ChatGPT

Container Runtimes and Orchestration for Offline/ Edge Environments

Introduction

Deploying containers in a self-reliant, offline-first architecture requires careful choice of containerization and orchestration tools. The goal is to start simple on a single node for development, then smoothly scale to a clustered setup as needed – all while staying open-source, resource-efficient, and not dependent on cloud services. The following research presents a comparison of several container runtimes and orchestration platforms that meet these criteria. We focus on options that are self-hosted (no managed cloud dependency), support air-gapped operation, can run on modest edge hardware, and integrate well with a typical DevOps stack (including services like PostgreSQL, Redis, Neo4j, Qdrant, RabbitMQ, MinIO, etc.). We emphasize strong community support, declarative configuration, low overhead, and integration with monitoring/logging.

Below is a comparison table of **six** suitable platforms (covering container runtimes and orchestrators), followed by detailed pros and cons for each. We then provide recommendations for initial single-node deployment, a roadmap to clustering, and notes on licensing and ecosystem considerations.

Comparison of Container Runtimes & Orchestrators

Platform	Type (License)	Single-Node Deployment	Path to Multi-Node Cluster	Resource Footprint	Offline-First Support	Ecosyst Suppor
Docker (Engine & Swarm)	Container runtime + basic orchestrator (Docker CE: Apache 2.0; Docker Desktop: Proprietary for large orgs) [1]	Very mature for single-node (Docker Engine and Compose are widely used). Easy installation on Linux/ Windows/ Mac; huge library of container images available. [2]	Built-in Swarm mode for clustering (simple setup, uses Docker CLI). Swarm provides multi-node orchestration, though development has stagnated [3]. Migration to Kubernetes later is possible but requires rewriting configs or using tools like Kompose.	Moderate overhead (Docker daemon consumes memory/ CPU) [2]. Suitable for most servers/ PCs; heavier than Podman due to always-on daemon.	Fully offline-capable. No internet needed if images are preloaded or pulled from a local registry. Docker supports air-gapped installs by loading images from tar or hosting a private registry.	Enormo commu and too Many th party to (e.g. Po watchto and De workflo suppor Docker. Howeve Swarm' ecosyst limited largely [3]. Do popular ensures experti availabi

Platform	Type (License)	Single-Node Deployment	Path to Multi-Node Cluster	Resource Footprint	Offline-First Support	Ecosyst Suppor
Podman	Container runtime (daemonless) (Apache 2.0, open-source)	Drop-in replacement for Docker on one node. Supports `podman run` and `podman-compose` for multi-container apps. Rootless mode enhances safety on single host. [4]	No built-in multi-node orchestrator. Pods can be run on each host, but clustering requires external tooling. Podman can generate Kubernetes YAML to ease later migration [5]. Often used in development, then moved to Kubernetes for production.	Lightweight – no central daemon, so idle overhead is low [6]. Memory usage generally less than Docker's. Efficient on edge devices.	Fully offline-capable. Podman does not phone home; works with local image registries or tarball-loaded images. All management is local CLI, which works offline.	Strong growing commu (led by Hat). Compa with Do CLI and images most ecosyst tools ar images Lacks a clusteri ecosyst (relies o Kubern system orchest

Platform	Type (License)	Single-Node Deployment	Path to Multi-Node Cluster	Resource Footprint	Offline-First Support	Ecosyst Suppor
containerd + nerdctl	Container runtime (OCI engine) (Apache 2.0, CNCF project)	Usable on a single node via CLI (e.g. using `nerdctl` which provides Docker-like commands [7]). Typically runs in the background on a node, managed by higher-level tools.	Not an orchestrator by itself. Designed to be embedded in cluster systems (it's the default runtime under Kubernetes, K3s, etc.). Multi-node clustering is achieved by using an orchestrator (K8s, Nomad) that leverages containerd on each node [8].	Very low overhead (streamlined, headless engine) [9]. Excellent performance and minimal resource use, ideal for resource-constrained hardware.	Fully offline-capable. Being a low-level runtime, it has no external dependencies. Images can be loaded from disk or a private registry (via nerdctl or CRI tools).	Widesp adoptic **under t hood** (u Docker, Kubern cloud provide Howeve direct u facing ecosyst smaller contain usually controll other to Fewer u friendly feature of-the-b built-in UI, com etc.).

Platform	Type (License)	Single-Node Deployment	Path to Multi-Node Cluster	Resource Footprint	Offline-First Support	Ecosyst Suppor
Kubernetes (vanilla)	Container orchestrator (full-featured) (Apache 2.0, CNCF project)	Can run single-node (e.g. Minikube, kind, or a single-node cluster via kubeadm). Typically more complex to set up and overkill for one node, but doable for testing.	**Built for clustering** – handles multi-node clusters natively. Scales to hundreds or thousands of nodes with high resiliency. Many distributions (kubeadm, etc.) exist for on-prem clusters. High learning curve and overhead in exchange for power.	High resource usage (multiple control-plane components: API server, etcd, scheduler, controllers) [10]. Needs more memory/CPU than lighter options (often several GB for control plane). On small edge devices it can be heavy.	Can run offline with planning. All components are self-hosted; cluster doesn't require internet. Air-gapped deployment entails preloading container images for K8s components and apps, and possibly hosting an internal container registry. (Many organizations run Kubernetes completely on-prem/ offline).	**Massiv ecosyst and commu** Kubern the de f standar thousa tools (H charts, Operat monito integra suppor Strong commu adoptic means term su and innovat but also comple manag

Platform	Type (License)	Single-Node Deployment	Path to Multi-Node Cluster	Resource Footprint	Offline-First Support	Ecosyst Suppor
K3s (Lightweight K8s)	Lightweight Kubernetes distribution (Apache 2.0, CNCF sandbox)	Easy single-node install (one binary ~50-100 MB) [13]. Designed for edge/dev use cases. Runs a full Kubernetes API but uses an embedded datastore (SQLite by default) for simplicity [14]. Great for initial development if you want to use K8s API without the bulk.	Supports multi-node clustering seamlessly. Additional nodes (agents) can join the K3s server; for HA you can use external DB or K3s's etcd mode. Scales to modest cluster sizes (hundreds of nodes) and is used in many edge deployments [15]. Upgrading to full Kubernetes (if ever needed) is straightforward since K3s is certified Kubernetes.	Much smaller footprint than standard K8s – roughly half the memory of equivalent upstream install [16]. Single binary includes control plane and containerd, trimming non-essentials (no cloud provider extras, uses SQLite). Still heavier than a simple Docker engine, but optimized for IoT/edge.	**Yes – designed for air-gapped and edge use.** Official docs provide an air-gap install guide [17]; you can preload K3s and its images on offline systems. K3s even has an embedded registry mirror option for cluster nodes to share images without external registry [18].	Good ecosyst and gro K3s is p for IoT, and dev clusters support standar Kubern tooling (kubect Helm, e [20]. Commu strong (origina from Ra SUSE). M Helm ch and monito solution work or Kubern will wor K3s.

Platform	Type (License)	Single-Node Deployment	Path to Multi-Node Cluster	Resource Footprint	Offline-First Support	Ecosyst Suppor
Nomad (HashiCorp)	General-purpose orchestrator (containers *and* other workloads) (Source-available: HashiCorp BSL 1.1 [21])	Simple single binary for server and client agents [22] – easy to run on one node (even a dev laptop). Nomad in dev mode can schedule containers similarly to Docker Compose. Great for initial testing without complex setup.	Built-in clustering and federation. Nomad servers form a cluster via gossip/Raft; add more agents to scale. Very easy to expand – just start Nomad on new nodes and join the cluster. Handles heterogeneous workloads (can orchestrate VMs, binaries, as well as Docker/ Podman containers) [23] . Lacks the rich built-in services of K8s (no native ingress or auto service mesh), so you integrate external tools (Consul, etc.) for those as you scale.	**Low overhead:** Nomad is extremely lightweight in CPU and memory terms. One study found Nomad to use the least CPU/RAM among edge orchestrators, more efficient than even K3s [24] . Runs on modest hardware (even Raspberry Pis) easily.	**Offline-ready:** Nomad has no external dependencies by default (optional integration with Consul/ Vault for service discovery or secrets can also be self-hosted). Uses local binaries; no need for internet after installation. Container images can be pulled from a local registry or stored on the node. Nomad's simplicity makes it reliable in poor connectivity scenarios.	Smaller passion commu Fewer c shelf ad compar Kubern [12] . Ho Nomad support many p and int with HashiC ecosyst (Consul Vault). F monito Nomad expose metrics can be scraped Promet etc., bu won't fi many tu "Helm c or third operato in K8s. Nomad UI and simplici reduce operati burden finding commu support as easy with Kubern

Table: Overview of self-hosted container runtimes and orchestration platforms suitable for single-node and clustered deployments in offline or edge environments.

Pros and Cons of Each Option

Below we discuss each platform in detail, highlighting pros and cons with respect to resource usage, adoption, scalability, ease of setup, and offline operation.

Docker (Engine & Swarm Mode)

Pros: Docker is the classic container runtime – easy to install and use on a single machine, with **vast community adoption and tooling**. It runs on all major OS platforms, enabling a convenient dev/test experience. Thousands of pre-built container images on Docker Hub make it simple to set up services like PostgreSQL or Redis. Docker Compose offers a **declarative YAML** way to define multi-container apps on one node. When it comes time to scale, Docker's built-in **Swarm mode** can convert a single-engine setup into a multi-node cluster with just a few commands (`docker swarm init`), reusing the same container images and Compose files (as stack files). Operational overhead is low for small clusters – no external databases or complex config needed. Docker's design is proven in production, and the **OCI image standard** it popularized ensures portability (images will run under other runtimes if needed) [25]. Finally, the core Docker Engine (Community Edition) remains open-source (Apache 2.0) and free to use; only Docker's proprietary add-ons like Desktop have licensing constraints for large enterprises [1] [26].

Cons: The Docker Engine uses a central daemon process that must run as root, which introduces a single point of failure and some security risk (if compromised, it has root access) [27] [6]. This daemon also consumes additional memory/CPU even when container workloads are idle, making Docker slightly heavier than daemonless alternatives like Podman. In edge scenarios with very tight resources, this overhead can matter. Docker Swarm, while easy to use, has **limited features** compared to Kubernetes – for example, it lacks built-in robust RBAC (Role-Based Access Control) and its networking and volume plugins are less extensive. Swarm's development has largely stagnated: the project is "massively under-maintained" and effectively frozen, with few bug fixes and no new features in recent years [3]. This means using Swarm for long-term might carry some risk, as you can't rely on active community innovation or timely patches (Mirantis maintains the code, but their focus is on Kubernetes) [28]. If a bug arises, there may be no official fix forthcoming. Additionally, Docker's ecosystem historically assumes an internet connection (for pulling images from Docker Hub); in offline setups you'll need to manually load images or maintain a local registry mirror – which is entirely feasible, but an extra step. Lastly, **licensing changes** by Docker, Inc. in recent years introduced some uncertainty – Docker Desktop now requires a paid subscription for large companies (250+ employees) [29] [1], which pushed many users to explore open alternatives. While the open-source Docker Engine is unaffected by these fees, organizations concerned about Docker's direction may prefer to avoid potential future licensing complications.

Podman

Pros: Podman is a drop-in replacement for Docker's core functionality that addresses some of Docker's drawbacks. It is **daemonless and runs containers as regular processes**, which means no always-running root daemon consuming resources [6]. This yields a smaller footprint and potentially improved security – each container can run under a non-root user (rootless containers) by default, limiting damage from any single container breakout [4]. Podman's CLI is Docker-compatible; in many cases you can alias

`docker=podman` and use it seamlessly [30]. It also supports Docker Compose files (via the `podman-compose` tool or by generating Kubernetes pods), so your existing Docker workflows can carry over [31]. Because it's backed by Red Hat and the open-source community, Podman has **strong long-term support** and integrations (Red Hat uses it in RHEL and OpenShift). Podman is particularly attractive in offline or edge scenarios: no daemon means that if the network is spotty or the device sleeps, there's no central process to get wedged – containers are simply child processes of your shell or system service. All actions (pulling images, starting containers) can be done offline with preloaded images. Podman's pods feature lets you group containers in a similar way to Kubernetes pods, which is useful for running things like app+sidecar on a single node. A big advantage for future growth is Podman's **Kubernetes integration** – it can **generate Kubernetes YAML** from running containers or pods [5]. This means you can develop your stack with Podman on one machine, then export a manifest and deploy that onto a Kubernetes cluster later, easing migration. Overall, Podman offers a more secure and slightly leaner alternative to Docker without losing compatibility [6].

Cons: Podman on its own is **not a full orchestration solution**. It is great for single-node container management, but it doesn't have an equivalent to Docker Swarm for multi-node clustering. If you need to orchestrate containers across multiple machines, you'll have to introduce another layer (for example, use Kubernetes itself, or use Ansible/scripts to coordinate Podman on each node, which is manual). So the "clear path" to clustering with Podman typically means migrating to Kubernetes (or another orchestrator) later, rather than simply "switching on" clustering as with Swarm or K3s. This is a consideration for long-term scaling – you should be prepared to adopt Kubernetes or Nomad when moving beyond one node. Podman's **networking in rootless mode** can be slightly slower or less straightforward for complex setups (since it uses user namespacing and slirp4net for rootless networking) [32]. In most cases this isn't a major issue, but high-performance network workloads might need rootful mode or tuning. Another limitation is on non-Linux platforms: Docker Desktop provided a polished Mac/Windows experience, whereas Podman on Mac/Windows is newer and can require using a VM backend (Podman Desktop or Lima). By 2025 Podman's cross-platform support is improving, but some developers find Docker's tools still more mature on those OS. Additionally, certain Docker Compose extensions or swarm-specific features (like Docker's fluent overlay networking or secrets handling) may not directly carry over in Podman. The community for Podman, while solid, is smaller than Docker's; you may find fewer ready-made tutorials or examples (though this is quickly changing as Podman gains popularity). In summary, Podman's main downside is the lack of an integrated multi-node orchestration – it excels at single-node container management but assumes you'll layer something else on top for clustering.

containerd (with nerdctl)

Pros: `containerd` is the high-performance container runtime that underpins Docker and Kubernetes. Using containerd directly (with a CLI like `nerdctl`) gives you a very **minimal, efficient container engine** that does nothing beyond the basics – it pulls images, launches containers, and manages storage/network through plugins [9]. This lean design means a **small resource footprint** and less complexity. On a constrained edge device, running containerd alone (without Docker's extra features) can save memory and CPU. It's proven at scale – since containerd is **industry-standard and CNCF-backed**, it has been battle-tested in huge cloud environments [9]. All major cloud Kubernetes services (EKS, GKE, AKS, etc.) use containerd, so its stability and performance are top-notch. In an offline context, containerd has no external requirements; it doesn't try to talk to any cloud API. It will only pull images when instructed – so if you manage an offline registry or pre-load images, containerd is perfectly content offline. Another advantage is that containerd is **OCI-compliant** and thus works with any OCI images – you're not tied to Docker-specific

image formats or tools [33]. When planning a long-term architecture, containerd provides flexibility: you could start with containerd + nerdctl on a single node (for lightweight operations), and later **plug containerd into an orchestrator** like Kubernetes or Nomad without changing the underlying runtime on each node. This means your lower-level container runtime remains constant as you scale up. containerd's design also supports plugins for different snapshotters (storage drivers) or even alternative runtimes (like Kata containers for sandboxing), giving future options if needed [34].

Cons: On its own, containerd is **not very user-friendly**. It lacks the convenient UX of `docker` or `podman` CLI – in fact, containerd's native interface is a gRPC API, not a human CLI [7]. Using it typically requires additional tools: e.g. `ctr` (the low-level containerd CLI) or `nerdctl` (which emulates Docker commands). This is fine for advanced users but can be cumbersome for daily development. Essentially, containerd by itself is a building block, not a full solution – you would normally run containerd as part of a larger system (Docker uses it under the hood, Kubernetes uses it via the CRI interface). Thus, while containerd meets all the criteria (open source, offline, single-node capable), in practice you might use Docker/Podman on top of it for convenience until you have a proper orchestrator. Another consideration: containerd doesn't include higher-level features like docker-compose, volume management commands, or image build tools. You would need other tools for those (for example, Buildah/Podman for building images, or manual volume setup). So the **operational burden** is a bit higher if you go with "raw containerd" – you have to script or manage many things yourself that Docker would normally handle. Community-wise, you won't find many end-user guides for "just containerd"; documentation is geared toward integrators and developers of orchestration systems. So support is mainly via the Kubernetes or Docker communities depending on your usage. In summary, containerd is superb as an underlying component – extremely efficient and open – but **not a turnkey solution** for an admin or developer on its own. You will likely pair it with something else (Docker, Kubernetes, etc.), which somewhat overlaps with using those directly. Think of containerd as an *engine*, not the whole car.

Kubernetes (Full "Vanilla" K8s)

Pros: Kubernetes is the **most widely adopted container orchestration platform** in the industry [11]. Its biggest strength is the rich set of features for managing containers at scale: automated binpacking of workloads, robust self-healing (restarting failed containers, rescheduling on healthy nodes), horizontal scaling, service discovery, load balancing, rolling updates, and advanced network and security policies. It has support for **stateful applications** via StatefulSets and persistent volumes, which is crucial for running databases like PostgreSQL, Neo4j, or MinIO in containers. Kubernetes will let you define storage classes and dynamically provision volumes through a variety of plugins, making it possible to manage stateful services in a declarative way (on-prem, this could mean hooking into Ceph, NFS, or just using local disks). The **ecosystem support is unparalleled** – as Portainer's CEO notes, Kubernetes enjoys an ecosystem of "over 1000 tools" and virtually every cloud and infrastructure vendor offers integrations [12]. Need monitoring? Tools like Prometheus and Grafana have Kubernetes-ready deployment charts. Logging? EFK (Elasticsearch-Fluentd-Kibana) stacks are readily available. Continuous deployment? Helm and ArgoCD can declaratively manage your apps. This vast ecosystem means you're never alone in solving a problem – likely someone has written a controller, operator, or Helm chart for it. Kubernetes' declarative configuration (YAML manifests) aligns well with GitOps and automation. Another pro in the context of future needs: **portability and hybrid cloud**. Because Kubernetes is available as managed services (EKS, GKE, AKS) and on all environments (on-prem distributions, edge distributions), adopting Kubernetes early means down the line you could integrate with cloud easily if desired. Your skills and config will translate to cloud-managed K8s or vice versa, since it's a common standard. Kubernetes is also highly flexible: though it's focused on container

orchestration, it's extensible to do batch jobs, event-driven serverless (with add-ons), and more. In terms of being offline-first, Kubernetes clusters can run disconnected from the internet – all core components run locally on your nodes. You just need to host your own registry for images and internal update processes. Kubernetes does not *require* cloud services at all (cloud provider integration plugins are optional and can be turned off for on-prem). For long-term maintainability, Kubernetes' huge community and CNCF governance ensure it will be around and improving for years, with lots of documentation and experts available.

Cons: The power of Kubernetes comes at the cost of **complexity and resource overhead**. A full vanilla Kubernetes (e.g. deployed with kubeadm or similar) will run a control plane consisting of API server, etcd (a distributed key-value store), controller-manager, scheduler, and usually some addons (DNS, etc.). These components collectively can consume a couple of gigabytes of RAM and appreciable CPU even with no workloads – which on an edge device could be a large tax [10]. Kubernetes was originally designed for data centers, so even "small" clusters assume decent hardware on masters. For a single-node developer setup, running all this is often overkill compared to Docker or Nomad. **Ease of setup** is another issue: bootstrapping a Kubernetes cluster (without managed services) is non-trivial for newcomers. Tools like kubeadm simplify it, but you still have to configure certificates, networking (CNI plugins), and so on [35]. The learning curve for Kubernetes is steep; understanding its objects (Pods, Deployments, Services, Ingress, etc.) and configuring YAML manifests can be time-consuming for a small team, especially if they are new to container orchestration. In an offline scenario, the initial setup is further complicated by needing all images (for control plane and addons) downloaded beforehand – though guides exist, it's extra upfront work. Operationally, a Kubernetes cluster has a lot of moving parts – etcd needs care (for backup and consistency), certificate rotations, upgrades of the control plane, etc., which translates to **higher operational overhead** unless you have dedicated platform engineers or use a simpler Kubernetes distro. Troubleshooting can also be complex (e.g. debugging why a pod isn't scheduling might involve digging into events, controller logs, etc.). While Kubernetes excels at scaling *up*, it may not scale *down* as gracefully to tiny footprints. Another con is that many features assume reliable connectivity (between control plane and nodes); if you truly have intermittent connectivity between nodes (as can happen in edge networks), Kubernetes might struggle unless you use specialized variants like K3s or KubeEdge [36]. Finally, regarding **licensing/support**: Kubernetes itself is Apache 2.0 and free, but some distributions or enterprise support offerings cost money. Also, the community moves fast – frequent version releases mean you need to upgrade regularly (approximately every 12 months to stay within support window). In summary, Kubernetes is powerful but heavy – for a small initial deployment it can be "using a sledgehammer to crack a nut," and the added complexity could be a drawback if your team is not experienced with it.

K3s (Lightweight Kubernetes)

Pros: K3s was created to make Kubernetes **half the weight and easier to deploy**, especially in resource-constrained and remote environments [37] [38]. It packages all the Kubernetes control plane components into a single binary and strips out non-essentials. By default, it uses an embedded SQLite database instead of etcd for cluster state on a single server – massively simplifying setup (you don't need to run a separate datastore for a single-node or small cluster) [14]. The result is a **much smaller memory and CPU footprint**: K3s can run a controller + agent in as little as 512MB-1GB of RAM for light workloads, whereas upstream K8s might need 1.5-2GB+ just for control plane. This makes a difference on edge hardware like Raspberry Pis or Intel NUCs. In fact, K3s officially supports ARM architecture out of the box, making it ideal for IoT projects [39] [40]. Another pro is the **ease of installation** – a single command can install K3s on a node (it's essentially a curl-able script that drops the binary and configures it). This lends itself well to automated, offline install too (download the binary and run it). K3s has **built-in components** to provide a "batteries-

included" experience: it comes with Containerd as the runtime, Flannel for CNI networking, CoreDNS, and even an ingress controller (Traefik) and Helm Controller built-in [20]. This means out-of-the-box you have networking, DNS, and ingress ready to use without extra setup. It supports basic PersistentVolume provisioning on local disk by default for stateful apps [20]. Essentially, it's Kubernetes but with sane defaults for a small, single-site deployment. K3s still speaks the standard Kubernetes API, so you can use **kubectl, Helm, and any other ecosystem tool** with it – it's a certified conformant Kubernetes distro [41]. This is great for leveraging existing Kubernetes knowledge or tooling (e.g. you can apply the same YAML manifests on K3s that you would on a full cluster). The path to clustering is straightforward: you can join additional nodes as agents to the K3s server, and if you need HA for the server, you can run multiple K3s servers with an external etcd or the new embedded etcd HA mode. K3s's design specifically **targets offline and edge use** – Rancher (its creator) provides guides for air-gapped usage, including distributing an image tarball with all required container images for K3s and configuring a private registry [17] [42]. It even has an **embedded registry mirror** feature where one node can act as a registry for others if you don't have a dedicated registry server [18]. Community and vendor support for K3s is strong as well – it's a CNCF project with growing adoption in edge deployments, and many tools (like Portainer, k3sup, Rancher management server) specifically cater to managing K3s clusters.

Cons: Despite being slimmed down, K3s is still Kubernetes at its core – which means the overall complexity of Kubernetes hasn't vanished, it's just repackaged. You still need to understand Kubernetes concepts (pods, deployments, services) and deal with YAML manifests for anything beyond trivial deployments. While K3s reduces memory and CPU requirements, it **still requires a non-negligible amount of resources**; compared to a plain Docker or Nomad setup, K3s will likely use more memory and background CPU because it's running a full control plane (API server, controllers, etc.) [43]. For example, if you just want to run two containers on a single node, running them directly with Docker uses essentially no overhead beyond the containers themselves, whereas running them on K3s means you have the Kubernetes control plane overhead in addition. Another consideration is that by default K3s trades etcd for SQLite on single node – this is fine for development, but if you scale to a multi-node cluster and want HA master nodes, you'll likely switch to etcd (or an external DB via Kine), which adds back some complexity for operations (managing etcd cluster or external datastore). K3s's "batteries included" approach can be a con if you need to customize components: for instance, it includes Traefik ingress by default; if you prefer a different ingress controller, you'd need to disable the built-in one and install yours. Not a big deal, but something to note. The embedded components like Flannel for networking are simple but not as flexible/performance-optimized as some advanced CNI plugins – in an edge scenario this is usually fine, but if you needed, say, Calico for network policy or high-performance networking, you'd have to manually configure it on K3s (which it supports, but not out-of-the-box). Also, while K3s aims for simplicity, **debugging issues** can still require Kubernetes know-how – e.g. if pods aren't scheduling or a daemonset fails, you need to inspect logs/events just like normal K8s. In terms of ecosystem, K3s is compatible with most K8s tools, but extremely large-scale or enterprise-focused Kubernetes add-ons might assume a full Kubernetes installation (for example, some might assume a standard etcd or certain cloud integrations which K3s might not include). However, these cases are rare in the intended use scope of K3s. Finally, K3s being relatively new (though now a few years old and quite stable) means when it does have a bug, you rely on the open-source community (or Rancher/SUSE support if you have it) to address it; its release cycle is tied to Kubernetes releases, which is frequent (you'll need to update K3s regularly to keep pace with Kubernetes CVEs and fixes). In summary, **K3s' drawbacks are mostly those of Kubernetes itself** – complexity and overhead – albeit in a reduced form. If your team is not familiar with Kubernetes, there will be a learning curve to using K3s effectively for orchestrating your services.

Nomad

Pros: Nomad takes a markedly simpler approach to orchestration. It runs as a **single lightweight binary** for both server and client, making deployment and maintenance very easy (just a single binary to upgrade, no complex cluster bootstrap) [22] . In many ways, Nomad has a lower barrier to entry than Kubernetes: you can download Nomad, run `nomad agent -dev`, and have a usable single-node scheduler in seconds. Its configuration (HCL job files) is declarative but simpler to grasp than sprawling Kubernetes YAMLs. Nomad is **flexible in workload support** – it can orchestrate not just OCI containers (with drivers for Docker, Podman, etc. [23]) but also raw executables, JARs, VMs (via QEMU or Firecracker drivers), and more. This means if your environment has a mix of containerized apps and maybe some legacy services or even batch scripts, Nomad can schedule all of them in one cluster. You don't need to containerize everything to benefit from Nomad (though container workloads are first-class). The scheduling capabilities of Nomad are powerful (bin packing, spread, affinities) yet it eschews a lot of the complexity of Kubernetes. **Resource efficiency** is a major plus: as noted earlier, Nomad has been shown to use very little CPU and memory relative to Kubernetes – one comparative study found Nomad the most efficient in CPU/memory among edge orchestrators [24] . This means you can run the Nomad server process on something like a Raspberry Pi without much trouble, and each client agent is lightweight too (the client is basically just a task runner with minimal overhead). Nomad's design favors an **"operate in harsh conditions"** mentality: it doesn't require a separate datastore like etcd (Nomad servers internally use the Raft protocol for consensus, which is built-in), and it doesn't require a service mesh or overlay network to function (it relies on OS networking or Consul integration for service discovery when needed). Fewer moving parts can mean more reliability in intermittent connectivity scenarios. For an offline-first setup, Nomad is ideal – after the binary is installed, everything is self-contained. You can schedule container jobs referencing images by name; as long as those images are present on the host or in a private registry that the host can reach, Nomad will run them. If an image isn't present, Nomad (via the Docker/Podman driver) will try to pull it – so in offline mode you either pre-load images or configure the drivers to use a local registry only. Nomad's **operational simplicity** also translates to easier maintenance: upgrading the cluster is as easy as swapping binaries (Nomad agents can be rolling upgraded one by one). There is no complex migration of cluster state – state is light and stored in a few MBs of Raft data. The failure modes are easier to reason about (fewer components to coordinate). When it comes to integration, while Nomad's ecosystem is smaller, it **integrates well with HashiCorp's other tools**: Consul for service discovery and health checks, Vault for secrets. These can be very powerful in a closed environment – for example, you can have Nomad automatically pull credentials from Vault to inject into jobs, or use Consul service registry to have services find each other without hardcoding addresses. But note that Consul and Vault are optional; Nomad can use its internal features if those aren't present. Another pro: Nomad's web UI (and CLI) provide a straightforward view of jobs and allocations, which some find more approachable than Kubernetes dashboards and kubectl outputs. Nomad's permissive scheduling (allowing mixed workloads and not enforcing containerization) offers **flexibility for edge** cases.

Cons: The primary drawback of Nomad is **smaller adoption and ecosystem** compared to Kubernetes. As Portainer's analysis pointed out, Kubernetes has the lion's share of mindshare and third-party tools, whereas Nomad has relatively few [12] . This means if you need a certain extension or solution, you might not find an off-the-shelf Nomad equivalent easily. For instance, Kubernetes has dozens of operators for various databases that automate clustering/backup/etc.; Nomad doesn't have an "operator" pattern out-of-the-box (though you can achieve similar with scheduling and external scripts, it's more manual). If you require things like automatic horizontal pod autoscaling based on custom metrics, you'd have to DIY in Nomad, whereas in Kubernetes it's an API away. Nomad also lacks an embedded concept of ingress or load balancers – you'd typically use Consul or a separate load balancer in front. In practice, setting up Nomad for

complex apps might involve assembling multiple HashiCorp tools: e.g. Nomad + Consul + Vault + possibly Fabio or another ingress proxy. Each of those is individually simpler than the equivalent in Kubernetes, but it's more pieces to wire together (HashiCorp's philosophy is modular design rather than one monolithic system). Another con is that Nomad's **job definitions (HCL syntax)**, while easier to read, are less ubiquitous than Kubernetes YAML, so there's a smaller pool of examples on the internet. You might end up writing your own job specs from scratch for everything, whereas with Kubernetes you can often find a Helm chart or manifest for common software. However, Nomad is introducing a community repository of Nomad Packs (similar to Helm charts) to address this. In terms of scalability, Nomad can handle large clusters (it boasts usage at scale at some organizations), but the ecosystem for managing very large Nomad deployments (1000+ nodes) is less proven in the open than Kubernetes which has many large-scale success stories. Another consideration: **HashiCorp's license change** in 2023 – Nomad (like other HashiCorp tools) transitioned to the Business Source License (BSL) [21]. While this doesn't affect your ability to use Nomad freely (unless you plan to offer Nomad as a service commercially), it means Nomad is no longer "truly open source" in the eyes of some (BSL is source-available with a usage condition and eventual conversion to MPL in 4 years). This may or may not matter to you, but it caused some unease in the community. Nomad itself remains free to use for running your own cluster, though. Support-wise, HashiCorp provides enterprise support for Nomad, but if you're relying on community help, it's more niche – you might get fewer answers on forums just because fewer people use it relative to Kubernetes. Finally, while Nomad supports container workloads well, some Kubernetes-specific patterns don't have direct analogs (for example, Kubernetes pods co-locate containers with shared namespaces; Nomad's "task groups" are similar but not identical, and networking between tasks might require additional configuration). If your team has already learned Kubernetes, switching to Nomad would be a shift. Conversely, if you learn Nomad, that knowledge is somewhat less transferable to other environments (since most companies use Kubernetes). In summary, **Nomad trades off ecosystem breadth for simplicity**. It's fantastically easy to get started and operate, but you may have to build more yourself and accept that it's not the industry standard tool – which could impact hiring or community support in the long run.

Recommendations for Initial Single-Node Deployment

For initial development and testing on a single node, the priority should be simplicity and alignment with your team's skills. Based on the research, two approaches stand out:

- **Use Docker (or Podman) with Compose for quick wins:** If your team is comfortable with Docker, there is nothing wrong with starting on a single Docker Engine with Docker Compose to manage your services. This approach has minimal setup: you can write a `docker-compose.yml` that defines PostgreSQL, Redis, Neo4j, Qdrant, RabbitMQ, MinIO, etc., and bring them up with one command. Tools like **Borgmatic** (for backups) can run on the host or in a container and target the data volumes directly. This environment will closely resemble many development setups and is easy to iterate on. It's also entirely self-hosted and offline-friendly – you'd pull the necessary images once (or load them from files) and then you can operate without internet. If you prefer to avoid Docker's licensing nuances or root daemon, **Podman** is an excellent alternative at this stage; it can even directly run your Compose file with minimal changes. The advantage here is low complexity – there's no orchestration layer to learn. As Portainer's CEO advises, *"if your team is not sufficiently equipped to deal with the complexity of Kubernetes, then **standalone Docker hosts are a good way to start**"*^[44]. The last thing you want in early development is to be fighting your orchestration system rather than building your application. So starting with Docker/Podman gives you a stable, well-understood baseline.

- **Use a lightweight orchestrator in single-node mode (if you want to familiarize early):** If you anticipate moving to a cluster soon and have the bandwidth to learn a new tool, you could start by deploying something like **K3s or Nomad on a single node from the get-go**. This means instead of Compose, you write either Kubernetes manifests (for K3s) or Nomad job files to define your services. The benefit of this approach is that you'll iron out the configuration in the orchestrator environment from day one – no need to later translate a Compose setup into K8s or Nomad. K3s in single-node mode would let you use Kubernetes primitives for everything (perhaps overkill for very early dev, but it ensures what you deploy is ready to scale). Nomad in single-node dev mode is extremely easy to run and could be a gentle introduction to orchestration without much overhead. In either case, you can still treat the single node almost like a "bunch of containers" – just managed through the orchestrator. Both K3s and Nomad can run on the same hardware you'd run Docker on, with only modest extra resource usage (Nomad's overhead is negligible; K3s will use more but still manageable on a dev box). The decision between K3s and Nomad at this stage could boil down to what you're more comfortable experimenting with: Kubernetes has more learning materials; Nomad is simpler to conceptually grasp. Also consider any tooling you already use – for example, if your CI/CD or monitoring is already geared towards Kubernetes, K3s might slot in naturally. Conversely, if you love HashiCorp tools and perhaps already use Consul or Vault, Nomad might feel more natural.

In summary, for a quick start, **Docker/Podman with Compose is the path of least resistance** – you'll get your entire stack up and running offline in no time, leveraging skills and configs you might already have. This environment can be integrated with your DevOps tasks (backups, etc.) easily by mounting volumes from the containers and running Borgmatic on the host or in a maintenance container. On the other hand, if you want to be forward-looking and don't mind the initial overhead, **starting with K3s or Nomad in single-node mode** lays a foundation for clustering later without a big jump. It might slow down initial progress a bit due to the learning curve, but it will pay off when you start scaling out.

One compromise approach could be: start with Docker Compose for development, but in parallel, **begin writing Kubernetes manifests (or Nomad jobs) for your stack**. You can even run a local K3s cluster (or kind/minikube) to test these manifests in the background. This way developers get the simplicity of Compose, while the ops-minded folks prepare the k8s configs. When the time comes to go multi-node, you'll have those ready. Similarly, if using Podman, you could utilize Podman's `generate kube` feature to export your running containers as Kubernetes YAML as a starting point for a K3s deployment [45] [46]. The key recommendation is: **don't introduce unnecessary complexity too early**. Ensure whatever you use on one node is something you're comfortable with. It's fine to start small – the containers and images you create will be portable to whichever orchestrator you choose later, thanks to OCI standards [25]. Just avoid anything proprietary or tied to cloud services in this phase (which you already plan to, by using open-source tools).

Roadmap to Clustered Orchestration

As your system grows and you need to deploy across multiple nodes or sites, you'll want to transition from a single-node setup to a clustered orchestrator. Here's a suggested migration roadmap keeping **long-term self-reliance** in mind:

1. **Container Image Strategy (Registry):** In an offline or hybrid environment, plan how nodes will get container images. For clustering, it's wise to set up a **private registry (such as Harbor, Sonatype Nexus, or even a simple registry:2)** that all nodes can access (over LAN or sneakernet). In a single-

node phase you might just load images locally, but as you add nodes, a centralized registry (or a distributed mirror like K3s offers [18]) will simplify updates. Populate this registry with all your required images (application images and any base images like Postgres, Redis, etc.). This ensures that when the orchestrator schedules something on a new node, it doesn't attempt to pull from the internet, but from your registry (or it already finds it cached). This step is largely orchestrator-agnostic and can be done early. Tools like `crane` or `skopeo` can help sync images from an online source to your offline registry during periodic connections.

2. **Choose Orchestrator and Prepare Configs:** By this point, you likely have a candidate in mind (from Docker Swarm, K3s, full Kubernetes, or Nomad). Given the analysis, **K3s or Nomad** are strong choices for long-term maintainability on self-hosted edge clusters. Kubernetes (via K3s) has the edge in ecosystem and community support, while Nomad has the edge in simplicity and efficiency. If unsure, consider a pilot with both: bring up a small 2-node K3s cluster and a 2-node Nomad cluster, deploy a couple of your services to each, and evaluate things like ease of use, performance, and how well your team groks the workflow. Also consider integration with your existing tools: for example, if you already use Prometheus, how easy was it to get metrics from each orchestrator? If you use Grafana, can it pull metrics easily (in K3s, yes via Prom+exporters; in Nomad, yes via Nomad's metrics API or Consul)? Based on this, pick one orchestrator to avoid over-extending. (It's also feasible to use **Docker Swarm** for clustering if your needs are basic – it requires almost no new learning if you know Docker, and you can deploy your Compose file with `docker stack deploy`. This might be perfectly sufficient if your cluster is small and you don't need advanced features. Just be aware of Swarm's stagnation and potential need to migrate in the future if it hits limitations.)

3. **Cluster Setup in Stages:** Start by **extending to a two-node cluster**. For example, if using K3s, install K3s on a server node (with your SQLite or etcd datastore) and then join a second node as an agent. If using Nomad, start a Nomad server on one node and a client on another (Nomad can run server and client on same node too for small clusters). Deploy a subset of your workloads in this mini-cluster and ensure they work (data volumes, networking, etc.). This is where you handle **storage and networking setup**:

4. *Storage:* Decide how to handle persistent volumes in cluster. With K3s/K8s, you might use local volumes for each node (with something like Rancher Longhorn or openEBS if you want distributed storage, or manual replication). K3s supports local persistent volumes out-of-box [20]. If a service (like PostgreSQL) should only run on one node with its local disk, use affinities or NodeSelectors to pin it, and document that in your manifests. In Nomad, you can use host volumes or Nomad's CSI support to similarly attach external or network storage [47] [48]. For example, some Nomad users set up NFS or GlusterFS and mount it on each client, or use a CSI plugin for Ceph [49] [50]. The goal is to ensure that if the scheduler places a DB on a node, the data is there and ideally redundant or backed up. Initially, it's simplest to use host-local volumes and rely on your Borgmatic backups to cover disaster recovery (restoring data to a new node if one fails).

5. *Networking:* In Kubernetes/K3s, the CNI (flannel by default in K3s) will handle cross-node networking for containers. Ensure your edge network allows the nodes to reach each other on the necessary ports (flannel typically uses UDP overlay, which should work on LAN). For Nomad, if you use Consul, it will help with service discovery across nodes; if not, Nomad tasks can be configured with host networking or you might manually manage some service advertisement. On an isolated network, you might set up a local DNS or simply use static IPs for nodes. Also set up any ingress or proxy

needed: with K3s, Traefik can remain as ingress for HTTP services, or you might introduce Nginx ingress. With Nomad, you might run an HAProxy or Traefik job that listens on a node's port 80/443 and routes to services. These are implementation details, but the idea is to ensure that multi-node doesn't break how services talk to each other and to users.

6. **Configuration as Code:** Migrate your service definitions to the orchestrator's declarative config. If you started with Docker Compose, now is the time to translate that into either Kubernetes manifests or Nomad job files. There are tools to assist (e.g. Kompose for docker-compose to K8s, or as mentioned, Podman can generate K8s YAML). For Nomad, it's manual but straightforward since a Nomad job can have multiple groups for your different services. Keep these configs in version control (Git), so you have an auditable, revertible history of your infrastructure. This also sets you up for GitOps-style deployment if desired (especially easy with Kubernetes, using something like ArgoCD or Flux, but also achievable with Nomad via CI pipelines triggering `nomad job run`). Aim to use **declarative configuration** exclusively – avoid making ad-hoc changes via CLI that aren't captured in code.

7. **Testing Failover and Scaling:** Once your two-node cluster is running, test critical scenarios. For example, if using K3s with an external etcd, what happens when the server node goes down – does the agent continue running workloads (yes, but you lose control plane until it returns, unless you had a multi-master setup)? For Nomad, kill the leader node and see if the cluster self-elects a new leader properly (Nomad Autopilot should handle this if you had multiple server nodes). Also test scaling a service: e.g., run multiple instances of a stateless service (like an API or a worker) across both nodes – does the orchestrator distribute them and can your clients or load balancer handle that? This will reveal any networking or service discovery issues to iron out. In an edge scenario, also test network partition behavior if relevant (what if the two nodes lose connection? Each orchestrator has different behaviors – K3s single-master would basically isolate the agent, Nomad would elect a new leader if a minority partition, etc.). While in a fully offline single-site deployment you might not have partitions, if you ever plan to run multi-site, consider something like **Kubernetes federated clusters or Nomad's federation** (both allow multiple clusters in different locations that can either operate independently or with some central control).

8. **Incremental Node Addition:** Add more nodes as needed following the established pattern. It could be tempting to add many at once, but doing it incrementally lets you update your automation (Ansible scripts, cloud-init, etc. that you use to set up new nodes with Docker/Containerd and orchestrator agent). Each new node in K3s requires the join token and perhaps setting roles (worker vs server); in Nomad, joining a new client is typically just starting it pointing at the servers. Verify each new node can pull images from your registry (network and credentials are OK) and that scheduling lands some test workload on it.

9. **Monitoring and Logging Integration:** As the cluster grows, integrate monitoring. If using Kubernetes, you might deploy the Prometheus Operator or a lightweight monitoring stack to scrape metrics from your nodes and pods. If using Nomad, you could run Prometheus as a Nomad job and use the Nomad metrics endpoint (Nomad and Consul both expose metrics). Nomad doesn't automatically scrape each task's metrics like K8s can with exporters, so you might rely on node-level metrics (CPU, memory) and perhaps instrument your apps. Logging-wise, consider a local ELK stack or even simpler, make sure container logs are written to host volumes that your backup covers or that you have a log shipping mechanism (Filebeat, etc.) in place. The orchestrator choice may dictate

some tools (e.g., Fluentd with Kubernetes metadata vs. just tailing Nomad alloc logs), but ensure whatever solution is also self-hosted and doesn't require cloud. This monitoring/logging setup will ensure you maintain visibility in an offline environment.

10. **Backup and DR:** With multiple nodes, adjust your **backup strategy**. Instead of one machine's volumes, you now might have data spread on different nodes. Borgmatic can still be used: you could run Borgmatic on each node (perhaps as a scheduled Nomad job or a CronJob in K8s) to back up local volumes to a central backup server or attached storage. Alternatively, centralize backups by mounting remote volumes. For cluster state backups: if Kubernetes with etcd (or K3s with etcd HA), set up etcd snapshots. If K3s single-master with SQLite, you can simply back up the SQLite DB file regularly (located under `/var/lib/rancher/k3s/server/db` – it's easy to back up). Nomad's state (Raft snapshots) can be backed up too, though if all jobs are in code and data volumes are backed up, you could reconstitute the cluster from those if needed. Practicing restoring a backup on a fresh node is worthwhile to ensure your recovery process is solid – especially since offline means you can't rely on cloud recovery services.

11. **Scale and Optimize:** With the cluster in place, you can start optimizing for performance and resource utilization. For instance, if using K3s, you might evaluate switching the datastore to etcd for better performance if you grow beyond say 1-3 server nodes. Or you might enable traefik's aggressive tuning for many services. In Nomad, you might introduce Consul at this point if you hadn't, to get better service discovery and filtering (Nomad service discovery is basic without Consul). Also consider scheduling policies – e.g., bin pack multiple services on one edge node vs spread out – to either save power or improve reliability. These are fine-tuning steps. The key is that by now you have an orchestrator managing your containers, so scaling out further is mostly rinse and repeat.

Throughout this migration, maintain **documentation and scripts** for everything (installation steps for a new node, how to add it to cluster, how to recover). This ensures the solution remains self-reliant – anyone in your team (or future you) can rebuild or fix things without external help. It's wise to also keep an eye on updates for your orchestrator: for example, K3s releases updates frequently; plan how you will apply those in an offline way (perhaps by downloading the new binary and upgrading node by node). Nomad's release cycle is slower, but you should track when a new version has important fixes (the BSL license means only community "core" will be updated, but 4 years later it becomes MPL – generally you'll just use the official binaries which are fine to use freely).

In essence, the roadmap is: **get the fundamentals right (images, storage, network)**, **gradually add orchestrator components** while translating your configuration to them, and **scale out stepwise**, testing as you go. This measured approach prevents a big-bang migration which could be error-prone. Because you emphasized open-source and standards from the start, each step leverages that portability – your Docker images run on Kubernetes or Nomad unchanged, your Compose services map to analogous definitions in the new system, and your team's DevOps practices (like backup and monitoring) adapt rather than reinvent.

Licensing and Support Considerations

When choosing your long-term platform, it's important to consider licensing and the vibrancy of community/support, especially since you're investing in self-hosted tools (where community support is key). Here are some notes:

- **Docker:** Docker's engine is open-source (Apache 2.0), but Docker as a company has made moves that affect usage. Notably, as of late 2021 Docker Desktop (the bundled GUI+VM for Mac/Windows) requires a paid subscription for larger companies [29] [1]. Additionally, Docker Hub implemented rate limits for anonymous image pulls. In your case (self-hosted, presumably on Linux servers), you can use Docker CE freely even in production – just avoid Docker Desktop or ensure compliance with its license if your team uses it. The Docker Engine itself remains free for commercial use up to certain enterprise sizes [26]. Docker did *not* change the license of the engine – it's still Apache-2 – so using it directly on servers is fine. The risk is more around Docker's ecosystem (Hub) and whether future versions introduce any restrictions. To be safe, many organizations (especially gov or those with strict policies) have gravitated to alternatives like Podman or have stuck with Docker CE 20.x LTS. Given that you prioritize self-reliance, you might choose to **decouple from Docker's tooling that could be subject to change** – e.g., use Podman or containerd where possible, host your own registry instead of heavily relying on Docker Hub, etc. This mitigates any surprise licensing or rate-limit issues.

- **Podman:** Entirely open-source (Apache) and backed by Red Hat, which traditionally keeps things open. No licensing traps here – Podman is part of the upstream project (libpod) and is included in RHEL, Fedora, etc. Red Hat's business model around it is support via RHEL, but the project itself is community-driven. So Podman is a safe choice license-wise. It also has the advantage of no trademark/premium features like Docker has – e.g., there's no "Podman Enterprise" to upsell; it's just open source. Using Podman could thus alleviate any Docker licensing concerns and still let you run containers with minimal changes. One caveat: if your team uses *Docker Desktop*, switching to Podman on Mac/Win might involve using Podman Desktop or some VM, which is a different workflow but those are open/free as well (Podman Desktop is open source and free).

- **Kubernetes / K3s:** Kubernetes is under CNCF governance and is Apache 2.0 licensed – no single company can change that unilaterally. K3s was originally a Rancher product, but it's been donated to CNCF as a sandbox project, also Apache licensed. So the core is free and open. Some Kubernetes distributions (OpenShift) add their own licensing on top, but you likely wouldn't use those since OpenShift is heavy and geared to enterprise with support contracts. In your context, sticking with the open-source K3s or upstream Kubernetes means no licensing fees. Support comes from the community unless you contract a vendor (companies like SUSE (Rancher), Canonical, or Red Hat offer paid support for Kubernetes distributions, including K3s via SUSE). If that's a future consideration (maybe your system becomes mission-critical and you want a support contract), Kubernetes has more options for enterprise support than Nomad, for instance. One more note: any Docker-developed component in Kubernetes (like Mirantis still maintains some ingress controllers) are also open, so no issues there. Just ensure any specific add-ons you use carry compatible licenses (most everything in the K8s ecosystem is open source or at least source-available).

- **Nomad:** HashiCorp's switch to BSL 1.1 license means that starting from Nomad 1.4 (for example), the code is source-available but with the condition that you can't offer it as a service or repackage it

to compete with HashiCorp. For practically running Nomad in your own environment, this has **no impact on usage** – you can use and modify Nomad freely inside your organization [51] [52]. The only scenario you'd hit the license is if, say, you decided to offer a hosted Nomad service to external customers – then it would be disallowed until the license converts to MPL (which happens 4 years after release). The license does guarantee it *will* become truly open (MPL2.0) after that time period [51]. The reason this matters is more philosophical: some consider BSL not an OSI-approved open source license, which is true – if that conflicts with your values or policies, you'd need to be aware. There are already forked communities (e.g. an open-source fork of Terraform popped up after HashiCorp's change); as of now Nomad hasn't had a major fork. Assuming you're fine with BSL, you get the benefit of official binaries and updates from HashiCorp. If you were not okay with it, you could stick to Nomad 1.3 (the last MPL version) indefinitely, or use an unofficial fork if one emerges. But realistically, for an internal deployment, the HashiCorp license won't restrict you. It's worth noting that HashiCorp still offers enterprise versions and support – but if you want that, you'd have to pay (similar to paid K8s support). If not, you rely on community – which for Nomad is smaller, but HashiCorp does maintain good docs and their discuss forum.

- **Docker Swarm:** Swarm is part of Moby (Docker Engine) and is open source. No license fee. The concern here is more the lack of support/maintenance. Mirantis (who took over Docker's enterprise business) nominally maintains it, but as mentioned, it's not active. So you wouldn't pay money, but you also might not get help unless community folks step up. If Swarm fulfills your needs, you could run it as-is (there are users still running Swarm happily). Just factor in that you might eventually have to migrate if Swarm doesn't keep up with your requirements or if any security issues arise that aren't fixed upstream.

- **Community vs Vendor support:** In an offline, self-reliant scenario, having a strong community or readily available knowledge base is crucial. Kubernetes wins here – you'll find countless Q&A, guides, books, and a large pool of professionals. Nomad's community is enthusiastic and HashiCorp provides a forum, but it's smaller in sheer numbers. Docker/Podman are somewhere in between – very large user base for Docker, growing one for Podman. If you foresee needing troubleshooting help, Kubernetes might give you more Googleable answers, whereas Nomad might have you digging in HashiCorp docs or asking questions on their forum/Slack. That said, Nomad's simplicity means there are fewer weird issues you encounter.

- **Long-term viability:** Kubernetes is basically an industry standard now, so it's not going anywhere – you can expect updates and a healthy ecosystem for the foreseeable future. Docker as a technology (the image format, etc.) is also standard, though Docker the company is not as dominant as before. Podman is likely to stick around given Red Hat's backing and its use in RHEL (plus its growing adoption as the container engine in many contexts). Nomad is more niche; HashiCorp is committed to it, but HashiCorp has historically had a smaller user base for Nomad compared to, say, Terraform or Vault. The BSL move had some folks worry if HashiCorp might eventually focus only on big paying customers. However, they've shown continued development of Nomad, and it's used in production at organizations like Cloudflare, Roblox, etc., which lends it credibility.

Summary: Your safest bet from a pure licensing and support stance is **Kubernetes (K3s)** – fully open, huge support community, and multiple vendors that can back it if needed. **Docker/Podman** are both fine to use (with Podman having zero licensing worries, Docker requiring a bit of caution for non-desktop usage in large orgs). **Nomad** introduces a minor licensing nuance but nothing that hinders on-prem use; just be

aware if “100% open source” is a strict requirement, Nomad’s newer versions don’t qualify under OSI rules (though functionally it’s free). In any case, maintaining self-reliance means minimizing external dependencies: so run your own registries, don’t rely on cloud-managed control planes, and keep everything documented. With those practices plus the open-source tools discussed, you will have a container platform that is **fully under your control, offline-capable, and scalable**, setting you up for long-term success in your hybrid edge system.

---

[1] [26] [29] Docker License Changes in 2025
https://www.knowledgehut.com/blog/devops/docker-license-change

[2] [4] [6] [7] [8] [9] [25] [27] [30] [31] [32] [33] [34] 15 Best Docker Alternatives for 2025: Complete Guide with Pros, Cons & Migration | SigNoz
https://signoz.io/comparisons/docker-alternatives/

[3] [12] [28] [44] Docker Swarm vs Kubernetes vs Nomad - the orchestrator wars continue?
https://www.portainer.io/blog/orchestrator-wars-continue

[5] From Podman to Kubernetes: A Practical Integration Guide | Better Stack Community
https://betterstack.com/community/guides/scaling-docker/podman-to-kubernetes/

[10] [11] [22] [23] Expanso | Kubernetes vs Nomad vs Expanso: Choosing the Right Orchestrator
https://www.expanso.io/kubernetes-vs-nomad-vs-expanso

[13] [14] [15] [16] [19] [20] [35] [37] [38] [41] K3s vs K8s: Differences, Use Cases & Alternatives
https://spacelift.io/blog/k3s-vs-k8s

[17] [18] [42] Air-Gap Install | K3s
https://docs.k3s.io/installation/airgap

[21] [51] [52] Hashicorp's New BSL License – What Changed?
https://infisical.com/blog/hashicorp-new-bsl-license

[24] [36] [39] [40] arxiv.org
https://www.arxiv.org/pdf/2504.03656

[43] K3s is definitely more lightweight than K8s, but it still requires non ...
https://news.ycombinator.com/item?id=25118425

[45] [46] podman-generate-kube — Podman documentation
https://docs.podman.io/en/v4.2/markdown/podman-generate-kube.1.html

[47] [48] Discover how to create stateful workloads | Nomad | HashiCorp Developer
https://developer.hashicorp.com/nomad/tutorials/archive/stateful-workloads

[49] Do you run Nomad on-premises? Please share your experience!
https://www.reddit.com/r/hashicorp/comments/17w1hp6/do_you_run_nomad_onpremises_please_share_your/

[50] Selfhosted Nomad with clustered storage? : r/hashicorp - Reddit
https://www.reddit.com/r/hashicorp/comments/w67kqd/selfhosted_nomad_with_clustered_storage/