draupnir — Manual

version 0.1.4

generated 2026-07-12

---

rendered by **nornir** · typst engine

Contents

Container Consolidation Plan

Container engine consolidation — jera → draupnir (status + deferred gaps)

Decided 2026-07-12. The nordisk consolidation law (`workspace_nordisk/.nornir/ consolidation-panels-maps-design.md` §1.1): the app→jera→draupnir chain must own provisioning in exactly ONE place. KVM was already unified (jera's `DraupnirVmBootRunner` boots via `draupnir::Boot`). Containers were NOT: jera had its own `BollardEngine` duplicating draupnir's `backend-oci` (both pin bollard 0.20). This doc records what landed and what is deliberately deferred.

What landed (this pass, additive, no regression)

draupnir (the one engine now lives here):

- `BootSpec` gained `cmd: Vec<String>` + `ports: Vec<u16>` (builders `with_cmd` / `with_port`) so the container backend reaches full parity with what jera's engine produced — entrypoint override + published ports, not just image + env.
- `container::Engine` now applies cmd + ports via a pure `create_body(image, env, cmd, ports)` builder (byte-for-byte the shape jera's `BollardEngine::create_body` emitted), streams container logs into a per-container buffer via a follow task, and exposes an exit-code-aware `container_state`.
- New public seam `container::ContainerControl` (`container_state` → `ContainerState{Running|Exited(i64)|Gone}`, `drain_logs`, `stop`) — the richer container control a job runner maps onto its own status/log model. The generic power `Lifecycle` collapses every non-running state to `Off` and so cannot tell a clean exit from a crash; `ContainerControl` preserves the distinction.

jera (delegates through draupnir, mirrors `vm.rs`):

- Feature `container-oci = ["draupnir/backend-oci"]`.
- `ContainerSpec::to_draupnir()` maps image + per-boot name + cmd/env/ports onto a `draupnir::BootSpec` (container mirror of `vm::BootSpec::to_draupnir`).
- `DraupnirContainerBootRunner<B>` (generic over the draupnir backend for daemon-free mock testing; production `B = draupnir::container::ContainerBoot`) boots via `draupnir::Boot` and drives the lifecycle over `ContainerControl`, mapping `ContainerState` → `BootStatus` exactly as `BollardProc` did.
- `StubContainerBootRunner` — honest not-wired failure when `container-oci` is off (mirror of `StubVmBootRunner`).
- `connect_default_runner()` returns `DraupnirContainerBootRunner` when `container-oci` is on; otherwise the pre-existing in-crate bollard path is unchanged. Delegation proven by a mock-`Boot` / `ContainerControl` test (no daemon, no feature needed).

DONE — the duplicate engines are removed; one engine, in draupnir

The consolidation is complete (2026-07-12). Every duplicate bollard container engine in the constellation has been deleted now that draupnir owns the one true engine; the removals were gated on the shared draupnir path being green + tested.

1. **jera's `run_container` repointed + `BollardEngine` DELETED (edda `e7c02cc`, jera 0.1.4).** `run_container` / `run_container_with_reports` now route through `draupnir::container::run_to_completion` (map `RunOutcome` → `ContainerOutcome`, forward the `Health` reporter). With the last reference gone, jera's `BollardEngine`, `BollardBootRunner`, `BollardProc`, the `ContainerEngine` trait, `EngineState`, and `ContainerRunSpec` were all deleted. `DraupnirContainerBootRunner`
   - `StubContainerBootRunner` are the only container runners left. A generic `run_container_on<B: Boot + ContainerControl>` keeps it mock-testable (no daemon).

`cargo test -p jera` GREEN on default / `--features container-oci` / `--no-default-features` (39 each).

2. **Default feature flipped OFF bollard.** jera now ships `default = []` (lean std build, no engine linked); the live OCI engine is opt-in behind `container-oci = ["draupnir/backend-oci"]`. The `bollard` / `futures` deps and the `bollard` feature were dropped. A single build has ONE engine (draupnir's), never two. `nornir-jobs` forwards `container-oci` to jera.
3. **nornir's THIRD copy repointed (nornir repo).** `nornir/src/viz/test_boot.rs` dropped its own independent `BollardEngine` / `BollardBootRunner` / `ContainerEngine` / `EngineState` / `ContainerRunSpec` for a `DraupnirContainerBootRunner<B>` that maps a `ContainerBuild` → `draupnir::BootSpec` and drives the lifecycle over draupnir's `Boot` + `ContainerControl`. nornir's direct `dep:bollard` is gone; the feature `test-boot-container-bollard` now forwards to `draupnir/backend-oci` (name kept for back-compat). The podman-subprocess `ContainerBootRunner` stays as the no-socket fallback. `cargo test -p nornir` (default) GREEN; the container tests are driven by a mock draupnir backend (no daemon).

Constellation grep — one container RUN engine (2026-07-12)

`grep -r 'bollard::'` across the constellation source now shows the container run/lifecycle engine in exactly ONE place — draupnir. The other two bollard users are **not** container run engines and are deliberately left:

- **`nornir/crates/nornir-build-thing`** (**`BollardImageBuilder`**) — an image *builder* (`build_image` over the REST API), a different concern from running a container. Not a duplicate of draupnir's run engine; out of scope.
- **`korp/src/demo_drive.rs`** — korp's own bollard bring-up of the demo FalkorDB (and Spark helper) containers. This IS a run-engine duplicate, but it belongs to **korp's own refactor slot** (main-only; not touched here). Follow-up: fold korp's demo container bring-up onto `jera` / `draupnir` too (korp already consumes knut's now-draupnir-routed `SparkServer`). Logged for the korp owner.

Behavior notes (parity preserved)

- **Container name.** jera mints `jera-boot-<uuid>` for an unnamed spec; `to_draupnir` passes that as the draupnir spec name, and draupnir prefixes `draupnir-`, so the delegated container is `draupnir-jera-boot-<uuid>`. Unique and kill-addressable (poll/kill use `Machine.id`); only the external name string differs from the in-crate path. No functional change.
- **poll / logs / exit codes / kill semantics** are mapped identically to the old `BollardProc` (`Exited(0)` → `BootedOk`, non-zero→ `Failed(code)`, `Gone` → `Failed` unless killed; `Health` Starting→Alive→Gone/Failed).
- No mounts/healthcheck/labels were set by either engine, so nothing to reconcile there.

Design

draupnir — design

Draupnir is the low-level boot / provisioning library of the nordisk constellation. It fires up a runtime instance from one [`BootSpec`] across three backends and drives its power lifecycle. Named for **Draupnir**, Odin's gold ring that drips eight identical copies of itself every ninth night — here a *light nod*: booting a **fleet of identical machines from one ISO** is the natural extension, not the core. The core is **one library, three boot backends**.

Charter

- **One `Boot` trait, three backends** — Draupnir's own code is the thin unifying seam; each backend *delegates* to an existing engine (Brokkr's minimal-dependency law: reuse, don't reinvent).
  - **KVM** → drives tunnr's `tunnr_vm::boot_test(BootSpec) -> BootHandle` primitive. Draupnir does **not** reimplement VM boot.
  - **Container** → an OCI runtime crate (podman/Docker REST).
  - **Redfish** → a BMC's out-of-band Redfish REST API to boot **bare metal**: insert a virtual-media ISO, set the one-time boot override, power on. This is the capability Draupnir uniquely owns.
- **Honest stubs.** The trait wiring + the pure bookkeeping (spec validation, the fleet fan-out) are real and unit-tested; every unimplemented backend internal is an honest `todo!("draupnir: …")` — never a fake-green stub or a sentinel string.
- **Lean by default.** A plain `cargo build` is **pure std, zero dependencies**. Each live backend sits behind a feature; a consumer pulls only what it drives.

Ownership — one home per capability (the no-duplication rule)

Draupnir is the **shared low-level boot lib**. Two high-level consumers depend on it **directly**; neither re-implements boot code. Draupnir knows nothing about jobs or services — it only fires up and controls instances.

Component	Layer	Owns	Depends on
Draupnir	low-level boot lib	the `Boot` / `Lifecycle` / `VirtualMedia` traits, `BootSpec` / `Machine` types, spec validation + fleet fan-out, and the **three backend adapters** (kvm/container/redfish)	tunnr (KVM); an OCI runtime crate; a Redfish client crate
tunnr	KVM boot backend	the QEMU + OVMF (UEFI) + KVM boot primitive (`BootSpec` / `boot_test` / `BootHandle`)	— (Draupnir's KVM backend drives it)
jera (edda job handler)	thin job policy	*job* semantics only — a job is `process | VM | container` ; the durable job ledger	**Draupnir** (directly, for VM/container job instances)
Skidbladnir	high-level service/airgap layer	service/systemd lifecycle, airgap pack/move/unfold, orchestration	**Draupnir** (directly, for service instances)

Call direction — two independent consumers, one shared engine:

```
jera ──────┐
           ├──▶ Draupnir ──▶ tunnr          (KVM VM)
```

```
Skidbladnir ──┘              ├─▶ OCI runtime      (container / redis service)
                             └─▶ Redfish BMC      (bare-metal virtual-media)
```

No-duplication rule. Instance *booting* lives in exactly one place: Draupnir. jera does not shell out to QEMU; Skidbladnir does not carry its own KVM/container drivers; neither reimplements the other's boot path. tunnr is the sole KVM boot primitive, reached only *through* Draupnir's KVM backend.

> **This revises the earlier "Skidbladnir is the KVM/container mechanism master" decision.** Skidbladnir now **delegates instance-firing to Draupnir** and becomes a *consumer* of it (service/systemd, airgap, orchestration stay in Skidbladnir). jera likewise depends on Draupnir directly rather than re-rolling VM/container boot.

Types & traits (the real core)

- `enum Backend { Kvm, Container, Redfish }` — selects the driver.
- `enum ImageSource { KernelRootfs{kernel,rootfs}, Disk, OciImage, Iso }` — the bootable payload; `suits(Backend)` gates which backend can boot which source.
- `struct BootSpec { name, backend, image, mem_mb, cores, cmdline, env, bmc }` — one self-contained request; `validate()` rejects mismatches (ISO to KVM; a Redfish spec with no BMC; a non-Redfish spec carrying a BMC) **before** any backend is touched.
- `struct BmcEndpoint { host, username, system_id }` — the out-of-band Redfish endpoint. **No credential field** — the secret is supplied at drive time, so a spec never carries one.
- `trait Boot { fn boot(&self, &BootSpec) -> Result<Machine>; }`
- `trait Lifecycle { power_on / power_off / status }`
- `trait VirtualMedia { insert_media / eject_media / set_boot_override }` — Redfish only.
- `fn plan_fleet(&BootSpec, n) -> Vec<BootSpec>` — the ring drips N identically- payloaded, distinctly-named specs (fleet-from-one-ISO).

Dependencies (reuse, don't reinvent — Brokkr's minimal-dependency law)

Draupnir's own code is the thin `Boot` glue; each backend delegates to an existing crate. Chosen (and why):

- **`tunnr-vm`** (path/optional, feature `backend-tunnr`) — the KVM boot backend *is* tunnr's primitive; wiring it as an optional path dep (interim until tunnr publishes) is exactly how `jera` does it, so there is one KVM boot path in the constellation, not two.
- **OCI runtime** — **`bollard`** (planned, feature `backend-oci`) — the async podman/Docker REST client `jera` already uses for its zero-shell container path; reusing it keeps a single container engine rather than a second bespoke one.
- **Redfish** — **`redfish-codegen`** (a generated, typed Rust Redfish client), or a thin **`reqwest`** wrapper over the handful of endpoints we need (`Managers/{id}/VirtualMedia` insert/eject, `Systems/{id}` `Boot` override, `ComputerSystem.Reset`) if the generated client's surface is heavier than the three actions warrant (planned, feature `backend-redfish`). Either way Draupnir does not hand-roll the Redfish protocol.
- **znippy** (nordisk, if/when an image/ISO content-cache is needed) — reuse Skidbladnir's content-addressed archive rather than a new one.

The network-touching backends (OCI, Redfish) sit behind OFF-by-default features so the default build stays pure-std and offline-green; only their justified crates are pulled when a consumer enables them.

Extraction plan — moving instance-firing out of Skidbladnir (staged)

Instance-firing code currently lives in **`Skidbladnir/src/machine.rs`** (behind its `machine` / `kvm` / `container` features): the `MachineController` trait with `kvm::KvmController` (qemu/KVM + cloud-init NoCloud seed) and `container::ContainerController` (podman/docker), plus `MachineSpec` / `Handle` / `ImageCache`. That is exactly Draupnir's remit. **Do not rip it now** — the scaffold is created first; the move is staged:

1. **Land Draupnir** (this repo) as the `Boot` / `Lifecycle` seam + backend adapters. *(done — this scaffold.)*
2. **Wire the KVM backend** to `tunnr_vm::boot_test` under `backend-tunnr` (signature already mapped in `src/kvm.rs`); port Skidbladnir's cloud-init NoCloud seed authoring into the KVM adapter.
3. **Wire the container backend** to `bollard` under `backend-oci`; fold in `ContainerController`'s start/stop/exec/wait_ready readiness model.
4. **Add the Redfish backend** (`backend-redfish`) — net-new capability, no Skidbladnir source to move.
5. **Repoint Skidbladnir** (and jera) at Draupnir: delete `machine.rs`'s duplicate drivers, keep only the high-level service/airgap logic that *calls* Draupnir. Skidbladnir's `ImageCache` either moves to Draupnir or is replaced by the znippy-backed image store.
6. **Repoint jera**'s `vm` / `container` seams at Draupnir instead of its own `tunnr-vm` path dep, so jera keeps only job policy.

Steps 5–6 are the actual code removal and happen only after Draupnir's backends are real (green under their features) — never a half-landed swap.

Why a separate repo

Booting/provisioning is a distinct, versionable capability with two independent consumers (jera, Skidbladnir). Its own leaf workspace lets both depend on one lean crate without dragging each other's dep trees, and lets the boot engine evolve on its own cadence — every backend a unit-testable Rust seam, not an opaque script.

Docs

This `.nornir/` is the **source** of the repo's docs (crisp holger-pattern): `design.md` (this file) · `guide.md` (usage) · `for-idiots.md` (the beginner walk-through).

draupnir — guide

Draupnir fires up a runtime instance from one `BootSpec` across three backends — KVM (via tunnr), OCI container, and Redfish bare-metal — and drives its power lifecycle. This is the usage walk-through; the architecture is in `design.md`.

Add it

```
[dependencies]
draupnir = { version = "0.1" }                # pure-std default: types + traits + validation
# draupnir = { version = "0.1", features = ["backend-tunnr"] }   # live KVM boot via tunnr
```

The default build has **no dependencies**. Each live backend is a feature:

feature	backend	delegates to
(default)	—	the trait surface + `BootSpec` / `Machine` + validation + `plan_fleet`
`backend-tunnr`	KVM VM	`tunnr_vm::boot_test`
`backend-oci` *(planned)*	OCI container	`bollard`
`backend-redfish` *(planned)*	bare metal	Redfish REST client

Describe an instance

```
use draupnir::{BootSpec, BmcEndpoint};

// A KVM appliance (kernel + rootfs) — booted through tunnr.
let vm = BootSpec::kvm_kernel_rootfs("wg-appliance", "/boot/bzImage", "/img/rootfs.cpio.gz");

// A redis service instance — an OCI container.
let redis = BootSpec::container("cache", "docker.io/library/redis:7")
    .with_env("REDIS_ARGS", "--save 60 1000");

// A bare-metal node — a Redfish virtual-media ISO boot.
let node = BootSpec::redfish_iso(
    "node-42",
    "/images/rhel-9.iso",
    BmcEndpoint {
        host: "https://bmc-42.dc.example".into(),
        username: "admin".into(),
        system_id: "System.Embedded.1".into(),
    },
);

// Reject an inconsistent spec before touching a backend.
vm.validate().unwrap();
node.validate().unwrap();
```

Fire it up

```
use draupnir::{Boot, kvm::KvmBoot, container::ContainerBoot, redfish::RedfishBoot};
```

```
let machine = KvmBoot::new().boot(&vm)?;          // feature backend-tunnr
let cache   = ContainerBoot::new().boot(&redis)?; // feature backend-oci
let bare    = RedfishBoot::new().boot(&node)?;    // feature backend-redfish
```

Each `boot()` returns a `Machine { id, spec_name, backend, power }`. Drive it with the `Lifecycle` trait (`power_on` / `power_off` / `status`). For Redfish, the `VirtualMedia` trait exposes the raw steps (`insert_media` / `set_boot_override` / `eject_media`) that `boot()` sequences for you.

Without its feature, a backend's `boot()` returns `Err(Error::Unsupported(..))` telling you which feature to enable — it never pretends to boot.

Boot a fleet from one ISO (the ring)

```
use draupnir::plan_fleet;

// One ISO -> eight identically-payloaded, distinctly-named specs (node-1 … node-8).
for member in plan_fleet(&node, 8) {
    RedfishBoot::new().boot(&member)?;
}
```

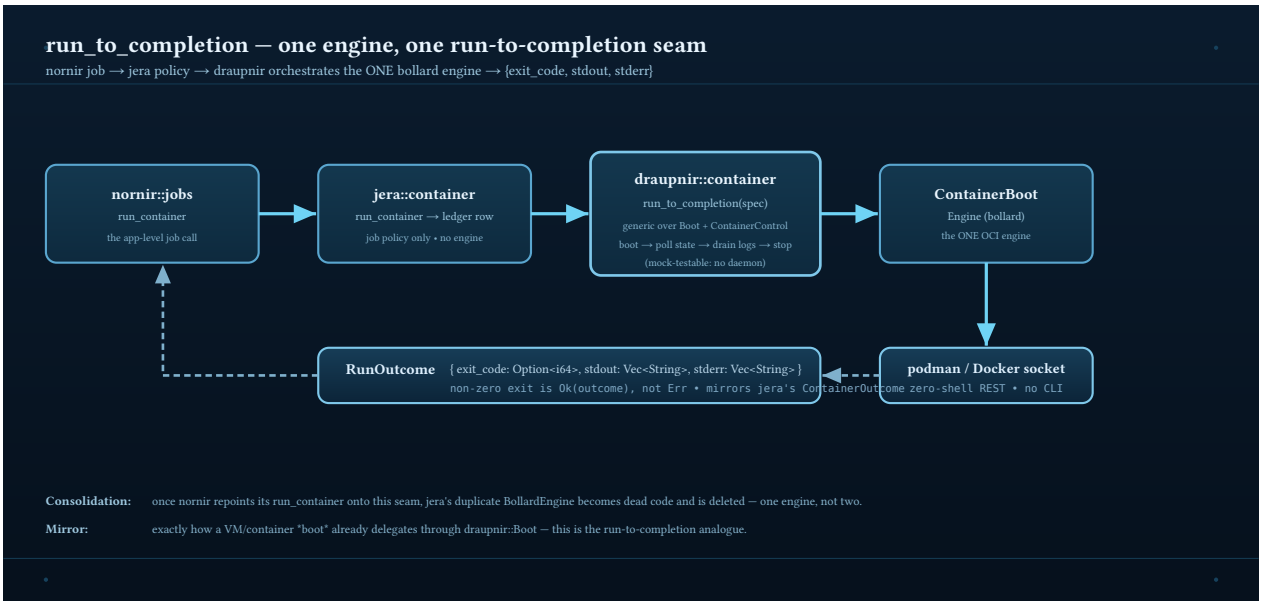
Where it sits

jera (job handler) and Skidbladnir (service/airgap) both depend on Draupnir directly and call down into it — neither reimplements boot. Draupnir drives tunnr for the KVM backend. See `design.md` for the ownership table.

Run To Completion Design

draupnir `run_to_completion` — the run-to-completion container seam

Added draupnir 0.1.4 (additive). This is the missing seam that completes the container-engine consolidation (`.nornir/container-consolidation-plan.md`): a single call that fires up a container, waits for it to exit, collects its logs, and returns an exit-code-aware result — the run-to-completion analogue of how a VM / container *boot* already delegates through `draupnir::Boot`.



Why it exists

The chain is `app -> jera (job) -> draupnir (provision)`. KVM and container *boot* were already unified: jera's `DraupnirVmBootRunner` / `DraupnirContainerBootRunner` drive `draupnir::Boot` + `ContainerControl`, one engine. But jera kept a **second, duplicate** bollard engine (`BollardEngine`) alive for exactly one reason: its `run_container` is a **run-to-completion** job (create -> start -> wait for exit -> capture stdout/stderr + exit code -> remove) and draupnir offered only *boot* + *lifecycle*, never that one-shot. `nornir::jobs::run_container` consumes jera's engine, so the duplicate could not be removed.

`run_to_completion` closes that gap. jera's `run_container` (and `nornir`'s above it) can repoint onto this seam, after which `BollardEngine` / `BollardBootRunner` / the `ContainerEngine` trait become dead code and are deleted — **one engine, not two.** (The repoint itself is jera/nornir work, owned by those repos' agents — this pass only provides + documents the draupnir API.)

The API (the only design — no alternatives)

```
pub struct RunOutcome {
    pub exit_code: Option<i64>,   // Some(0) = clean; None = gone before a code was read
    pub stdout: Vec<String>,      // captured stdout lines, in order
    pub stderr: Vec<String>,      // captured stderr lines, in order
}

pub struct RunOptions {
    pub timeout: Option<Duration>,  // None = wait forever (jera's wait_container
parity)
    pub poll_interval: Duration,    // default 200ms
}
```

```
// The generic, mock-testable seam — drives a live ContainerBoot in production
// AND a mock in a unit test with no daemon.
pub fn run_to_completion<B>(backend: &B, spec: &BootSpec, opts: &RunOptions) ->
Result<RunOutcome>
where B: Boot + ContainerControl;

// The ergonomic production entry point (forwards to the free fn with `self`).
impl ContainerBoot {
    pub fn run_to_completion(&self, spec: &BootSpec, opts: &RunOptions) ->
Result<RunOutcome>;
}
```

Design decisions, and why:

- **Built on the always-compiled `Boot` + `ContainerControl` seam, generic over the backend.** No new bollard code, no reach into `Engine`. That is what makes it **mock-testable with no daemon** — a scripted backend transitions `Running -> Running -> Exited(code)` and hands out log batches; the driver proves `start -> wait -> exit-code -> logs` end to end. `ContainerBoot::run_to_completion` is a thin production convenience over the same free function.
- **A non-zero exit is `Ok`, not `Err`.** The *container* failed, the *call* succeeded — inspect `exit_code`. Only a boot/connect failure or a `timeout` returns `Err`. This is byte-for-byte jera's `run_container` contract, so the repoint is behaviour-preserving.
- **`RunOutcome` mirrors jera's `ContainerOutcome` field-for-field** (`exit_code: Option<i64>`, plus stdout/stderr) so jera maps one onto the other with no loss. draupnir's type carries **no serde** (the lean pure-std default is preserved); jera owns the serde-serialised ledger shape.
- **Split stdout/stderr preserved additively.** The follow-task log buffer now tags each line `(is_stderr, line)`. `ContainerControl::drain_logs` still returns the combined ordered stream (unchanged public behaviour); a new **defaulted** `drain_logs_split(&Machine) -> (Vec<String>, Vec<String>)` returns the two streams apart. The default routes every combined line to stdout, so every existing `ContainerControl` impl (incl. jera's mock) keeps compiling untouched; `ContainerBoot` overrides it to surface the real tag.
- **Timeout is opt-in.** `None` waits indefinitely (parity with jera's blocking `wait_container`); `Some(d)` stops + removes the container and returns a clear `Error::Backend` naming the instance — no leak, no fake success.

Tests (all no-daemon)

`src/container.rs`:

- `run_to_completion_starts_waits_collects_split_logs_and_exit_code` — the full path: booted the spec, polled to `Exited(0)`, logs split across ticks + a final post-exit flush, container removed.
- `run_to_completion_surfaces_a_nonzero_exit_as_ok_not_err` — `Exited(137)` comes back as `Ok(RunOutcome{exit_code: Some(137)})`.
- `run_to_completion_reports_none_when_the_container_is_gone` — `Gone` -> `None` (never a fake 0), still removed.
- `run_to_completion_times_out_with_a_clear_error_and_stops_the_container`.
- `run_to_completion_propagates_a_boot_failure_without_polling`.
- `default_drain_logs_split_routes_combined_logs_to_stdout` — the trait default.

Perf / bench

N/A — no bench added, by design. `run_to_completion` is a poll loop gated on container start/exit + log-stream IO; there is no compute hot path to guard, and a real measurement would need a live podman socket on the bencher (Loki), which it does not have. Forcing a bench here would measure daemon latency, not draupnir. Noted in `bencher_agents_problems.md`.