SYSTEMS ENGINEERING PROJECT REPORT

Container Runtime & Orchestration Platform

Linux Primitives • Custom Scheduler • Service Mesh • mTLS

Youdahe Asfaw

Computer Science • Gustavus Adolphus College
Distributed Systems • Infrastructure • Reliability Engineering

youdaheasfaw@gmail.com • linkedin.com/in/youdaheasfaw • youdahe.dev

Abstract

This report presents the design of a complete container platform built entirely from first principles. Starting from raw Linux kernel primitives—namespaces, cgroups, OverlayFS, and seccomp-BPF—the project implements a container runtime, OCI-compatible image system, custom CNI networking plugin, multi-node distributed scheduler with bin-packing and affinity rules, rolling deployment strategies with automated rollback, and a service mesh providing mutual TLS, L7 load balancing, and circuit breaking. The platform demonstrates mastery of every abstraction layer that Kubernetes provides.

Table of Contents

1. System Architecture Overview

This project constructs an entire container platform from first principles, spanning every layer of abstraction from Linux kernel syscalls to a distributed orchestration control plane. The architecture is organized into four tiers: kernel primitives that provide isolation and resource control, a container runtime and image system built on those primitives, a control plane that manages scheduling and reconciliation across a cluster of nodes, and a networking and service mesh layer that handles discovery, routing, security, and resilience.

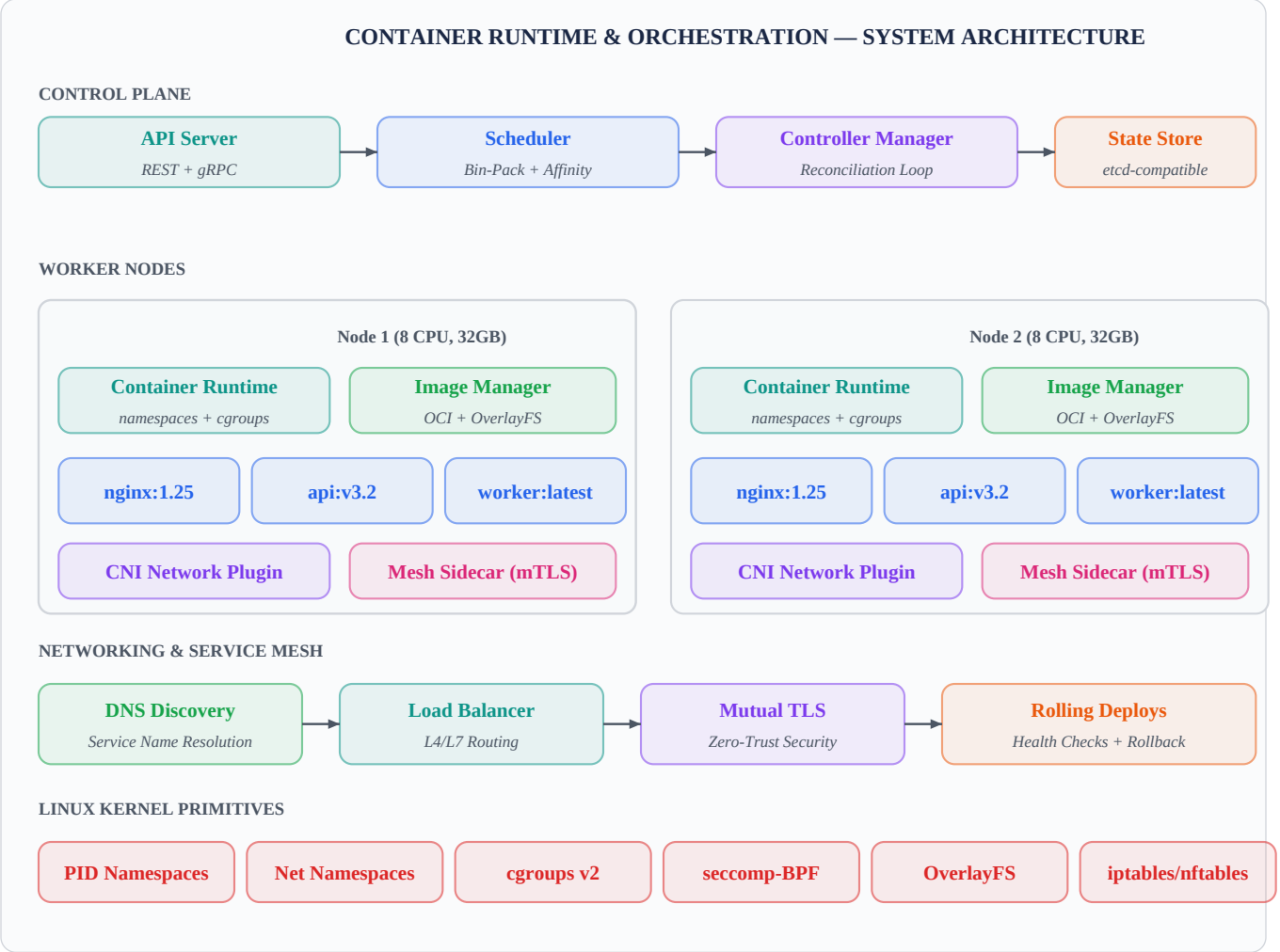

Figure 1 — Full platform architecture from kernel primitives to control plane.

2. Container Runtime (Linux Primitives)

2.1 Process Isolation with Namespaces

The runtime creates isolated execution environments using the clone(2) system call with namespace flags. Each container receives its own PID namespace (process 1 inside the container is the entrypoint), network namespace (its own network stack, interfaces, and routing table), mount namespace (isolated filesystem view),

UTS namespace (independent hostname), and user namespace (UID/GID mapping for rootless containers). The combination of these namespaces creates a process that believes it is running on its own machine, while actually sharing the host kernel.

2.2 Resource Control with cgroups v2

The unified cgroup v2 hierarchy enforces resource limits on each container. CPU limits are expressed as a quota/period ratio—for example, 100ms of CPU time per 100ms period equates to 1 full core. Memory limits trigger the OOM killer when exceeded, with configurable OOM score adjustments. I/O bandwidth limits use the BFQ scheduler to allocate proportional disk throughput. Process count limits prevent fork bombs. All limits are written to the cgroup filesystem and inherited by all child processes.

2.3 Filesystem Isolation with OverlayFS

Container filesystems use OverlayFS to layer a writable upper directory on top of one or more read-only lower directories (image layers). When a container writes to a file that exists in a lower layer, copy-on-write semantics copy the file to the upper layer before modification. Deletions are represented as whiteout files. This approach enables instant container startup (no file copying), efficient disk usage (shared base layers across containers), and clean teardown (delete the upper directory).

2.4 Security with seccomp-BPF

Each container runs with a seccomp-BPF filter that restricts which system calls it can execute. The default profile allows approximately 300 of the 400+ available syscalls, blocking dangerous operations like mount, reboot, and raw socket creation. Custom profiles can further restrict syscalls based on the container's actual needs, reducing the attack surface. Filters are loaded before the container entrypoint executes and cannot be relaxed once applied.

3. Image Format & Registry

Images use an OCI-compatible format consisting of a manifest (listing layer digests and configuration), a configuration blob (environment variables, entrypoint, exposed ports), and content-addressable layers (tar archives identified by their SHA-256 hash). The custom registry server stores layers with deduplication—if two images share a common base layer, it is stored and transferred only once. The build system supports declarative image definitions similar to Dockerfiles, with layer caching that skips unchanged layers during rebuilds.

4. Scheduler & Control Plane

The control plane follows a declarative, level-triggered model. Users submit desired state (e.g., "run 3 replicas of service X with 2 CPU and 4GB memory each"), and reconciliation loops continuously drive actual state toward desired state. The API server validates and persists desired state to an etcd-compatible store. The controller manager runs specialized controllers for each resource type—a replica controller ensures the correct number of instances, a node controller monitors node health, and a service controller manages load balancer configuration.

4.1 Scheduling Algorithm

The scheduler uses a multi-pass pipeline. The **filtering pass** eliminates nodes that cannot satisfy hard constraints: insufficient CPU or memory, missing capabilities, taint-based exclusions, or affinity rules requiring co-location with specific workloads. The **scoring pass** ranks remaining candidates using weighted criteria: resource balance (preferring nodes that would be most evenly utilized), data locality (preferring nodes with cached image layers), spread (distributing replicas across failure domains), and custom user-defined priorities. The highest-scoring node receives the binding.

5. Networking & CNI Plugin

The custom CNI plugin implements a full cluster networking model. Each node runs a virtual bridge, and every container receives a unique IP address from a cluster-wide CIDR allocation (e.g., 10.244.0.0/16, with each node owning a /24 subnet). A veth pair connects each container's network namespace to the node bridge. Inter-node traffic uses VXLAN encapsulation—packets destined for containers on other nodes are wrapped in UDP and sent to the target node, which decapsulates and delivers them to the correct container. Alternatively, in environments supporting it, direct routing via BGP peering eliminates encapsulation overhead.

A built-in DNS server resolves service names to the current set of container IPs backing that service. DNS records are updated in real-time via watch-based notifications from the control plane, ensuring that service discovery reflects the latest state within seconds of changes.

6. Service Mesh

The service mesh operates as a sidecar proxy co-located with each container. All inbound and outbound traffic is transparently redirected through the sidecar via iptables rules. The sidecar handles **mutual TLS**—automatically provisioning and rotating X.509 certificates for each service, encrypting all inter-service communication, and verifying peer identity. No application code changes are required.

L7 load balancing distributes requests across healthy instances using configurable algorithms (round-robin, least-connections, consistent hashing). **Circuit breaking** monitors error rates per upstream service; when failures exceed a threshold, the circuit opens and requests are immediately rejected with a fallback response, preventing cascade failures. **Distributed tracing** context is propagated through all proxies, enabling full request-path visualization without application instrumentation.

7. Deployment Strategies

Rolling deployments gradually replace old containers with new ones, parameterized by max surge (how many extra containers can exist during the rollout) and max unavailable (how many can be down simultaneously). Each new container must pass both liveness probes (is the process running?) and readiness probes (is it ready to serve traffic?) before the rollout proceeds. If probes fail beyond a configurable threshold, the deployment automatically rolls back to the previous version.

Blue-green deployments maintain two complete environments. Traffic is routed entirely to the active (blue) environment while the new version is deployed to the standby (green) environment. Once validated, traffic is switched atomically. **Canary deployments** route a configurable percentage of traffic to the new version, gradually increasing it as metrics confirm healthy behavior.

8. Observability & Monitoring

Every component exports Prometheus-compatible metrics: scheduler queue depth and binding latency, container CPU/memory utilization vs. limits, network throughput and packet drops per container, service mesh request latency and error rates per upstream. Grafana dashboards provide real-time cluster overview, per-node resource utilization, and per-service golden signals (latency, traffic, errors, saturation). Alerts fire on anomalous conditions such as sustained high error rates, approaching resource limits, or frequent container restarts.

9. Technology Stack Summary

Component	Technology / Approach
Container Runtime	Go/Rust using clone(2), cgroups v2, OverlayFS, seccomp-BPF
Image System	OCI-compatible format, content-addressable registry with dedup
Control Plane	Go API server, Raft-backed state store, reconciliation controllers
Scheduler	Multi-pass pipeline: filter + score + bind with pluggable policies
Networking	CNI plugin with VXLAN tunnels, iptables/nftables, BGP option
Service Mesh	Rust sidecar proxy: mTLS, L7 LB, circuit breaking, tracing
DNS	Watch-based service discovery with sub-second propagation
Deployments	Rolling, blue-green, canary with automated rollback
Observability	Prometheus + Grafana dashboards, request-level tracing
Languages	Go (control plane, runtime), Rust (sidecar proxy, networking)