored in content-addressable storage
- ☐ Manifest and config parsed correctly
- ☐ Container state persisted across runtime restarts

- [ ] CLI commands produce expected output format
- [ ] Error messages are user-friendly and actionable
- [ ] All resources cleaned up after container removal

Cross-Cutting Testing Concerns

Performance Testing:

- Container startup time should be < 100ms for small images
- Memory overhead per container should be < 5MB
- Concurrent container creation should scale linearly up to resource limits

Security Testing:

- No privilege escalation opportunities
- No information leakage between containers
- Resource exhaustion doesn't affect host stability
- All user input properly sanitized and validated

Compatibility Testing:

- Works with multiple base images (Alpine, Ubuntu, BusyBox)
- Compatible with different Linux distributions (Ubuntu, Fedora, CentOS)
- Kernel version compatibility (4.4+ for basic features, 5.4+ for cgroup v2)

Test Environment Requirements

Minimum Test Environment Setup:

- Linux kernel 4.4 or higher (5.10+ recommended for cgroup v2)
- Root privileges (via sudo or root user)
- 1GB free disk space for test images
- Network connectivity for pulling images
- Go 1.18+ for building and testing

Test Isolation Recommendations:

- Use separate temporary directory for each test run
- Run tests in disposable VMs or containers for complete isolation
- Implement test tags to separate unit, integration, and system tests
- Cleanup routines that run even on test failure (defer pattern)

Implementation Guidance

This section provides practical guidance for implementing the testing strategy in Go, with a focus on making tests reliable, isolated, and maintainable.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Unit Testing	Standard Go `testing` package with `testify/mock`	Custom mock generators, property-based testing with `gopter`
Integration Testing	Root-requiring tests with `skip` unless flag set	Test containers that run tests in isolated environments
System Testing	Bash scripts calling CLI	Go tests that exec CLI and verify output
Network Testing	Loopback interface only, skip real network tests	`github.com/vishvananda/netns` for namespace testing
Filesystem Testing	Temp directories with cleanup	Mount namespace isolation for safe mount operations
Cgroup Testing	Skip unless in CI with specific permissions	Detect available controllers and test only supported ones

B. Recommended File/Module Structure

```
project-root/
├── cmd/
│   └── byod/                   # CLI entry point
│       └── main.go
├── internal/
│   ├── runtime/                # Core runtime logic
│   │   ├── runtime.go
│   │   ├── runtime_test.go     # Unit tests
│   │   └── runtime_integration_test.go  # Integration tests
│   ├── container/              # Container management
│   │   ├── store.go
│   │   ├── state.go
│   │   └── store_test.go
│   ├── cgroups/                # cgroup management
│   │   ├── manager.go
│   │   ├── manager_integration_test.go
│   │   └── mock_manager.go     # Mock for unit tests
│   ├── namespaces/             # Namespace management
│   │   ├── manager.go
│   │   └── manager_test.go
│   ├── filesystem/             # Filesystem isolation
│   │   ├── manager.go
│   │   ├── overlay.go
│   │   └── manager_integration_test.go
│   ├── network/                # Network management
│   │   ├── manager.go
│   │   ├── iptables.go
│   │   └── manager_integration_test.go
│   ├── image/                  # Image handling
│   │   ├── store.go
│   │   ├── registry.go
│   │   └── store_test.go
│   └── cli/                    # CLI command implementations
│       ├── commands.go
│       └── commands_test.go
├── tests/                      # System and end-to-end tests
│   ├── system/                 # Full system tests
│   │   ├── container_lifecycle_test.go
│   │   └── network_test.go
│   └── fixtures/               # Test fixtures (small rootfs, etc.)
│       └── busybox-rootfs.tar
├── pkg/                        # Public APIs (if any)
│   └── types/                  # Shared types
│       ├── container.go
│       └── image.go
└── scripts/                    # Test support scripts
    ├── setup-test-env.sh
    └── cleanup-test-resources.sh
```

C. Infrastructure Starter Code

Test Helper for Root-Requiring Tests:

```go
// internal/testutil/root_required.go

package testutil

import (
    "os"
    "testing"
)

// SkipIfNotRoot skips the test if not running as root.
// Use: defer testutil.SkipIfNotRoot(t)()
func SkipIfNotRoot(t *testing.T) func() {
    if os.Geteuid() != 0 {
        t.Skip("Test requires root privileges")
    }
    return func() {} // No-op cleanup
}

// TempCgroupPath creates a temporary directory for cgroup testing.
// Automatically cleans up after test.
func TempCgroupPath(t *testing.T) string {
    // Implementation creates test-specific cgroup path
}

// WithTestNamespace runs test code in a new mount namespace.
func WithTestNamespace(t *testing.T, fn func()) {
    // Implementation uses unshare to isolate mounts
}
```

Mock Cgroup Manager for Unit Tests:

```go
// internal/cgroups/mock_manager.go

package cgroups

import (
    "github.com/stretchr/testify/mock"
)

type MockCgroupManager struct {
    mock.Mock
}

func (m *MockCgroupManager) CreateCgroup(containerID string) error {
    args := m.Called(containerID)
    return args.Error(0)
}

func (m *MockCgroupManager) SetLimits(containerID string, limits ResourceLimits) error {
    args := m.Called(containerID, limits)
    return args.Error(0)
}

// ... implement all methods with mock.Called
```

Test Fixture Setup Helper:

```go
// tests/fixtures/setup.go

package fixtures

import (
    "archive/tar"
    "compress/gzip"
    "io"
    "os"
    "path/filepath"
)

// CreateTestRootFS creates a minimal rootfs for testing.
func CreateTestRootFS(dir string) error {
    // Create essential directories
    dirs := []string{"bin", "dev", "etc", "proc", "sys", "tmp", "usr/bin"}
    for _, d := range dirs {
        if err := os.MkdirAll(filepath.Join(dir, d), 0755); err != nil {
            return err
        }
    }

    // Create minimal /etc/passwd
    passwdContent := "root:x:0:0:root:/root:/bin/sh\n"
    if err := os.WriteFile(filepath.Join(dir, "etc/passwd"),
        []byte(passwdContent), 0644); err != nil {
        return err
    }

    // Create /bin/sh as symlink (or copy busybox if available)
    shPath := filepath.Join(dir, "bin/sh")
    if _, err := os.Stat("/bin/sh"); err == nil {
        // Copy host's sh (for testing only)
        data, err := os.ReadFile("/bin/sh")
        if err == nil {
            os.WriteFile(shPath, data, 0755)
        }
    } else {
        // Create dummy executable
        os.WriteFile(shPath, []byte("#!/bin/sh\nexit 0"), 0755)
```

```
        os.Chmod(shPath, 0755)
    }

    return nil
}
```

D. Core Logic Skeleton Code for Tests

Integration Test for PID Namespace Isolation:

```go
// internal/namespaces/manager_integration_test.go
package namespaces

import (
    "os"
    "testing"

    "github.com/yourproject/internal/testutil"
)

func TestPIDNamespaceIsolation(t *testing.T) {
    defer testutil.SkipIfNotRoot(t)()

    // TODO 1: Create a temporary directory for test artifacts
    // TODO 2: Initialize NamespaceManager with test configuration
    // TODO 3: Use RunInNamespaces with CLONE_NEWPID flag
    // TODO 4: In child function, check that PID is 1 via os.Getpid()
    // TODO 5: In parent, verify child process had correct namespace
    // TODO 6: Clean up all test resources (process, namespaces)
    // TODO 7: Verify no orphaned processes or namespaces remain
}
```

System Test for Container Lifecycle:

```go
// tests/system/container_lifecycle_test.go

package system


import (

    "encoding/json"

    "os/exec"

    "testing"


    "github.com/yourproject/internal/testutil"

)


func TestContainerCreateStartStopRemove(t *testing.T) {

    defer testutil.SkipIfNotRoot(t)()


    // TODO 1: Build the CLI binary if not already built

    // TODO 2: Create a test container with minimal image

    // TODO 3: Verify container appears in list with 'created' state

    // TODO 4: Start the container and verify state changes to 'running'

    // TODO 5: Check container process is running with correct namespaces

    // TODO 6: Stop the container and verify state changes to 'stopped'

    // TODO 7: Remove the container and verify it's gone from list

    // TODO 8: Verify all resources cleaned up (cgroups, mounts, network)

}
```

Property-Based Test for Configuration Validation:

```go
// internal/container/config_test.go

package container


import (

    "testing"



    "github.com/leanovate/gopter"

    "github.com/leanovate/gopter/gen"

    "github.com/leanovate/gopter/prop"

)


func TestContainerConfigValidation(t *testing.T) {

    parameters := gopter.DefaultTestParameters()

    parameters.MinSuccessfulTests = 100

    properties := gopter.NewProperties(parameters)



    // TODO 1: Define generator for valid ContainerConfig

    configGen := gen.Struct(reflect.TypeOf(ContainerConfig{}), map[string]gopter.Gen{

        "MemoryMB":  gen.IntRange(1, 1024),  // 1MB to 1GB

        "CPUShares": gen.IntRange(1, 1024),

        "PidsLimit": gen.IntRange(1, 100),

    })



    // TODO 2: Test property: valid config should pass validation

    properties.Property("Valid config passes validation", prop.ForAll(

        func(config ContainerConfig) bool {

            err := ValidateConfig(config)

            return err == nil

        },

        configGen,

    ))



    // TODO 3: Test property: config with negative memory should fail

    properties.Property("Negative memory fails validation", prop.ForAll(

        func(memory int) bool {

            config := ContainerConfig{MemoryMB: memory}

            err := ValidateConfig(config)

            return (memory <= 0 && err != nil) || (memory > 0 && err == nil)

        },
```

```
        gen.IntRange(-100, 100),
    ))


    properties.TestingRun(t)
}
```

E. Language-Specific Hints for Go Testing

1. **Use `t.Cleanup()` for test resource cleanup:**

```go
func TestWithCleanup(t *testing.T) {
    dir := t.TempDir()  // Automatically cleaned up
    // Or manually:
    f, err := os.CreateTemp("", "test-")
    if err != nil { t.Fatal(err) }
    t.Cleanup(func() { os.Remove(f.Name()) })
}
```

2. **Parallelize safe tests with `t.Parallel()`:**

```go
func TestParallel(t *testing.T) {
    t.Parallel()  // Runs concurrently with other parallel tests
    // Test code that doesn't share global state
}
```

3. **Use table-driven tests for multiple test cases:**

```go
func TestNamespaceFlags(t *testing.T) {
    tests := []struct{
        name string
        input []string
        want int
    }{
        {"PID only", []string{"pid"}, syscall.CLONE_NEWPID},
        {"PID and NET", []string{"pid", "net"}, syscall.CLONE_NEWPID | syscall.CLONE_NEWNET},
    }
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            got := flagsFromNames(tt.input)
            if got != tt.want {
                t.Errorf("got %v, want %v", got, tt.want)
            }
        })
    }
}
```

4. **Test CLI commands with `exec.Command`:**

```go
func TestCLICommand(t *testing.T) {
    cmd := exec.Command("./byod", "run", "alpine", "echo", "test")
    output, err := cmd.CombinedOutput()
    if err != nil { t.Fatalf("command failed: %v\n%s", err, output) }
    if !strings.Contains(string(output), "test") {
        t.Errorf("unexpected output: %s", output)
    }
}
```

F. Milestone Checkpoint Validation Commands

After completing each milestone, run these validation commands:

Milestone 1 Checkpoint:

```bash
# Run namespace isolation tests
$ sudo go test -v ./internal/namespaces/... -run TestPIDNamespace
# Expected: All tests pass, showing PID isolation works

# Manual verification
$ sudo ./byod run --pid-isolation --uts-isolation test/fixtures/minimal-rootfs /bin/sh -c "echo \$\$ && hostname"
# Expected: Outputs "1" and a hostname (not the host's)
```

Milestone 2 Checkpoint:

```bash
# Run cgroup integration tests
$ sudo go test -v ./internal/cgroups/... -run TestMemoryLimit
# Expected: Tests pass, showing OOM kill triggers

# Manual verification of memory limit
$ sudo ./byod run --memory 10m test/fixtures/minimal-rootfs /bin/sh -c "tail /dev/zero" &
$ sleep 2
$ dmesg | tail -5 | grep "Memory cgroup out of memory"
# Expected: OOM killer message appears in dmesg
```

Milestone 3 Checkpoint:

```bash
# Run filesystem isolation tests
$ sudo go test -v ./internal/filesystem/... -run TestPivotRoot
# Expected: Tests pass, showing filesystem isolation

# Manual verification
$ sudo ./byod run --rootfs test/fixtures/busybox-rootfs /bin/ls /
# Expected: Shows container root directory, not host root
```

Milestone 4 Checkpoint:

```bash
# Run OverlayFS tests
$ sudo go test -v ./internal/filesystem/... -run TestOverlayMount
# Expected: Tests pass, showing copy-on-write behavior

# Manual verification
$ sudo ./byod run alpine:latest /bin/sh -c "echo test > /modified && ls /"
# Expected: File created successfully, visible in container
```

Milestone 5 Checkpoint:

```bash
# Run network tests
$ sudo go test -v ./internal/network/... -run TestBridgeNetwork
# Expected: Tests pass, showing network connectivity

# Manual verification
$ sudo ./byod run --network bridge alpine:latest ping -c 1 8.8.8.8
# Expected: Ping successful (requires internet connection)
```

Milestone 6 Checkpoint:

```bash
# Run full system tests
$ sudo go test -v ./tests/system/... -run TestContainerLifecycle
# Expected: All tests pass, showing complete workflow

# End-to-end validation
$ sudo ./byod pull alpine:latest
$ sudo ./byod run alpine:latest echo "Hello, world!"
# Expected: Image downloads, container runs, outputs message
```

G. Debugging Tips for Tests

Symptom	Likely Cause	How to Diagnose	Fix
Test passes locally but fails in CI	Different kernel version, missing kernel features	Check kernel version, available cgroup controllers	Skip test if feature unavailable, document requirements
Tests leave orphaned resources	Cleanup not running on test failure	Use `t.Cleanup()`, check `/proc/mounts`, `/sys/fs/cgroup` after test	Ensure cleanup runs even on panic, use defer with named return values
Intermittent test failures	Race conditions, timing issues	Run with `-race` flag, add retries for eventual consistency	Use proper synchronization, increase timeouts for slow operations
Permission denied even as root	AppArmor/SELinux restrictions, namespace permissions	Check audit logs (`dmesg`, `journalctl`), test in permissive mode	Adjust security policies, use correct flags for namespace operations
Network tests fail	Firewall rules, network namespace leaks	List iptables rules, check network namespace count in `/proc`	Ensure proper cleanup, disable firewall temporarily for tests
Mount tests fail	Mount propagation, shared subtrees	Check `/proc/self/mountinfo` for propagation flags	Use `MS_PRIVATE` or `MS_SLAVE` for test mounts
Cgroup tests fail on cgroup v2	Different API, controller availability	Check cgroup version at `/sys/fs/cgroup/cgroup.controllers`	Implement both v1 and v2 support, detect and adapt
Image pull tests timeout	Network issues, registry rate limiting	Check HTTP logs, use smaller test images	Mock registry for unit tests, use local image fixtures

Debugging Guide

> **Milestone(s):** All milestones (debugging is essential throughout all implementation stages)

Debugging a container runtime presents unique challenges because we're interacting with multiple Linux kernel subsystems that create complex, layered isolation boundaries. When something goes wrong, the symptoms can manifest in unexpected ways—processes failing to start, resource limits not being enforced, network connectivity issues, or mysterious permission errors. This guide provides a systematic approach to diagnosing and fixing common problems you'll encounter while building your container runtime. Think of debugging a container runtime as being a building superintendent who can't enter the apartments directly—you have to use specialized tools to inspect what's happening inside each isolated unit without breaking the isolation itself.

Symptom → Cause → Fix Tables

These tables map observable symptoms to their root causes and provide concrete fixes. They're organized by the milestone areas where problems typically occur.

Namespace Isolation Issues

Symptom	Likely Cause	How to Diagnose	Fix
Process inside container can still see host processes (ps, top show host PIDs)	PID namespace not properly set up or `/proc` not mounted in container's mount namespace	Run `ls -la /proc/self/ns/pid` inside container to check namespace ID; compare with host's PID namespace. Check if `/proc` is mounted with `mount | grep proc`.	Ensure `CLONE_NEWPID` flag is passed to `clone()` or `unshare()`. Use `pivot_root()` followed by mounting `/proc` inside the container with `mount("proc", "/proc", "proc", 0, "")`.
Container cannot set its own hostname (hostname command fails or shows host's hostname)	Missing `CLONE_NEWUTS` flag or UTS namespace not created	Check `/proc/self/ns/uts` inside container vs host. Try running `hostname newname` —if it succeeds but isn't isolated, UTS namespace is missing.	Add `CLONE_NEWUTS` to namespace creation flags. Verify the `sethostname()` syscall is called after namespace creation but before `execve()`.
Mount/unmount operations in container affect host filesystem	Missing `CLONE_NEWNS` (mount namespace) flag	Create a temporary directory, mount a tmpfs inside container, then check if it appears on host with `mount | grep tmpfs`.	Ensure `CLONE_NEWNS` is included. Note: In Go, this is `syscall.CLONE_NEWNS` (all mount namespace operations use this single flag).
Container cannot create shared memory segments	IPC namespace not isolated (`CLONE_NEWIPC` missing)	Try creating a shared memory segment with `shmget()` inside container and check if host processes can see it with `ipcs -m`.	Add `CLONE_NEWIPC` flag. Also ensure semaphore and message queue isolation if those are needed.
User ID mapping issues (permission denied on files that should be accessible)	User namespace not set up or `/etc/passwd` not mapped correctly	Check `/proc/self/uid_map` and `/proc/self/gid_map` inside container. Verify files have correct ownership.	Implement user namespace with `CLONE_NEWUSER`, write appropriate mappings to `/proc/self/uid_map` and `/proc/self/gid_map` before executing container process.
Container process exits immediately with "operation not permitted"	Insufficient capabilities when using namespaces	Check `dmesg | tail` for kernel capability warnings. Try running container runtime as root.	Grant necessary capabilities (CAP_SYS_ADMIN for mount operations, CAP_NET_ADMIN for network setup) or run as root during development.

cgroup Resource Limit Issues

Symptom	Likely Cause	How to Diagnose	Fix
Memory limit not enforced (container uses more memory than specified)	Memory cgroup controller not enabled or limits written to wrong file	Check `/sys/fs/cgroup/memory/memory.limit_in_bytes` (v1) or `/sys/fs/cgroup/container-<id>/memory.max` (v2). Verify controller is available: `cat /proc/cgroups | grep memory`.	Ensure memory controller is mounted and available. Write limit in bytes to correct control file. For v2, also check `memory.swap.max`.
Container not OOM-killed when exceeding memory limit	Memory limit set but OOM killer disabled or swap allowed	Check `memory.oom_control` (v1) or `memory.oom.group` (v2). Verify swap limit: `memory.memsw.limit_in_bytes` (v1) or `memory.swap.max` (v2).	Disable swap by setting swap limit equal to memory limit. Enable OOM killer: `echo 1 > memory.oom_control` (v1) or set `memory.oom.group` (v2).
CPU limit not respected (container uses 100% CPU despite shares/quota)	CPU controller not enabled or wrong control files used	Check CPU usage via `top` or `cat /sys/fs/cgroup/cpu,cpuacct/cpuacct.usage`. Verify `cpu.shares` (v1) or `cpu.weight` (v2) values.	For v1, set `cpu.cfs_period_us` and `cpu.cfs_quota_us`. For v2, set `cpu.max` with format `max 100000`.
PIDs limit not working (container can fork unlimited processes)	`pids` controller not enabled or limit not set	Check `pids.current` vs `pids.max` in cgroup directory. Verify controller is in `/proc/cgroups`.	Enable `pids` controller. Write limit to `pids.max` (both v1 and v2 use same interface).
cgroup cleanup fails ("device or resource busy" error)	Processes still running in cgroup or subcgroups exist	Check `cgroup.procs` and `tasks` files for remaining PIDs. Look for nested cgroups: `ls /sys/fs/cgroup/memory/<container-id>/`.	Kill all processes in cgroup before removal. Use `cgdelete` or recursively remove subdirectories.
cgroup v2 vs v1 detection fails	Hybrid or mixed hierarchy on system	Check `/proc/filesystems` for `cgroup2`. Examine mount points: `mount | grep cgroup`.	Use `CgroupManager.DetectCgroupInfo()` to detect version and available controllers. Support both versions with fallback.

Filesystem Isolation Issues

Symptom	Likely Cause	How to Diagnose	Fix
Container cannot see /proc or /sys (ps, top fail)	`/proc` and `/sys` not mounted after `pivot_root()`	Run `mount | grep -E "proc|sys"` inside container. Check if directories exist in rootfs.	Call `MountProc()` and `MountSys()` after `pivot_root()` but before `execve()`.
Container can still access host filesystem	`pivot_root()` succeeded but old root not unmounted	Check `mount | grep /oldroot` or try accessing `/oldroot/etc/passwd` from container.	Unmount old root with `MNT_DETACH`: `unmount("/oldroot", MNT_DETACH)`.
Permission denied on device nodes (`/dev/null`, `/dev/zero`)	Device nodes not created in container's `/dev`	Try `ls -la /dev/null` inside container. Check if major/minor numbers are correct.	Call `SetupDev()` to create basic device nodes with `mknod()` using appropriate major/minor numbers.
Container startup fails with "no such file or directory" on binary	Rootfs missing shared libraries or binary not in `PATH`	Use `ldd` on binary inside rootfs to check missing libraries. Verify `PATH` environment variable.	Use complete rootfs from base image (alpine, ubuntu). Set correct `PATH` in `ContainerConfig.Env`.
Files created in container disappear after exit	Using `chroot()` without mount namespace or wrong upperdir in OverlayFS	Check if files exist in upperdir of OverlayFS after container stops. Verify mount propagation.	Use `pivot_root()` with mount namespace. For OverlayFS, ensure writes go to upperdir and it's preserved.
Mount propagation doesn't work (mounts in container not visible in sub-mounts)	Mount namespace setup with wrong propagation flags	Check `/proc/self/mountinfo` for propagation flags (`shared`, `private`, `slave`).	Set root mount as `MS_PRIVATE` before `pivot_root()`: `mount("", "/", "", MS_REC|MS_PRIVATE, "")`.

OverlayFS Issues

Symptom	Likely Cause	How to Diagnose	Fix
"Invalid argument" when mounting OverlayFS	Missing lowerdir, upperdir, or workdir directories	Check `dmesg | tail` for kernel OverlayFS errors. Verify all directories exist with correct permissions.	Create lowerdir, upperdir, workdir directories before mount. Ensure workdir is empty.
File modifications not persisted (writes disappear)	Upperdir not writable or OverlayFS mounted read-only	Check mount options: `mount | grep overlay`. Verify permissions on upperdir: `ls -ld upperdir`.	Ensure upperdir has write permissions. Include `lowerdir=...,upperdir=...,workdir=...` in mount options.
Whiteout files not working (deleted files reappear)	Whiteout special files not created properly	Check upperdir for files with `chardev 0:0` (whiteout). Use `ls -la upperdir` and look for character devices.	When deleting files in container, create whiteout device in upperdir: `mknod("whiteout", S_IFCHR, 0)`.
Copy-on-write causes high CPU/I/O	Too many small files triggering copy-up	Monitor `cp` operations during container run. Check if upperdir grows unexpectedly.	This is expected behavior; consider using `nocopy` mount option for read-only lower layers where appropriate.
OverlayFS mount fails on certain filesystems (e.g., NFS)	Underlying filesystem not supporting OverlayFS features	Check kernel logs: `dmesg | grep overlay`. Test with ext4 or xfs as backing filesystem.	Use supported filesystem (ext4, xfs, btrfs) for storage directory.
Multiple containers sharing layers cause conflicts	Same upperdir used by multiple containers	Check if upperdir paths are unique per container. Verify containers don't share workdir.	Generate unique upperdir and workdir per container using container ID in path.

Container Networking Issues

Symptom	Likely Cause	How to Diagnose	Fix
Container has no network connectivity (ping fails)	veth pair not set up correctly or no default route	Check interfaces inside container: `ip link show`. Verify default route: `ip route show`.	Ensure veth pair created, moved to container namespace, and configured with IP. Add default route to gateway.
Container cannot reach external internet	NAT masquerading not configured	Check iptables rules: `iptables -t nat -L -n`. Verify `FORWARD` chain policy is `ACCEPT`.	Call `IPTablesManager.SetupMasquerade()` to add `POSTROUTING` masquerade rule. Enable IP forwarding: `echo 1 > /proc/sys/net/ipv4/ip_forward`.
Port forwarding not working (can't connect to host port)	iptables rules missing or wrong interface	Check DNAT rules: `iptables -t nat -L PREROUTING -n`. Verify traffic reaches bridge interface.	Add rule with `IPTablesManager.AddPortForwarding()` that matches protocol, host port, and redirects to container IP:port.
DNS resolution fails inside container	`/etc/resolv.conf` not configured	Try `nslookup google.com` inside container. Check `/etc/resolv.conf` contents.	Copy host's `/etc/resolv.conf` or create one with nameserver `8.8.8.8` in container rootfs before start.
Containers on same bridge cannot ping each other	Bridge not set up or firewall blocking	Check bridge membership: `brctl show`. Verify containers have IPs in same subnet.	Ensure both veth ends are connected to same bridge. Disable firewall or add accept rules for bridge traffic.
Network namespace cleanup fails (veth left on host)	veth not deleted when container stops	Check leftover veth interfaces: `ip link show | grep veth`.	Delete veth pair in host namespace during `NetworkManager.CleanupNetwork()`.
"Network is unreachable" for loopback	Loopback interface not up in container	Check `ip link show lo` inside container—should be `UP`.	Bring loopback up: `ip link set lo up` inside network namespace.

Image and CLI Issues

Symptom	Likely Cause	How to Diagnose	Fix
Image pull fails with "manifest not found"	Wrong image reference format or tag missing	Check image ref parsing. Verify tag exists on registry: `curl -L https://registry.hub.docker.com/v2/library/alpine/tags/list`.	Use standard format: `image:tag` (default to `latest`). Handle Docker Hub vs other registries.
Layer extraction fails with "invalid tar header"	Corrupted layer download or wrong compression	Verify layer digest matches. Check file type: `file layer.tar`.	Use `ImageStore.StoreBlob()` which verifies digest. Support multiple compression formats (gzip, zstd).
Container starts but immediately exits	Entrypoint command not found or fails	Check container logs (stdout/stderr). Verify `ImageConfig.Entrypoint` and `Cmd` are valid.	Provide fallback to `/bin/sh` if no command specified. Log exit code and signal.
State file corruption after crash	Concurrent writes to container state or partial write	Check `ContainerStore.Save()` for atomic write pattern. Verify file permissions.	Use atomic file write: write to temp file then rename. Add file locking with `sync.Mutex`.
CLI command hangs indefinitely	Container process waiting for input or deadlock	Check if process is running with `ps aux | grep <container-id>`. Look for open file descriptors.	Implement timeout for `StartCommand()`. Ensure stdin is properly handled (close or connect to terminal).
"Container already exists" error	Container ID collision or stale state file	Check if container ID already in `StateManager.containers` map. Look for leftover directories.	Use UUID with `uuid.Generate()` for uniqueness. Clean up stale state on startup with orphan detection.

Domain-Specific Debugging Techniques

Debugging container runtimes requires specialized techniques that account for the layered isolation. These methods allow you to peer inside containers without breaking their isolation boundaries.

The "Inception" Debugging Method

> **Mental Model:** Think of debugging containers like the movie Inception—you need to enter nested dream levels (namespaces) to see what's really happening. But unlike the movie, you can use special kernel APIs to project yourself into these isolated spaces without actually entering them.

The key insight is that you can attach to namespaces from outside using `/proc` filesystem entries. Every process has a symbolic link for each namespace type at `/proc/<pid>/ns/<type>`. By opening these files, you can perform operations in the context of that namespace.

Technique 1: Inspecting Namespace Membership

```bash
# From host, find container process PID
CONTAINER_PID=$(pgrep -f "container-id")

# Check which namespaces it's in
ls -la /proc/$CONTAINER_PID/ns/

# Compare namespace IDs with host
readlink /proc/$CONTAINER_PID/ns/pid
readlink /proc/self/ns/pid
```

If the namespace IDs match, isolation isn't working. Each namespace should have a unique ID.

Technique 2: Entering Network Namespace for Debugging

```bash
# Use nsenter to run commands in container's network namespace

nsenter --net=/proc/$CONTAINER_PID/ns/net ip addr show

nsenter --net=/proc/$CONTAINER_PID/ns/net ping -c 3 8.8.8.8
```

This lets you see exactly what network interfaces and routes the container sees.

Technique 3: Mount Namespace Inspection

```bash
# View container's mount table from outside

nsenter --mount=/proc/$CONTAINER_PID/ns/mnt mount

cat /proc/$CONTAINER_PID/mountinfo | grep -v overlay
```

The `mountinfo` file shows the complete mount tree with propagation flags.

The "Resource Accounting" Audit

When cgroups aren't working properly, you need to audit the entire resource control hierarchy.

Technique 4: cgroup Hierarchy Walk

```bash
# For cgroup v2

find /sys/fs/cgroup -name "container-*" -type d | xargs -I {} sh -c 'echo {}; cat {}/cgroup.procs'


# Check specific controller limits

cat /sys/fs/cgroup/container-abc123/memory.max

cat /sys/fs/cgroup/container-abc123/cpu.max
```

Technique 5: Real-time Resource Monitoring

```bash
# Watch memory usage of container

watch -n 1 'cat /sys/fs/cgroup/container-abc123/memory.current'


# Monitor CPU usage via cpuacct

echo "CPU usage (nanoseconds): $(cat /sys/fs/cgroup/container-abc123/cpu.stat | grep usage_usec)"
```

The "Filesystem Forensics" Approach

When filesystem isolation fails, you need to understand the exact mount hierarchy and where writes are actually going.

Technique 6: OverlayFS Layer Inspection

```bash
# Find container's merged directory

find /var/lib/container-runtime -name "merged" -type d | grep container-id


# Check what's in each layer

ls -la /var/lib/container-runtime/layers/sha256:abc123/

ls -la /var/lib/container-runtime/containers/container-id/upper/


# See copy-on-write in action

strace -e openat,rename,unlink,write <container-process-pid> 2>&1 | grep -i overlay
```

Technique 7: Bind Mount Debugging

```bash
# Create a debug bind mount from host into container
mkdir -p /tmp/container-debug
mount --bind /proc/$CONTAINER_PID/root /tmp/container-debug

# Now inspect container rootfs from host
ls -la /tmp/container-debug/
cat /tmp/container-debug/etc/os-release

# Clean up
umount /tmp/container-debug
```

The "Network Packet Surgery"

For networking issues, you need to trace packets through the entire network stack—from container interface to host bridge to external network.

Technique 8: Packet Tracing with tcpdump

```bash
# Capture on container veth interface
tcpdump -i vethabc123 -n -v

# Capture on bridge interface
tcpdump -i br0 -n -v

# Capture in container namespace
nsenter --net=/proc/$CONTAINER_PID/ns/net tcpdump -i eth0 -n -v
```

Technique 9: iptables Rule Tracing

```bash
# Add trace rule for specific traffic
iptables -t raw -A PREROUTING -p tcp --dport 80 -j TRACE

# Watch kernel logs for trace output
dmesg -w | grep TRACE

# Check packet flow through chains
iptables -t nat -L -n -v
iptables -t filter -L -n -v
```

The "Process Relationship Mapping"

Understanding parent-child relationships and process trees across namespaces is critical for PID namespace issues.

Technique 10: Cross-Namespace Process Tree

```bash
# Show process tree with namespace contexts
ps ax -o pid,ppid,pgid,sid,tty,time,comm,ns --forest

# Find all processes in specific PID namespace
for pid in $(ls /proc/ | grep '^[0-9]'); do
  if [ -e /proc/$pid/ns/pid ]; then
    ns=$(readlink /proc/$pid/ns/pid)
    if [ "$ns" = "$CONTAINER_NS" ]; then
      echo "PID $pid is in container namespace"
      cat /proc/$pid/cmdline
    fi
  fi
done
```

Tools for Inspecting System State

The Linux ecosystem provides powerful tools for inspecting the kernel state that underpins container isolation. Mastering these tools is essential for effective debugging.

Namespace Inspection Tools

Tool	Purpose	Example Command	What It Reveals
`lsns`	List all namespaces on system	`lsns -t pid`	Shows PID namespaces with process counts and PIDs
`nsenter`	Enter one or more namespaces	`nsenter -t $PID -m -u -n -p /bin/bash`	Spawns shell in container's combined namespaces
`unshare`	Run program in new namespaces	`unshare --pid --fork --mount-proc=/proc /bin/bash`	Tests namespace creation independently of runtime
`/proc/<pid>/ns/`	Namespace file descriptors	`readlink /proc/self/ns/mnt`	Gets namespace ID for comparison
`/proc/<pid>/status`	Namespace membership	`grep NSpid /proc/$PID/status`	Shows PID in each ancestor PID namespace

cgroup Inspection Tools

Tool	Purpose	Example Command	What It Reveals
`systemd-cgls`	Show cgroup hierarchy	`systemd-cgls -u`	Visual tree of cgroups and their processes
`systemd-cgtop`	Resource usage per cgroup	`systemd-cgtop`	Real-time CPU/memory usage by cgroup
`cgget`	Get cgroup parameters	`cgget -g memory:container-id`	Reads control file values for specific cgroup
`cat /proc/cgroups`	List available controllers	`cat /proc/cgroups`	Shows which controllers are enabled and hierarchy count
`findmnt cgroup`	Find cgroup mount points	`findmnt -t cgroup2`	Shows where cgroup filesystems are mounted

Filesystem Inspection Tools

Tool	Purpose	Example Command	What It Reveals
`findmnt`	Show mount tree	`findmnt -R /var/lib/container-runtime`	Complete mount hierarchy with options and propagation
`mount`	List mounted filesystems	`mount | grep overlay`	All active mounts with their types and options
`lsof`	List open files	`lsof +D /var/lib/container-runtime/containers/`	Which processes have files open in container directories
`strace`	Trace system calls	`strace -f -e mount,pivot_root,chdir <pid>`	Exact sequence of filesystem operations performed
`inotifywait`	Monitor filesystem events	`inotifywait -rm /var/lib/container-runtime/`	Real-time notifications of file creation, deletion, modification

Network Inspection Tools

Tool	Purpose	Example Command	What It Reveals
`ip netns`	Network namespace management	`ip netns list`	All network namespaces and their associated PIDs
`ip link`	Network interface listing	`ip -o link show | grep veth`	All veth interfaces and their peer indices
`bridge`	Bridge management	`bridge link show`	Which interfaces are connected to bridges
`conntrack`	Connection tracking	`conntrack -L | grep container-ip`	NAT translations and connection states
`ss`	Socket statistics	`ss -tulpn | grep $CONTAINER_PID`	Open ports and sockets in container

Process and Container-Specific Tools

Tool	Purpose	Example Command	What It Reveals
`pstree`	Process tree visualization	`pstree -p $CONTAINER_PID`	Parent-child relationships across namespaces
`cat /proc/<pid>/cgroup`	Process cgroup membership	`cat /proc/$PID/cgroup`	Which cgroups the process belongs to for each controller
`cat /proc/<pid>/mountinfo`	Process mount namespace	`cat /proc/$PID/mountinfo | head -20`	Mount points visible to the process with IDs and options
`capsh`	Capability inspection	`capsh --decode=$(cat /proc/$PID/status | grep CapEff | awk '{print $2}')`	Effective capabilities of the process
`runc`	Reference OCI runtime	`runc events container-id`	Container lifecycle events from reference implementation

Custom Debugging Utilities for Your Runtime

You should build these debugging aids directly into your container runtime:

1. **Debug Mode Flag**: Add `--debug` flag to CLI that enables verbose logging of all system calls and operations.
2. **Inspection Subcommands**:
   - `container-runtime inspect <container-id>` : Show complete container state including namespace IDs, cgroup paths, mount points, and network configuration.
   - `container-runtime debug exec <container-id> <command>` : Execute command in container namespaces without starting the container process.
   - `container-runtime logs <container-id>` : Capture and display stdout/stderr from container process.
3. **State Dump on Failure**: When a container fails to start, automatically dump:
   - Current namespace IDs
   - cgroup hierarchy

- Mount table
- Network interfaces
- Last 20 lines of kernel log (`dmesg`)

4. **Orphan Detection Cleanup**: Periodically scan for resources (cgroups, network namespaces, mount points) that don't have corresponding container state records and clean them up.

> **Key Insight:** The most powerful debugging technique is building observability directly into your runtime. Instrument every major operation with structured logging that includes the container ID, namespace IDs, and resource paths. When something fails, you'll have a complete audit trail of what was attempted and where it went wrong.

Implementation Guidance

This subsection provides practical implementation guidance for adding debugging capabilities to your container runtime.

Technology Recommendations

Component	Simple Option	Advanced Option
Logging	Structured logging with `logrus` or `zap`	OpenTelemetry tracing with Jaeger
Inspection	CLI commands using `nsenter` and `cgget`	HTTP API with WebSocket for real-time metrics
Tracing	`strace` wrapper for system call tracing	eBPF-based tracing with `bcc` tools
Metrics	Reading cgroup stats files directly	Prometheus metrics exporter

Recommended File/Module Structure

```
project-root/
  cmd/
    container-runtime/
      main.go              # CLI entry point with debug flags
      debug.go             # Debug subcommands implementation
  internal/
    debug/
      inspector.go         # Container state inspection logic
      tracer.go            # System call tracing wrapper
      metrics.go           # Metrics collection from cgroups
    runtime/
      container.go         # Add debug methods to Container type
      manager.go           # Add inspection methods to ContainerRuntime
    utils/
      nsenter.go           # Wrapper for nsenter operations
      cgroup-inspect.go    # cgroup inspection utilities
```

Infrastructure Starter Code

Here's a complete, ready-to-use debugging inspector that can be integrated into your runtime:

```go
// internal/debug/inspector.go

package debug


import (

    "encoding/json"

    "fmt"

    "io/ioutil"

    "os"

    "path/filepath"

    "strings"

    "syscall"


    "github.com/sirupsen/logrus"

)


// ContainerInspector provides methods to inspect container state

type ContainerInspector struct {

    basePath string

    logger   *logrus.Logger

}


// NewContainerInspector creates a new inspector instance

func NewContainerInspector(basePath string) *ContainerInspector {

    return &ContainerInspector{

        basePath: basePath,

        logger:   logrus.New(),

    }

}


// InspectContainer gathers comprehensive debugging information about a container

func (ci *ContainerInspector) InspectContainer(containerID string) (map[string]interface{}, error) {

    info := make(map[string]interface{})



    // Basic container info

    info["container_id"] = containerID

    info["timestamp"] = time.Now().Format(time.RFC3339)



    // Try to find container process

    pid, err := ci.findContainerPID(containerID)

    if err != nil {

        ci.logger.Warnf("Could not find container process: %v", err)
```

```
    } else {
        info["pid"] = pid

        // Collect namespace information
        nsInfo, err := ci.collectNamespaceInfo(pid)
        if err != nil {
            ci.logger.Warnf("Could not collect namespace info: %v", err)
        } else {
            info["namespaces"] = nsInfo
        }

        // Collect cgroup information
        cgroupInfo, err := ci.collectCgroupInfo(pid)
        if err != nil {
            ci.logger.Warnf("Could not collect cgroup info: %v", err)
        } else {
            info["cgroups"] = cgroupInfo
        }

        // Collect mount information
        mountInfo, err := ci.collectMountInfo(pid)
        if err != nil {
            ci.logger.Warnf("Could not collect mount info: %v", err)
        } else {
            info["mounts"] = mountInfo
        }
    }

    // Collect container filesystem information
    fsInfo, err := ci.collectFilesystemInfo(containerID)
    if err != nil {
        ci.logger.Warnf("Could not collect filesystem info: %v", err)
    } else {
        info["filesystem"] = fsInfo
    }

    // Collect network information
    netInfo, err := ci.collectNetworkInfo(containerID)
```

```
    if err != nil {
        ci.logger.Warnf("Could not collect network info: %v", err)
    } else {
        info["network"] = netInfo
    }

    return info, nil
}

// findContainerPID searches for the container's main process
func (ci *ContainerInspector) findContainerPID(containerID string) (int, error) {
    // Check in container state directory
    statePath := filepath.Join(ci.basePath, "containers", containerID, "state.json")
    if _, err := os.Stat(statePath); err == nil {
        data, err := ioutil.ReadFile(statePath)
        if err != nil {
            return 0, err
        }

        var state struct {
            Pid int `json:"pid"`
        }
        if err := json.Unmarshal(data, &state); err != nil {
            return 0, err
        }

        if state.Pid > 0 {
            // Verify process still exists
            if _, err := os.Stat(filepath.Join("/proc", fmt.Sprintf("%d", state.Pid))); err == nil {
                return state.Pid, nil
            }
        }
    }

    // Fallback: search through all processes
    return ci.scanProcessesForContainer(containerID)
}

// collectNamespaceInfo reads namespace IDs from /proc/<pid>/ns/
```

```go
func (ci *ContainerInspector) collectNamespaceInfo(pid int) (map[string]string, error) {
    nsInfo := make(map[string]string)
    nsTypes := []string{"pid", "mnt", "net", "uts", "ipc", "user"}

    for _, nsType := range nsTypes {
        nsPath := filepath.Join("/proc", fmt.Sprintf("%d", pid), "ns", nsType)
        target, err := os.Readlink(nsPath)
        if err != nil {
            return nil, err
        }
        nsInfo[nsType] = target
    }

    return nsInfo, nil
}

// DumpToFile writes inspection results to a JSON file
func (ci *ContainerInspector) DumpToFile(containerID string, outputPath string) error {
    info, err := ci.InspectContainer(containerID)
    if err != nil {
        return err
    }

    data, err := json.MarshalIndent(info, "", "  ")
    if err != nil {
        return err
    }

    return ioutil.WriteFile(outputPath, data, 0644)
}
```

Core Logic Skeleton Code

Here's skeleton code for adding debug commands to your CLI:

```go
// cmd/container-runtime/debug.go

package main


import (

    "encoding/json"

    "fmt"

    "os"


    "github.com/spf13/cobra"

    "your-project/internal/debug"

    "your-project/internal/runtime"

)


// NewDebugCommand creates the debug command tree

func NewDebugCommand(r *runtime.ContainerRuntime) *cobra.Command {

    cmd := &cobra.Command{

        Use:   "debug",

        Short: "Debugging commands for container inspection",

        Long:  "Commands to inspect container state, namespaces, cgroups, and other low-level details",

    }


    cmd.AddCommand(

        newDebugInspectCommand(r),

        newDebugExecCommand(r),

        newDebugLogsCommand(r),

        newDebugTraceCommand(r),

    )


    return cmd

}


// newDebugInspectCommand creates the inspect subcommand

func newDebugInspectCommand(r *runtime.ContainerRuntime) *cobra.Command {

    return &cobra.Command{

        Use:   "inspect [container-id]",

        Short: "Inspect container state and configuration",

        Args:  cobra.ExactArgs(1),

        RunE: func(cmd *cobra.Command, args []string) error {

            containerID := args[0]
```

```
            // TODO 1: Get the container from runtime state manager
            // container, err := r.GetContainer(containerID)
            // if err != nil { return fmt.Errorf("container not found: %v", err) }

            // TODO 2: Create a ContainerInspector instance
            // inspector := debug.NewContainerInspector(r.StorePath())

            // TODO 3: Collect inspection data
            // info, err := inspector.InspectContainer(containerID)
            // if err != nil { return fmt.Errorf("inspection failed: %v", err) }

            // TODO 4: Output as JSON or formatted text based on --format flag
            // if jsonOutput { json.NewEncoder(os.Stdout).Encode(info) }
            // else { printFormattedInfo(info) }

            // TODO 5: If --output flag specified, write to file
            // if outputFile != "" { inspector.DumpToFile(containerID, outputFile) }

            return nil
        },
    }
}

// newDebugExecCommand creates the exec subcommand for debugging
func newDebugExecCommand(r *runtime.ContainerRuntime) *cobra.Command {
    return &cobra.Command{
        Use:   "exec [container-id] [command]",
        Short: "Execute command in container namespaces",
        Long:  "Run a command in the container's namespaces for debugging purposes",
        Args:  cobra.MinimumNArgs(2),
        RunE: func(cmd *cobra.Command, args []string) error {
            containerID := args[0]
            command := args[1:]

            // TODO 1: Get container process PID
            // pid, err := getContainerPID(containerID)
            // if err != nil { return err }

            // TODO 2: Build namespace paths
```

```
            // nsPaths := []string{
            //     fmt.Sprintf("/proc/%d/ns/pid", pid),
            //     fmt.Sprintf("/proc/%d/ns/mnt", pid),
            //     fmt.Sprintf("/proc/%d/ns/net", pid),
            // }

            // TODO 3: Use nsenter to execute command in those namespaces
            // cmd := exec.Command("nsenter", append([]string{"-t", fmt.Sprintf("%d", pid), "--mount", "--net", "--pid"}, command...)...)
            // cmd.Stdin = os.Stdin
            // cmd.Stdout = os.Stdout
            // cmd.Stderr = os.Stderr
            // return cmd.Run()

            return nil
        },
    }
}
```

Language-Specific Hints

- **Go-specific tips for debugging:**
  - Use `syscall.PtraceAttach()` and `syscall.PtraceDetach()` for low-level process tracing
  - The `os/exec` package's `Cmd.SysProcAttr` field lets you set namespace flags for child processes
  - Use `gops` (Go process inspector) to debug the runtime itself: `import "github.com/google/gops/agent"`
  - For logging, use structured logging with fields: `logrus.WithField("container_id", containerID).Error("Failed to start")`
  - Use `runtime.Caller()` to add stack trace information to error messages
- **System call tracing in Go:**

```go
// Wrap system calls with logging

func tracedSyscall(name string, fn func() error) error {

    logrus.WithField("syscall", name).Debug("Enter")

    start := time.Now()

    err := fn()

    elapsed := time.Since(start)

    logrus.WithFields(logrus.Fields{

        "syscall": name,

        "elapsed": elapsed,

        "error":   err,

    }).Debug("Exit")

    return err

}



// Usage

tracedSyscall("pivot_root", func() error {

    return syscall.PivotRoot(newroot, putold)

})
```

Milestone Checkpoint

After implementing debugging features, verify them with these tests:

1. **Basic Inspection Test:**

```bash
# Start a container

./container-runtime run alpine echo "hello"



# Get the container ID from output or list

CONTAINER_ID=$(./container-runtime list | awk 'NR==2 {print $1}')



# Inspect it

./container-runtime debug inspect $CONTAINER_ID --format json



# Expected: JSON output with namespace IDs, cgroup paths, mount points

# If empty or error, check that the inspector can find the container process
```

2. **Namespace Debugging Test:**

```bash
# Start a container with network
./container-runtime run --network bridge alpine sleep 60 &

# Inspect network namespace
./container-runtime debug inspect $CONTAINER_ID | grep -A5 "network"

# Expected: Shows network namespace ID, interfaces, IP address
# If missing, check /proc/<pid>/ns/net exists and is readable
```

3. **Filesystem Debugging Test:**

```bash
# Create a file in container
./container-runtime run alpine sh -c "echo test > /tmp/debug.txt"

# Inspect filesystem layers
./container-runtime debug inspect $CONTAINER_ID | grep -A10 "filesystem"

# Expected: Shows overlayfs layers, upperdir path, merged view
# Check that upperdir exists and contains the debug.txt file
```

Debugging Tips Implementation

Add these debugging helpers directly to your components:

```go
// internal/runtime/manager.go (add to ContainerRuntime)
func (r *ContainerRuntime) EnableDebugLogging() {
    r.debugMode = true
    // TODO: Set verbose logging for all components
    // TODO: Enable system call tracing
    // TODO: Log all state transitions
}

func (r *ContainerRuntime) CollectDebugInfo(containerID string) (string, error) {
    // TODO: Gather dmesg output
    // TODO: Capture iptables rules
    // TODO: Dump cgroup hierarchies
    // TODO: Save to timestamped debug directory
    return debugDir, nil
}
```

Remember to build debugging capabilities incrementally—start with basic inspection commands, then add more sophisticated tracing as you encounter specific problems. The most valuable debugging tool is the one you build to solve your current problem.

Future Extensions

> **Milestone(s):** All milestones (future extensions build upon the foundation established by all six implementation milestones)

This section explores potential enhancements to our container runtime beyond the core implementation. While the current design provides a functional container runtime with all essential isolation primitives, real-world container systems include numerous additional features for security, usability, and production readiness. Understanding these extensions helps contextualize how our educational implementation compares to production systems like Docker and Kubernetes, and provides a roadmap for continued learning.

Potential Enhancement Ideas

The following extensions represent natural evolution points for the container runtime, categorized by their primary focus area:

Security Hardening

User Namespace Support: Currently, our container runs with the same user privileges as the host process (typically root). User namespaces would allow mapping host user IDs to different user IDs within the container, enabling containers to run as "root" inside while being non-root on the host.

Seccomp BPF Filters: System call filtering using seccomp would restrict the container's ability to make dangerous system calls (like `reboot`, `mount`, or `ptrace`), significantly reducing the attack surface.

Capabilities Management: Instead of running containers with all Linux capabilities, we could drop dangerous capabilities (like `CAP_SYS_ADMIN`, `CAP_NET_RAW`) and grant only necessary ones based on the container's purpose.

SELinux/AppArmor Integration: Mandatory Access Control (MAC) systems provide additional security boundaries beyond namespace isolation, enforcing fine-grained policies on file access, network operations, and process interactions.

Container Orchestration Features

Container Health Checks: Periodic execution of user-defined health check commands within the container, with automatic container restart if checks fail repeatedly.

Logging Driver Infrastructure: Support for multiple logging backends (syslog, journald, fluentd) with configurable log rotation, formatting, and aggregation.

Metrics Collection: Automatic collection and export of container resource usage statistics (CPU, memory, network, block I/O) via cgroup metrics and process monitoring.

Container Dependency Management: Start order coordination for containers that depend on services provided by other containers (e.g., database container must be ready before web application container).

Storage and Networking Enhancements

Volume Management: First-class support for persistent storage volumes with configurable drivers (local directories, NFS, cloud storage) and lifecycle management.

Network Plugins: Extensible network architecture supporting multiple network drivers (bridge, host, macvlan, ipvlan, overlay networks) via a plugin interface.

IPv6 Support: Full dual-stack IPv4/IPv6 networking with proper address allocation, routing, and firewall rules for IPv6.

Service Discovery: Built-in DNS-based service discovery allowing containers to find each other by name without static IP configuration.

Usability and Production Features

Container Pause/Resume: Support for freezing and thawing container execution using cgroup freezer controller, enabling live migration scenarios and debugging.

Container Checkpoint/Restore: Save container state to disk and restore it later, possibly on a different host, using CRIU (Checkpoint/Restore In Userspace).

Build System: Dockerfile-like build system for creating container images from source code with layer caching and dependency tracking.

Registry Authentication: Support for authenticated access to private container registries with token-based authentication and credential storage.

Windows Container Support: Cross-platform runtime supporting Windows containers using Windows-native isolation primitives (job objects, namespace, Hyper-V isolation).

The following table compares these enhancements by their implementation complexity and educational value:

Enhancement	Implementation Complexity	Educational Value	Production Relevance
User Namespace Support	Medium	High (deepens namespace understanding)	Critical
Seccomp BPF Filters	Low-Medium	Medium (system call security)	Critical
Volume Management	Medium	High (storage architecture)	Critical
Network Plugins	High	Medium (plugin architecture)	High
Health Checks	Low	Low (orchestration patterns)	High
Container Pause/Resume	Low	Medium (cgroup freezer)	Medium
Build System	High	High (image construction)	Critical
Checkpoint/Restore	High	High (process state serialization)	Medium

Architecture Decision: Incremental vs. Complete Rewrite

> **Decision: Incremental Enhancement Strategy**
>
> - **Context**: As an educational project, we want to maximize learning while maintaining a codebase that can evolve. Some enhancements require significant architectural changes that might conflict with the current simple design.
> - **Options Considered**:
>   1. **Incremental additions**: Add features to the existing codebase with minimal refactoring
>   2. **Modular plugin architecture**: Refactor core components to support pluggable implementations
>   3. **Complete rewrite**: Start fresh with a more extensible architecture incorporating lessons learned
> - **Decision**: Use incremental additions for simple enhancements and a gradual move toward plugin architecture for complex subsystems (networking, storage).
> - **Rationale**: The educational value comes from understanding the underlying Linux primitives, not building enterprise-grade extensibility. Incremental additions allow focused learning on specific features without overwhelming complexity. However, introducing plugin interfaces for networking and storage teaches important architectural patterns.
> - **Consequences**: Some refactoring will be needed when adding certain features, but the core isolation logic remains stable. The codebase may become slightly less elegant but remains approachable for learners.

Design Accommodations for Extensions

The current design includes several intentional accommodations that make future extensions easier. However, some enhancements require significant architectural changes.

Security Hardening Accommodations

Current Design Accommodations:

- The `ContainerConfig` type includes fields for security settings that are currently unimplemented
- The `NamespaceManager.CreateNamespaces` method accepts flags as a bitmask, making it easy to add `CLONE_NEWUSER`
- The cleanup stack pattern ensures resources are properly released even when security features fail

Required Modifications for User Namespaces:

Component	Current State	Required Changes
`NamespaceManager`	Supports PID, UTS, mount, network, IPC namespaces	Add `CLONE_NEWUSER` support and uid/gid mapping
`ContainerConfig`	Has basic security fields	Add `UserNamespaceConfig` with uidMaps and gidMaps
`FilesystemManager`	Assumes root access for mount operations	Handle file ownership when running as non-root user
CLI	No user namespace flags	Add `--user` flag and uid/gid mapping options

Implementation Approach:

1. Add `UserNamespaceConfig` to `ContainerConfig` with slice of `UIDMap` and `GIDMap` structs
2. Modify `NamespaceManager.CreateNamespaces` to handle `CLONE_NEWUSER` flag
3. Write uid/gid maps to `/proc/[pid]/uid_map` and `/proc/[pid]/gid_map` after clone

4. Update `FilesystemManager` to use `syscall.Chown` for critical directories when running mapped

Seccomp Integration Strategy:

```go
// Current ContainerConfig would expand to include:
type SecurityConfig struct {
    SeccompProfile  *SeccompProfile `json:"seccomp,omitempty"`
    AppArmorProfile string          `json:"apparmor,omitempty"`
    Capabilities    *Capabilities   `json:"capabilities,omitempty"`
    NoNewPrivileges bool            `json:"noNewPrivileges,omitempty"`
    ReadonlyRootfs  bool            `json:"readonlyRootfs,omitempty"`
}
```

Storage System Extensions

Volume Management Architecture:

The current filesystem isolation uses OverlayFS for the root filesystem but lacks persistent volume support. A volume system would require:

1. **Volume Abstraction Layer**: Interface defining volume operations (create, mount, unmount, delete)
2. **Volume Drivers**: Pluggable implementations for different storage backends
3. **Volume Lifecycle**: Integration with container lifecycle (mount before start, unmount after stop)
4. **Volume Specification**: Extended `ContainerConfig` to declare volume mounts

Proposed Volume Data Model:

```go
type VolumeMount struct {
    Source      string   // Volume name or host path
    Destination string   // Container path
    Type        string   // "bind", "volume", "tmpfs"
    Options     []string // "ro", "rw", "z", "Z"
}

type VolumeDriver interface {
    Create(name string, opts map[string]string) (Volume, error)
    Remove(name string) error
    Mount(name string) (string, error)  // returns mount path
    Unmount(name string) error
}

type VolumeManager struct {
    drivers map[string]VolumeDriver
    volumes map[string]Volume
}
```

Integration Points:

- Modify `FilesystemManager.SetupRootfs` to handle volume mounts after rootfs setup
- Extend `ContainerRuntime.createContainerSequence` to call volume mounting in phase 4 (filesystem preparation)
- Add volume cleanup to `CleanupManager` for proper resource cleanup

Networking Plugin Architecture

Current Network Limitations: The `NetworkManager` implements a single bridge network driver hardcoded into the component. A plugin architecture would allow multiple network types (host, none, macvlan, overlay).

Proposed Plugin Interface:

```go
type NetworkDriver interface {
    CreateNetwork(name string, opts map[string]interface{}) (Network, error)
    DeleteNetwork(name string) error
    SetupContainerNetwork(containerID string, netnsPath string, config NetworkConfig) error
    TeardownContainerNetwork(containerID string, netnsPath string, config NetworkConfig) error
}

type NetworkManager struct {
    drivers   map[string]NetworkDriver
    ipam      *IPAM
    // ... existing fields
}

func (nm *NetworkManager) RegisterDriver(name string, driver NetworkDriver) {
    nm.drivers[name] = driver
}
```

Required Refactoring:

1. Extract current bridge network logic into `BridgeDriver` implementing `NetworkDriver`
2. Modify `NetworkManager.SetupNetwork` to delegate to the appropriate driver based on `NetworkConfig.Mode`
3. Add driver discovery and loading mechanism (static compilation or dynamic loading)
4. Update CLI to support network driver selection via `--network` flag

Network Configuration Evolution:

```go
// Current NetworkConfig
type NetworkConfig struct {
    Mode         string        `json:"mode"`           // "bridge", "host", "none"
    BridgeName   string        `json:"bridgeName"`     // "byod-bridge"
    IPAddress    string        `json:"ipAddress"`      // "10.0.0.2"
    PortMappings []PortMapping `json:"portMappings"`   // port forwards
}

// Enhanced NetworkConfig with plugin support
type NetworkConfig struct {
    Driver       string                 `json:"driver"`       // "bridge", "macvlan", "overlay"
    NetworkName  string                 `json:"networkName"`  // "byod-network"
    Options      map[string]interface{} `json:"options"`      // driver-specific options
    IPAddress    string                 `json:"ipAddress"`     // optional override
    PortMappings []PortMapping          `json:"portMappings"`
}
```

Orchestration Features Integration

Health Check Implementation:

Health checks require periodic execution of commands within the container and state tracking:

```go
type HealthCheckConfig struct {
    Test        []string      `json:"test"`        // ["CMD", "curl", "-f", "http://localhost/health"]
    Interval    time.Duration `json:"interval"`    // 30s
    Timeout     time.Duration `json:"timeout"`     // 30s
    StartPeriod time.Duration `json:"startPeriod"` // 0s
    Retries     int           `json:"retries"`     // 3
}

type ContainerHealth struct {
    Status      string    `json:"status"`      // "starting", "healthy", "unhealthy"
    FailingStreak int     `json:"failingStreak"`
    Log         []HealthCheckResult `json:"log"`
}

type HealthMonitor struct {
    containers map[string]*ContainerHealthTracker
    executor   HealthCheckExecutor
    mu         sync.RWMutex
}
```

Integration Challenges:

1. **Namespace Access**: Health check commands must execute within the container's namespaces
2. **Resource Isolation**: Health checks should respect container resource limits
3. **State Management**: Health status must be stored and accessible via CLI/API
4. **Automatic Remediation**: Unhealthy containers might need automatic restart

Design Accommodation: The current `ContainerRuntime` tracks container state but would need to extend `Container` struct with health fields and add a background monitoring goroutine.

Metrics Collection System

Current Capabilities: The `CgroupManager.GetStats` method already collects basic resource usage statistics. This could be extended and exposed.

Enhanced Metrics Architecture:

```go
type ContainerMetrics struct {
    Timestamp   time.Time              `json:"timestamp"`
    CPU         CPUMetrics             `json:"cpu"`
    Memory      MemoryMetrics          `json:"memory"`
    BlockIO     BlockIOMetrics         `json:"blockIO"`
    Network     NetworkMetrics         `json:"network"`
    PIDs        PIDMetrics             `json:"pids"`
}

type MetricsCollector struct {
    cgroupManager *CgroupManager
    networkManager *NetworkManager
    interval      time.Duration
    subscribers   []MetricsSubscriber
    containers    map[string]*ContainerMetricsHistory
}

type MetricsSubscriber interface {
    OnMetrics(containerID string, metrics ContainerMetrics)
}
```

Integration Points:

1. Extend `CgroupManager.GetStats` to collect additional metrics (throttling, pressure)
2. Add network statistics collection via `/sys/class/net` or netlink
3. Create metrics aggregation goroutine in `ContainerRuntime`
4. Add CLI command `byod stats` to display live metrics
5. Optional: Export metrics via Prometheus endpoint or OpenMetrics format

Build System Implementation

Current Image Handling: The runtime pulls and extracts existing images but doesn't build them from source.

Build System Components:

Component	Responsibility	Integration Points
Dockerfile Parser	Parse build instructions into execution graph	Standalone component, outputs build spec
Layer Builder	Execute build steps, creating new layers	Uses `FilesystemManager` for layer creation
Build Cache	Cache intermediate layers for faster rebuilds	Integrates with `LayerCache`
Image Assembler	Create final image manifest and config	Uses `ImageStore` to save final image

Build Process Flow:

1. Parse Dockerfile into sequence of build stages
2. For each stage, start with base image or scratch
3. Execute instructions (RUN, COPY, ADD, etc.) in temporary container
4. Commit each successful step as a new layer
5. Apply layer to next step (like OverlayFS during build)
6. Finalize image with metadata and entrypoint

Key Technical Challenge: The build process needs to execute commands in isolated environments (containers) but also needs to copy files between build stages and host. This requires temporary container creation and filesystem snapshotting after each step.

Windows Container Support

Architectural Implications: Windows containers use fundamentally different isolation primitives, requiring a complete abstraction layer.

Proposed Platform Abstraction:

```go
type IsolationProvider interface {
    CreateIsolatedProcess(config ProcessConfig) (IsolatedProcess, error)
    SetupFilesystem(rootfs string, layers []string) (string, error)
    SetupNetwork(netConfig NetworkConfig) (NetworkEndpoint, error)
    ApplyResourceLimits(pid int, limits ResourceLimits) error
    GetStats(pid int) (ResourceStats, error)
}

type LinuxIsolationProvider struct {
    // Uses namespaces, cgroups, pivot_root
}

type WindowsIsolationProvider struct {
    // Uses job objects, silos, Windows namespace
}

type ContainerRuntime struct {
    isolationProvider IsolationProvider
    // ... other fields
}
```

Design Impact: This represents a major architectural shift from the current Linux-specific implementation. The abstraction would need to cover all isolation primitives while maintaining consistent behavior across platforms.

Registry Authentication System

Current Limitation: The `RegistryClient` assumes anonymous access to public registries.

Authentication Flow:

1. Attempt anonymous pull
2. If 401/403, check for credentials in config file
3. Request authentication token from registry auth service
4. Retry request with Bearer token
5. Cache token for subsequent requests

Required Changes:

```go
type RegistryAuth struct {
    Username string `json:"username"`
    Password string `json:"password"`
    Auth     string `json:"auth"` // Base64 encoded "username:password"
    Server   string `json:"server"`
}

type AuthStore interface {
    GetCredentials(registry string) (RegistryAuth, error)
    SetCredentials(registry string, auth RegistryAuth) error
}

type AuthenticatedRegistryClient struct {
    baseClient    *RegistryClient
    authStore     AuthStore
    tokenCache    map[string]string // registry -> token
    mu            sync.RWMutex
}
```

Integration Points: Modify `RegistryClient.PullManifest` and `RegistryClient.PullLayer` to handle authentication challenges and token acquisition.

Summary of Extension Readiness

The current design accommodates extensions through several key characteristics:

1. **Modular Component Design**: Each manager (Namespace, Cgroup, Filesystem, Network) operates independently, allowing replacement or enhancement of individual components.
2. **Clean Separation of Concerns**: The 8-phase container creation sequence provides clear hooks for adding new preparation steps (e.g., volume mounting between phases 4 and 5).
3. **Cleanup Stack Pattern**: The `CleanupManager` ensures resources are properly released, which is essential when adding complex features that might fail mid-operation.
4. **Config-Driven Architecture**: The `ContainerConfig` type can be extended with new fields without breaking existing functionality.
5. **State Machine Foundation**: The container state machine provides a framework for adding new states (e.g., "Paused", "Unhealthy") and transitions.

However, some extensions would require significant refactoring:

- Plugin architectures for networking and storage
- Cross-platform support
- Build system integration
- Comprehensive security features

For educational purposes, implementing selected extensions (like user namespaces or volume support) provides excellent learning opportunities while keeping the project manageable. Production container runtimes have evolved over years with contributions from hundreds of developers—our implementation captures the essential essence while remaining approachable for learning.

Implementation Guidance

Technology Recommendations for Extensions

Extension Category	Simple Implementation	Advanced Implementation
Security Hardening	Basic seccomp profiles from OCI spec	Dynamic BPF program generation based on container behavior
Volume Management	Local bind mounts with configurable propagation	Plugin system with NFS, iSCSI, cloud storage drivers
Network Plugins	Hardcoded bridge and host modes	CNI (Container Network Interface) compatible plugin loader
Health Checks	Simple command execution via `exec` in container	Integrated with process supervision and automatic restart
Metrics Collection	Periodic cgroup stats polling	Real-time metrics via eBPF and Prometheus endpoint
Build System	Simple layer-by-layer execution without cache	Full Dockerfile support with smart caching and multi-stage builds

Recommended File Structure for Extensions

```
byod/
  internal/
    security/                    # Security enhancements
      seccomp/
        seccomp.go               # Seccomp profile parsing and application
        default-profile.json     # Default seccomp profile
      capabilities/
        capabilities.go          # Linux capabilities management
      apparmor/
        apparmor.go              # AppArmor profile loading
    volumes/                     # Volume management system
      volume.go                  # Volume interface and types
      localdriver.go             # Local volume driver
      manager.go                 # Volume manager
      mount.go                   # Mount helpers
    networking/
      drivers/                   # Network driver implementations
        bridge.go                # Existing bridge driver (refactored)
        host.go                  # Host network driver
        macvlan.go               # Macvlan driver
        plugin.go                # Plugin interface
      cni/                       # CNI integration
        cni.go                   # CNI plugin executor
    health/                      # Health check system
      checker.go                 # Health check execution
      monitor.go                 # Background health monitoring
      executor.go                # Command execution in container
    metrics/                     # Metrics collection
      collector.go               # Metrics collection goroutine
      prometheus.go              # Prometheus exporter
      types.go                   # Metrics data structures
    build/                       # Build system
      dockerfile/
        parser.go                # Dockerfile parser
        instructions.go          # Instruction implementations
      builder.go                 # Build orchestration
      cache.go                   # Build cache
      executor.go                # Build step execution in container
    registry/                    # Enhanced registry client
      auth/
        authenticator.go         # Registry authentication
        credentialstore.go       # Credential storage (~/.docker/config.json)
        token.go                 # Token management
    platform/                    # Cross-platform abstractions
      isolation.go               # IsolationProvider interface
      linux/                     # Linux implementation
        isolation_linux.go
      windows/                   # Windows implementation (stub)
        isolation_windows.go
  cmd/
    byod-build/                  # Build command
      main.go
```

Security Hardening Starter Code

```go
// internal/security/seccomp/seccomp.go

package seccomp

import (
    "encoding/json"
    "fmt"
    "os"

    "github.com/seccomp/libseccomp-golang"
    "golang.org/x/sys/unix"
)

// SeccompProfile represents a seccomp filter configuration
type SeccompProfile struct {
    DefaultAction string           `json:"defaultAction"`
    Architectures []string         `json:"architectures"`
    Syscalls      []SyscallRule    `json:"syscalls"`
}

// SyscallRule defines allowed/disallowed syscalls
type SyscallRule struct {
    Names  []string `json:"names"`
    Action string   `json:"action"`
    Args   []Arg    `json:"args,omitempty"`
}

// LoadProfile loads a seccomp profile from JSON file
func LoadProfile(path string) (*SeccompProfile, error) {
    data, err := os.ReadFile(path)
    if err != nil {
        return nil, fmt.Errorf("read seccomp profile: %w", err)
    }

    var profile SeccompProfile
    if err := json.Unmarshal(data, &profile); err != nil {
        return nil, fmt.Errorf("parse seccomp profile: %w", err)
    }

    return &profile, nil
```

```
}

// ApplyProfile applies the seccomp profile to the current process
func ApplyProfile(profile *SeccompProfile) error {
    if profile == nil {
        return nil // No profile to apply
    }

    filter, err := seccomp.NewFilter(parseAction(profile.DefaultAction))
    if err != nil {
        return fmt.Errorf("create seccomp filter: %w", err)
    }
    defer filter.Release()

    // Add architecture support
    for _, arch := range profile.Architectures {
        if err := addArchitecture(filter, arch); err != nil {
            return fmt.Errorf("add architecture %s: %w", arch, err)
        }
    }

    // Add syscall rules
    for _, rule := range profile.Syscalls {
        for _, name := range rule.Names {
            sc, err := seccomp.GetSyscallFromName(name)
            if err != nil {
                // Syscall might not exist on this architecture
                continue
            }

            if err := filter.AddRule(sc, parseAction(rule.Action)); err != nil {
                return fmt.Errorf("add rule for %s: %w", name, err)
            }
        }
    }

    return filter.Load()
}
```

```
// parseAction converts string action to seccomp.Action
func parseAction(action string) seccomp.Action {
    switch action {
    case "SCMP_ACT_ALLOW":
        return seccomp.ActAllow
    case "SCMP_ACT_ERRNO":
        return seccomp.ActErrno.SetReturnCode(int16(unix.EPERM))
    case "SCMP_ACT_KILL":
        return seccomp.ActKill
    case "SCMP_ACT_KILL_PROCESS":
        return seccomp.ActKillProcess
    case "SCMP_ACT_TRACE":
        return seccomp.ActTrace.SetReturnCode(0)
    default:
        return seccomp.ActErrno.SetReturnCode(int16(unix.EPERM))
    }
}

// addArchitecture adds architecture support to filter
func addArchitecture(filter *seccomp.ScmpFilter, arch string) error {
    switch arch {
    case "SCMP_ARCH_X86_64":
        return filter.AddArch(seccomp.ArchAMD64)
    case "SCMP_ARCH_X86":
        return filter.AddArch(seccomp.ArchX86)
    case "SCMP_ARCH_ARM":
        return filter.AddArch(seccomp.ArchARM)
    case "SCMP_ARCH_AARCH64":
        return filter.AddArch(seccomp.ArchARM64)
    default:
        return fmt.Errorf("unsupported architecture: %s", arch)
    }
}

// DefaultProfile returns a default seccomp profile for containers
func DefaultProfile() *SeccompProfile {
    // Returns a sensible default profile similar to Docker's default
    // This would include the actual JSON profile
    return &SeccompProfile{
```

```
        DefaultAction: "SCMP_ACT_ERRNO",
        Architectures: []string{"SCMP_ARCH_X86_64"},
        Syscalls: []SyscallRule{
            {
                Names:  []string{"accept", "accept4", "access", "alarm", "bind"},
                Action: "SCMP_ACT_ALLOW",
            },
            // ... more syscalls
        },
    }
}
```

Volume Management Skeleton

```go
// internal/volumes/volume.go

package volumes

import (
    "fmt"
    "path/filepath"

    "github.com/yourusername/byod/internal/types"
)

// Volume represents a persistent storage volume
type Volume struct {
    Name       string                 `json:"name"`
    Driver     string                 `json:"driver"`
    Mountpoint string                 `json:"mountpoint"`
    Options    map[string]interface{} `json:"options"`
    Labels     map[string]string      `json:"labels"`
    CreatedAt  time.Time              `json:"createdAt"`
}

// VolumeDriver is the interface for volume driver implementations
type VolumeDriver interface {
    // Create creates a new volume with the given name and options
    Create(name string, opts map[string]interface{}) (Volume, error)

    // Remove deletes a volume
    Remove(name string) error

    // Mount mounts the volume and returns the mount path
    Mount(name string) (string, error)

    // Unmount unmounts the volume
    Unmount(name string) error

    // Path returns the path where the volume is stored (for bind mounts)
    Path(name string) (string, error)

    // DriverName returns the driver's name
    DriverName() string
```

```go
}

// VolumeManager manages all volumes and their drivers
type VolumeManager struct {
    drivers map[string]VolumeDriver
    volumes map[string]Volume
    baseDir string
    mu      sync.RWMutex
}

// NewVolumeManager creates a new volume manager
func NewVolumeManager(baseDir string) (*VolumeManager, error) {
    // TODO 1: Create base directory if it doesn't exist
    // TODO 2: Initialize drivers map
    // TODO 3: Load existing volumes from persistence file
    // TODO 4: Register default "local" driver
    // TODO 5: Return initialized manager
    return nil, fmt.Errorf("not implemented")
}

// CreateVolume creates a new volume
func (vm *VolumeManager) CreateVolume(name, driver string, opts map[string]interface{}) (Volume, error) {
    // TODO 1: Validate volume name (alphanumeric, underscores, hyphens)
    // TODO 2: Check if volume already exists
    // TODO 3: Get driver implementation
    // TODO 4: Call driver.Create() with name and options
    // TODO 5: Store volume metadata in volumes map
    // TODO 6: Persist volume metadata to disk
    // TODO 7: Return created volume
    return Volume{}, fmt.Errorf("not implemented")
}

// MountVolume mounts a volume for use by a container
func (vm *VolumeManager) MountVolume(name, containerID string) (string, error) {
    // TODO 1: Look up volume by name
    // TODO 2: Get volume driver
    // TODO 3: Call driver.Mount() to mount the volume
    // TODO 4: Return mount path
    return "", fmt.Errorf("not implemented")
}
```

```go
// UnmountVolume unmounts a volume after container stops
func (vm *VolumeManager) UnmountVolume(name, containerID string) error {
    // TODO 1: Look up volume by name
    // TODO 2: Check if any other containers are using this volume
    // TODO 3: If no other users, call driver.Unmount()
    // TODO 4: Update usage tracking
    return fmt.Errorf("not implemented")
}

// GetVolume returns volume information
func (vm *VolumeManager) GetVolume(name string) (Volume, error) {
    // TODO 1: Look up volume in volumes map
    // TODO 2: Return volume if found
    // TODO 3: Return error if not found
    return Volume{}, fmt.Errorf("not implemented")
}

// ListVolumes returns all volumes
func (vm *VolumeManager) ListVolumes() []Volume {
    // TODO 1: Iterate through volumes map
    // TODO 2: Collect all volumes into slice
    // TODO 3: Return slice
    return nil
}

// RemoveVolume removes a volume
func (vm *VolumeManager) RemoveVolume(name string) error {
    // TODO 1: Check if volume exists
    // TODO 2: Check if volume is in use by any container
    // TODO 3: Get volume driver
    // TODO 4: Call driver.Remove()
    // TODO 5: Remove from volumes map
    // TODO 6: Persist changes to disk
    return fmt.Errorf("not implemented")
}

// LocalDriver implements VolumeDriver for local directories
type LocalDriver struct {
    baseDir string
}
```

```go
func (ld *LocalDriver) Create(name string, opts map[string]interface{}) (Volume, error) {
    // TODO 1: Create directory at filepath.Join(ld.baseDir, name)
    // TODO 2: Apply options (size limits, permissions)
    // TODO 3: Return Volume with Mountpoint set to directory path
    return Volume{}, fmt.Errorf("not implemented")
}

func (ld *LocalDriver) Remove(name string) error {
    // TODO 1: Check if directory exists
    // TODO 2: Remove directory and all contents
    return fmt.Errorf("not implemented")
}

func (ld *LocalDriver) Mount(name string) (string, error) {
    // TODO 1: Return path to volume directory (already mounted as filesystem)
    return "", fmt.Errorf("not implemented")
}

func (ld *LocalDriver) Unmount(name string) error {
    // TODO 1: For local driver, nothing to unmount
    return nil
}

func (ld *LocalDriver) Path(name string) (string, error) {
    // TODO 1: Return path to volume directory
    return "", fmt.Errorf("not implemented")
}

func (ld *LocalDriver) DriverName() string {
    return "local"
}
```

Language-Specific Hints for Extensions

Go-Specific Implementation Tips:

1. **Security Context**: Use `golang.org/x/sys/unix` package for seccomp and capabilities operations
2. **Plugin Loading**: For dynamic network/storage plugins, consider `plugin` package or compile-time registration
3. **Concurrent Health Checks**: Use `context.Context` with timeouts for health check execution
4. **Metrics Collection**: Use `prometheus/client_golang` for Prometheus integration if desired
5. **Windows Support**: Use build tags (`//go:build windows`) for platform-specific implementations
6. **Authentication**: Use `golang.org/x/oauth2` for registry token authentication flows
7. **Build Caching**: Use content-addressable storage with SHA256 digests for build cache keys

Integration Testing for Extensions:

```go
// Example test for volume management

func TestVolumeManagerIntegration(t *testing.T) {

    tmpDir := t.TempDir()

    vm, err := volumes.NewVolumeManager(tmpDir)

    require.NoError(t, err)



    vol, err := vm.CreateVolume("testvol", "local", map[string]interface{}{

        "size": "1GB",

    })

    require.NoError(t, err)

    require.Equal(t, "testvol", vol.Name)



    mountPath, err := vm.MountVolume("testvol", "test-container")

    require.NoError(t, err)

    require.DirExists(t, mountPath)



    // Write test file to volume

    testFile := filepath.Join(mountPath, "test.txt")

    require.NoError(t, os.WriteFile(testFile, []byte("test"), 0644))



    require.NoError(t, vm.UnmountVolume("testvol", "test-container"))



    // Volume should still exist

    vol2, err := vm.GetVolume("testvol")

    require.NoError(t, err)

    require.Equal(t, vol.Name, vol2.Name)



    require.NoError(t, vm.RemoveVolume("testvol"))

}
```

Milestone Checkpoints for Extensions

After Implementing Volume Support:

```
$ ./byod volume create mydata --driver=local
Volume "mydata" created successfully

$ ./byod run -v mydata:/data alpine ls /data
(should show contents of empty volume)

$ ./byod run -v mydata:/data alpine touch /data/test.txt

$ ./byod run -v mydata:/data alpine ls /data
test.txt

$ ./byod volume ls
NAME    DRIVER    MOUNTPOINT
mydata  local     /var/lib/byod/volumes/mydata/_data
```

After Implementing User Namespaces:

```
$ ./byod run --user 1000:1000 alpine id
uid=1000 gid=1000 groups=1000

$ ps aux | grep alpine
(container process should run as non-root user on host)

$ ./byod run --user 0:0 --cap-add=ALL alpine sh
(should fail if capabilities are properly dropped)
```

After Implementing Health Checks:

```
$ ./byod run --health-cmd="curl -f http://localhost/health" \
             --health-interval=5s \
             --health-retries=3 \
             myapp

$ ./byod inspect mycontainer | jq .State.Health
{
  "Status": "healthy",
  "FailingStreak": 0,
  "Log": [...]
}
```

Debugging Tips for Extensions

Symptom	Likely Cause	How to Diagnose	Fix
Volume mount fails with "permission denied"	SELinux/AppArmor blocking mount	Check `dmesg` for SELinux/AppArmor denials	Use `z` or `Z` volume option for SELinux labeling
Container with user namespace can't write to mounted volume	UID/GID mismatch between host and container	Check file ownership on host with `ls -la`	Use `:uid=1000,gid=1000` volume option or ensure UID mapping
Health checks always fail	Command not found in container image	Run health command manually in container	Ensure health command exists in container PATH
Metrics collection causes high CPU usage	Too frequent polling of cgroup files	Check metrics collection interval	Increase collection interval or use more efficient polling
Build cache not working	Cache keys not matching actual inputs	Check cache key generation logic	Include all build context files in cache key hash
Registry authentication fails	Token expired or invalid	Enable debug logging in registry client	Implement token refresh logic with retry
Network plugin fails to load	Missing dependencies or wrong ABI	Check plugin logs and required symbols	Ensure plugin is built with compatible Go version and imports

Glossary

> **Milestone(s):** All milestones (this section provides a reference for all technical terms used throughout the design document)

Terms and Definitions

This glossary defines all key technical terms, acronyms, and domain-specific vocabulary used throughout the Build Your Own Docker design document. Terms are organized by conceptual domain to facilitate understanding of related concepts.

Container Runtime Core Concepts

Term	Definition
Container	An isolated, resource-controlled process environment that appears to the process as a complete operating system but shares the host kernel with other containers.
Container Runtime	The software responsible for creating, managing, and destroying containers using Linux primitives like namespaces and cgroups.
Process Isolation	The technique of using kernel features to prevent processes from seeing or interfering with processes, filesystems, network interfaces, and other system resources belonging to other containers or the host system.
Linux Kernel	The core operating system component that provides namespaces, cgroups, and other containerization primitives as system calls and kernel features.
System Call (syscall)	A programmatic way for a userspace program to request services from the Linux kernel, such as creating new processes, managing memory, or interacting with filesystems.
State Machine	A mathematical model of computation with a finite number of states, where the system transitions between states in response to events or inputs. Our container runtime uses a state machine to manage container lifecycle states (`Created` , `Running` , `Paused` , `Stopped` , `Removed`).
Content-Addressable Storage (CAS)	A storage paradigm where content is retrieved based on its cryptographic hash (digest) rather than its location. This ensures integrity and enables deduplication of identical content across multiple images.
Copy-on-Write (CoW)	An optimization strategy where multiple callers share the same resource until one needs to modify it, at which point a private copy is created. Used extensively in container filesystems to share base image layers.
Reference Counting	A technique for tracking the number of references to a shared resource, allowing the resource to be safely freed when the count reaches zero. Used for managing layer cache entries and container resources.
Configuration Merging	The process of combining settings from multiple sources (image configuration, user configuration, default values) with well-defined precedence rules to produce the final container configuration.
Compensating Transactions	A design pattern where for every operation that modifies system state, there is a corresponding cleanup operation that can undo or clean up that modification in case of failure.
Cleanup Stack	A pattern where operations push cleanup functions onto a stack as they succeed, ensuring resources are properly released in reverse order of acquisition if an error occurs.
Idempotent Cleanup	Cleanup operations that can be safely called multiple times without causing errors or side effects, crucial for robust error recovery in distributed systems.
Orphan Detection	The process of finding resources (cgroups, network interfaces, mount points) not associated with any known container, typically during system startup or cleanup operations.
Transaction Logging	Writing intentions to a log before performing operations, enabling crash recovery by replaying or compensating logged operations after a system failure.
Content Integrity	Ensuring that downloaded or stored content hasn't been corrupted, typically verified using cryptographic hashes (SHA-256 digests) before use.
Structured Logging	Logging with key-value fields instead of plain text messages, enabling better filtering, searching, and analysis of runtime events.
Debug Mode	A runtime mode with verbose logging, additional diagnostics, and potentially slower execution for troubleshooting container runtime issues.

Process Isolation (Namespaces)

Term	Definition
Namespace	A Linux kernel feature that isolates global system resources so that processes in one namespace have their own independent view of system resources like process IDs, hostnames, user IDs, mount points, and network interfaces.
PID Namespace	A namespace that isolates process IDs, allowing processes in different namespaces to have the same PID. The first process in a PID namespace gets PID 1 and typically acts as an init process for that namespace.
Network Namespace	A namespace that isolates network stack resources including network interfaces, IP addresses, routing tables, and firewall rules. Each network namespace has its own private loopback interface and can be connected to other namespaces via virtual Ethernet pairs.
Mount Namespace	A namespace that isolates mount points, allowing processes in different namespaces to have different views of the filesystem hierarchy. Mount operations (mount, unmount) in one namespace do not affect other namespaces.
UTS Namespace	A namespace that isolates system identifiers, specifically the hostname and domain name, allowing containers to have their own hostname independent of the host system.
IPC Namespace	A namespace that isolates inter-process communication resources such as System V IPC objects and POSIX message queues.
User Namespace	A namespace that isolates user and group ID mappings, allowing a process to have root privileges inside the namespace while being an unprivileged user outside.
Namespace ID	A unique identifier for a Linux namespace instance, visible as an inode number in `/proc/[pid]/ns/[type]` and used with the `setns()` system call to join existing namespaces.
clone()	A Linux system call that creates a new process (child) that may share parts of its execution context with the calling process (parent). Flags like `CLONE_NEWPID`, `CLONE_NEWNET` specify which namespaces to create for the new process.
unshare()	A Linux system call that allows a process to disassociate parts of its execution context that were previously shared with other processes, effectively creating new namespaces for the calling process without creating a new process.
setns()	A Linux system call that allows a process to join an existing namespace specified by a file descriptor, typically obtained from `/proc/[pid]/ns/[type]`.
SysProcAttr	A Go struct in the `syscall` package that allows setting process creation attributes, including namespace configuration flags for the `clone()` system call when creating new processes.
/proc/self/exe	A symbolic link in the proc filesystem that points to the current executable. Used in container runtimes to re-execute the runtime binary with different arguments for the child process in new namespaces.

Resource Limits (cgroups)

Term	Definition
cgroup (Control Group)	A Linux kernel feature for limiting, accounting for, and isolating resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes.
cgroup v1	The original cgroup implementation with separate hierarchies for different resource controllers (memory, cpu, blkio, etc.), allowing different process groupings for each controller.
cgroup v2	The unified cgroup hierarchy that provides a consistent interface with all controllers mounted under a single filesystem hierarchy, offering improved consistency and new features.
Controller	A cgroup subsystem that manages a specific resource type, such as `memory` for memory limits, `cpu` for CPU scheduling, or `pids` for process count limits.
cgroup Hierarchy	A tree structure of cgroups where child cgroups inherit resource limits from parent cgroups, with further restrictions possible at each level.
OOM Killer (Out-Of-Memory Killer)	A kernel mechanism that selects and terminates processes when the system is critically low on memory. cgroups can trigger the OOM killer within specific cgroups when memory limits are exceeded.
RSS (Resident Set Size)	The portion of a process's memory held in physical RAM (as opposed to swapped out to disk). cgroup memory limits typically apply to RSS plus kernel memory allocations.
CPU Shares	A relative weight in the CPU controller that determines the proportion of CPU time available to a cgroup relative to sibling cgroups when the system is under contention.
CPU Quota	An absolute limit in the CPU controller that caps the amount of CPU time a cgroup can consume within a specified period, expressed as `quota/period` microseconds.
PIDs Limit	A cgroup controller that restricts the maximum number of processes (including threads) that can be created within a cgroup, preventing fork bombs and runaway process creation.

Filesystem Isolation

Term	Definition
rootfs (Root Filesystem)	The filesystem mounted at the root directory (`/`) of a container, containing all binaries, libraries, and configuration files needed for the containerized process to run.
chroot	A system call that changes the root directory for the calling process and its children, restricting filesystem access to the subtree under the new root. Without mount namespace isolation, chroot is considered escapable.
pivot_root	A system call that atomically swaps the root filesystem of the calling process with a new root filesystem, then unmounts the old root. This provides stronger isolation than chroot when combined with mount namespaces.
Mount Propagation	Determines how mount and unmount events propagate between mount namespaces. `MS_SHARED` mounts propagate to peer namespaces, `MS_PRIVATE` mounts do not, and `MS_SLAVE` mounts receive but do not send propagation events.
procfs (proc Filesystem)	A virtual filesystem mounted at `/proc` that provides information about processes and other system information in a hierarchical file-like structure. Containers need their own procfs mount to see their own processes.
sysfs (sys Filesystem)	A virtual filesystem mounted at `/sys` that provides a hierarchical view of kernel devices, drivers, and other kernel parameters. Typically mounted read-only in containers.
tmpfs (Temporary Filesystem)	A temporary in-memory filesystem that uses RAM and/or swap space. Used for `/dev` , `/run` , and other temporary directories in containers.
Bind Mount	A mount operation that makes an existing directory or file available at another location in the filesystem hierarchy. Changes to the bind mount are reflected at the original location and vice versa.
Device Node	A special file in `/dev` that represents a hardware or virtual device. Containers typically need a minimal set of device nodes (`null` , `zero` , `random` , `urandom` , `tty` , `console`) for applications to function.

Layered Filesystem (OverlayFS)

Term	Definition
OverlayFS	A union filesystem that layers multiple directories (called layers) into a single unified view, commonly used in container runtimes to stack image layers efficiently.
Lower Directory (lowerdir)	In OverlayFS, one or more read-only directories that form the base layers of the union. Files are read from the highest priority lower directory where they exist.
Upper Directory (upperdir)	In OverlayFS, a writable directory where all modifications (creates, writes, deletes) are stored. This layer captures container filesystem changes without modifying the read-only lower layers.
Work Directory (workdir)	In OverlayFS, an empty directory used by the kernel for preparing files before moving them to the upper directory during copy-on-write operations. Must be on the same filesystem as the upper directory.
Merged View	The unified filesystem view presented to the container process, combining files from all lower directories with modifications from the upper directory.
Whiteout	A special file (character device with device numbers 0,0) in the upper directory that marks a file or directory in a lower layer as deleted. Named `.wh.<filename>` in OverlayFS.
Opaque Directory	A special extended attribute (`trusted.overlay.opaque="y"`) or directory named `overlay-opaque` that marks a directory in the upper layer as opaque, hiding all contents from lower layers within that directory.
Layer	A read-only filesystem snapshot representing a set of filesystem changes. Container images are composed of stacked layers, each representing a step in the image build process.
Copy-on-Write (CoW)	In filesystem context, a technique where a file is copied from a lower (read-only) layer to the upper (writable) layer only when it is modified, conserving disk space and reducing write amplification.
Union Mount	A mount that presents a composite view of multiple directories, with precedence rules determining which file appears when the same path exists in multiple source directories. OverlayFS is one implementation of union mounting.

Container Networking

Term	Definition
Network Namespace	A Linux namespace that provides an isolated network stack with its own network interfaces, IP addresses, routing tables, and firewall rules.
Virtual Ethernet Pair (veth pair)	A pair of virtual network devices that act as a tunnel between network namespaces. Packets transmitted on one device are received on the other, connecting namespaces like a virtual network cable.
Linux Bridge	A virtual switch implemented in the kernel that forwards packets between multiple network interfaces (physical or virtual). Used to connect multiple container veth endpoints on the same subnet.
iptables	A userspace utility for configuring Linux kernel firewall rules (netfilter). Used in container networking for NAT masquerading, port forwarding, and network policy enforcement.
NAT Masquerading	A form of network address translation where outbound packets from containers have their source IP address replaced with the host's IP address, allowing containers to reach external networks without publicly routable IPs.
Port Forwarding	The mapping of a port on the host to a port inside a container, allowing external connections to reach services running inside containers. Implemented via iptables DNAT (Destination Network Address Translation) rules.
IPAM (IP Address Management)	The process of allocating, tracking, and assigning IP addresses to containers from defined subnets, ensuring no address conflicts and efficient utilization of address space.
Dual-Stack Networking	Support for both IPv4 and IPv6 addressing simultaneously in container networks, requiring configuration of both address families in network namespaces and routing tables.

Image Format and OCI

Term	Definition
OCI (Open Container Initiative)	An open governance structure for creating open industry standards around container formats and runtimes, maintaining specifications for container images and runtime behavior.
OCI Image Specification	A specification that defines the format for container images, including the manifest, configuration, and layer descriptors, ensuring interoperability between different container tools.
OCI Runtime Specification	A specification that defines the configuration, execution environment, and lifecycle of containers, ensuring consistent behavior across different container runtimes.
Manifest	A JSON document describing the components of a container image, including layer digests, image configuration, and metadata. Serves as the entry point for locating and verifying image contents.
Digest	A cryptographic hash (typically SHA-256) used to uniquely and content-addressably identify a blob (layer, config, manifest) in an OCI image. Expressed as `algorithm:hex` (e.g., `sha256:abc123...`).
Blob	A binary large object stored in an OCI registry, such as a layer tar archive, image configuration JSON, or manifest JSON. Identified by its digest.
Tag	A human-readable alias that points to a specific image manifest in a registry, allowing users to reference images with names like `ubuntu:latest` rather than cryptographic digests.
Registry	A server that stores and distributes container images, implementing the OCI Distribution Specification. Docker Hub is a popular public registry; private registries can be deployed internally.
Authentication Flow	The process of obtaining and presenting credentials (username/password, tokens) to a registry to authenticate requests for pulling or pushing images.
Token Authentication	A bearer token-based authentication mechanism where the registry returns a short-lived token after initial authentication, used for subsequent API requests.

CLI and User Interface

Term	Definition
CLI (Command Line Interface)	The text-based interface through which users interact with the container runtime by typing commands, options, and arguments in a terminal or shell.
Inspection Command	A CLI subcommand (e.g., `inspect`) that gathers and displays detailed information about a container's configuration, state, network settings, and resource usage.
Sequence Diagram	A visual representation of component interactions over time, showing messages passed between actors and components in a specific sequence. Used in design documents to illustrate complex workflows.
State Transition	A change from one state to another in a state machine, triggered by an event or condition. In our container runtime, transitions like `Created` → `Running` occur when the user issues a `start` command.

State Management and Data Structures

Term	Definition
Container	The primary data structure representing a container instance, containing fields for `ID`, `Name`, `State`, `Config`, `Pid`, and `CreatedAt`.
ContainerConfig	Configuration settings for a container, including `Image` name, command (`Cmd`), environment variables (`Env`), working directory (`WorkingDir`), hostname (`Hostname`), resource limits (`Limits`), and network settings (`Network`).
ContainerState	An enumeration representing the current lifecycle state of a container: `Created`, `Running`, `Paused`, `Stopped`, or `Removed`.
ResourceLimits	Configuration for container resource constraints, including memory limit in megabytes (`MemoryMB`), CPU shares (`CPUShares`), and maximum process count (`PidsLimit`).
NetworkConfig	Network configuration for a container, specifying the network mode (`Mode`), bridge name (`BridgeName`), IP address (`IPAddress`), and port mappings (`PortMappings`).
PortMapping	A mapping between a host port and a container port with a specified protocol (TCP/UDP), allowing external access to containerized services.
Image	A data structure representing a container image, containing fields for `Name`, `Digest`, `Layers`, and `Config`.
ImageLayer	A single layer in a container image, identified by its `Digest`, with `Size` in bytes and local filesystem `Path` where the extracted layer contents are stored.
ImageConfig	Configuration embedded in an image, specifying the default `Entrypoint`, `Cmd`, `Env`, and `WorkingDir` for containers created from this image.
ContainerStore	A component responsible for persisting container state to disk, typically storing JSON representations in a directory structure under a `basePath`.

Error Handling and Debugging

Term	Definition
Property-Based Testing	A testing methodology that verifies properties hold for automatically generated inputs, using frameworks like QuickCheck to generate thousands of test cases and find edge cases.
Fuzz Testing	A testing technique that provides random, unexpected, or invalid inputs to a program to uncover security vulnerabilities, crashes, or incorrect behavior.
Integration Test	A test that combines multiple components to verify they work together correctly, often involving actual system calls and external dependencies.
System Test	An end-to-end test of the complete system from user command to container execution, verifying the entire workflow matches expected behavior.
Mock	A test double that simulates the behavior of a real component, allowing isolated testing of a component without its dependencies.
Test Fixture	A fixed state used as a baseline for running tests, ensuring tests start with known, reproducible conditions.
Race Condition	A defect where the output or behavior depends on the sequence or timing of uncontrollable events, often occurring in concurrent programming when multiple threads access shared data without proper synchronization.
Test Pyramid	A concept describing the ideal distribution of tests: many fast, isolated unit tests; fewer integration tests; and even fewer slow, expensive system tests.
Packet Tracing	Monitoring network packets through the networking stack using tools like `tcpdump`, `Wireshark`, or kernel tracing facilities to diagnose container networking issues.
Process Tree	The hierarchical relationship between parent and child processes, visible via commands like `pstree`. Container runtimes must manage process trees within containers to properly handle orphaned processes.
System Call Tracing	Monitoring system calls made by a process using tools like `strace` or `sysdig` to understand behavior, diagnose failures, or detect security issues.
Resource Accounting	Tracking resource usage (CPU, memory, I/O) by containers over time, enabling monitoring, billing, and capacity planning.

Security

Term	Definition
seccomp (Secure Computing Mode)	A Linux kernel feature for filtering system calls available to a process, allowing restriction of the kernel surface area exposed to containers to reduce attack vectors.
Capabilities	Fine-grained permissions that divide the omnipotent root privilege into distinct units, allowing containers to be granted specific privileges (e.g., `CAP_NET_ADMIN` for network configuration) without full root access.
Seccomp Profile	A JSON configuration defining which system calls are allowed, which are restricted, and what action to take (allow, error, kill) for filtered calls.
SyscallRule	A rule within a seccomp profile specifying system call names, action, and optional argument constraints.
User Namespace	A namespace that isolates user and group IDs, enabling privilege escalation within the namespace while maintaining unprivileged execution on the host—a key security feature for rootless containers.

Storage and Volumes

Term	Definition
Volume	A unit of persistent storage that can be mounted into containers, surviving container lifecycle operations (stop, remove). Volumes can be backed by host directories, network storage, or other storage drivers.
VolumeDriver	An interface defining operations for volume management: `Create`, `Remove`, `Mount`, `Unmount`, `Path`, and `DriverName`. Different drivers implement storage for different backends (local, NFS, cloud storage).
VolumeManager	A component that manages volumes using registered drivers, tracking volume state and providing a unified API for volume operations.
VolumeMount	A specification for mounting a volume into a container, including `Source` (volume name or host path), `Destination` (container mount path), `Type` (bind, volume), and mount `Options`.
Bind Mount	A type of volume mount that maps a host directory or file directly into a container, with changes reflected both directions.
tmpfs Mount	A mount of a temporary in-memory filesystem into a container, useful for sensitive data that shouldn't persist on disk.

Health and Metrics

Term	Definition
Health Check	A periodic command execution or HTTP request to verify that a containerized application is functioning correctly, with configurable interval, timeout, retries, and start period.
HealthCheckConfig	Configuration for health checks: `Test` command, `Interval` between checks, `Timeout` for each check, `StartPeriod` before checks begin, and `Retries` before marking unhealthy.
ContainerHealth	Current health status of a container: `Status` (healthy, unhealthy, starting), `FailingStreak` count, and `Log` of recent health check results.
HealthMonitor	A component that periodically executes health checks for running containers and updates their health status.
Metrics	Quantitative measurements of container resource usage, including CPU, memory, block I/O, network, and process counts.
ContainerMetrics	A snapshot of container metrics at a specific `Timestamp`, with detailed `CPU`, `Memory`, `BlockIO`, `Network`, and `PIDs` measurements.
MetricsCollector	A component that periodically gathers metrics from cgroups, network interfaces, and process information for running containers.

Future Extensions

Term	Definition
CNI (Container Network Interface)	A plugin standard for configuring container networking, allowing interchangeable network implementations (bridge, macvlan, ipvlan, etc.) via executable plugins.
Plugin Architecture	A system design that supports interchangeable components via defined interfaces, allowing third-party extensions for networking, volumes, logging, and other functionality.
Dual-Stack	Networking that supports both IPv4 and IPv6 simultaneously, requiring configuration of both address families in container network namespaces.
Credential Store	Secure storage for authentication credentials (e.g., Docker config.json, keychain) used to authenticate with container registries.
Build System	A toolchain for creating container images from source code (Dockerfiles, Buildpacks) rather than pulling pre-built images from registries.
Checkpoint/Restore	The process of saving a container's state (memory, CPU registers, open files) to disk and later restoring it to resume execution from the saved point, enabling live migration and faster startups.