ChatGPT

Orchestration Strategies for Podman in Offline, Edge Environments

Overview

Podman is a daemonless container engine (often used as a Docker alternative) that excels in *offline-first, single-node deployments* and can run in **rootless** mode for better security. However, Podman alone is not an orchestrator – you need additional tooling or patterns to manage multiple containers, handle auto-start, restarts, dependencies, and eventually scale to multiple nodes. In an edge or disconnected scenario (no cloud/SaaS dependencies), the goal is to use lightweight, self-hosted orchestration that works offline and can later expand to a cluster.

This report examines various Podman-compatible orchestration methods, including **systemd integration (with Podman's Quadlet)**, **Podman's Kubernetes YAML support**, **Podman Compose (Compose files with Podman)**, and **HashiCorp Nomad**. We compare their features, trade-offs (including *rootless vs rootful* deployments), and how they support automation, observability, health checks, secrets, and fault recovery. Finally, we provide recommendations for an initial single-node setup and a roadmap for scaling to multi-node orchestration using Podman.

Podman + systemd Units (Legacy and Quadlet Integration)

One of Podman's strengths is its tight integration with **systemd**, the init system on most Linux distributions. Podman uses a simple fork/exec model (no daemon), which fits well with systemd's process management. In fact, Red Hat recommends using systemd to manage Podman containers for single-node or edge deployments [1]. By treating each container (or Podman pod) as a systemd service, you leverage a tool that is *already present and always running* on the host [2]. Systemd can then handle starting containers at boot, supervising them, restarting on failure, and ordering services.

Originally, admins could write systemd unit files manually or use `podman generate systemd` to create service files from running containers [3]. This allowed basic orchestration but resulted in large, hard-to-maintain unit files. **Podman Quadlet** was introduced (merged in Podman v4.4) to simplify this process [4] [1].

Quadlet: Declarative systemd for Containers

Quadlet allows writing **declarative** `.container` **unit files** (and related types) instead of full systemd service files [5] [6]. You define the container image and settings under a `[Container]` section, and Quadlet's systemd generator converts it into a proper service unit at runtime. *Think of Quadlet files as similar to a Docker Compose or Kubernetes YAML, but for systemd* [7]. For example, a simple Quadlet file might specify:

```
[Unit]
Description=My App Container
After=network.target

[Container]
Image=myapp-image:latest
Exec=/start-command.sh

[Install]
WantedBy=multi-user.target
```

When placed in `/etc/containers/systemd/` (or the user's `~/.config/containers/systemd/` for rootless), systemd will **auto-generate** a service (e.g. `myapp.service`) and manage it like a normal service [8] [9]. Quadlet supports several file types for complex setups: `.container` (single container), `.kube` (a Kubernetes YAML to run via `podman play kube`), `.network` (define a Podman network), and `.volume` (define a Podman named volume) [10]. This means you can either define containers one-by-one or define a multi-container pod using a Kubernetes YAML, all managed under systemd via Quadlet [11] [10].

Pros (Podman + systemd/Quadlet):

- *Lightweight & Offline:* Uses systemd (already on the system) – no extra daemons or external services. Perfect for edge devices with limited resources and no cloud connectivity. Quadlet was explicitly designed for single-node, disconnected/edge use cases [1].
- *Automation & Reliability:* Systemd ensures containers start on boot and can auto-restart on crash. You can use standard systemd features like `Restart=on-failure` and service ordering with `After=` dependencies [12] [13]. For example, a web container's unit can specify `After=db.service` to ensure the database container starts first [12]. If a container process stops, systemd will restart it automatically [13].
- *Security (rootless support):* You can run containers as systemd **user services** (rootless mode), keeping the container processes unprivileged. Quadlet works in rootless mode by placing files in the user's `~/.config/containers/systemd/`. This improves security – an exploited container can't gain root on the host – though it may require enabling lingering for the user so services run on boot.
- *Declarative & Maintainable:* Quadlet's declarative files hide the complexity of Podman's CLI flags [6]. The config is easy to version control and deploy to other nodes ("write once, deploy anywhere" principle [6]). This improves maintainability over large manual unit files.
- *Integration:* Podman's systemd integration lays the groundwork for advanced features like container auto-update and rollback. For example, Podman can use container healthchecks and image update labels in concert with systemd to periodically update containers and rollback on failure [14].
- *Observability:* Logs from Podman containers go to journald by default when run under systemd. This centralizes logs with other system services. Standard tools (`journalctl`, etc.) can be used to view logs and status. Basic metrics (CPU/memory) can be managed via cgroups.

Cons (Podman + systemd/Quadlet):

- *Single-node only:* Systemd orchestrates only the local machine. There's no built-in multi-node awareness or scheduling across hosts. Scaling out means manually replicating the setup on each

node (possibly via Ansible or similar). This can become hard to manage beyond a few nodes, since there's no central scheduler.
- *Limited High-Level Features:* Systemd handles process-level supervision well, but it lacks container-specific orchestration features. For example, rolling updates, health probes (beyond process alive checks), or dynamic scaling must be handled manually or via custom scripts. Systemd can restart a crashed process, but it doesn't know if your app is *unhealthy* (e.g., stuck but still running). You'd need external scripts or health-checker services to detect and restart unhealthy containers.
- *Complex Config for Many Containers:* While Quadlet simplifies single services, a deployment with many containers still means many unit files or a mixture of .container and .kube files. It's more fragmented than using one Compose or Kubernetes YAML. However, grouping related containers in a Podman pod (via a single `.kube` file) can mitigate this by treating a pod as one unit.
- *Rootless Caveats:* Running rootless containers under systemd user services requires some setup (enable lingering for the user or run systemd --user at boot). Not all networking features are available in rootless mode (rootless uses slirp or rootless CNI which may have performance/port limitations). If your app needs low ports (<1024) or access to host devices, rootless might pose challenges. Rootful (system service) might be simpler in those cases (at the cost of running containers with root privileges on the host).

Podman Compose (Docker Compose with Podman)

Using **Compose YAML** to define multi-container applications is popular for single-node setups. Podman supports this pattern in two ways:

1. **Podman Compose** – a Python-based community tool that reads a Docker Compose file and executes Podman commands (essentially a lightweight wrapper).
2. **Docker Compose with Podman's API** – Podman provides a Docker-compatible REST API socket, allowing the official Docker Compose (v2) to deploy containers via Podman as if it were a Docker daemon [15] [16].

Both approaches let you use the familiar `docker-compose.yml` format (the **Compose Spec**) to describe your services, networks, and volumes. This is declarative and easier to manage than raw Podman CLI for each container.

Pros (Compose with Podman):

- *Declarative Multi-Container Config:* You can describe an entire application stack (with multiple containers, their images, environment variables, ports, volumes, dependencies) in a single YAML file. This is simpler to maintain than numerous separate podman run commands or unit files.
- *Ease of Use:* Many developers are comfortable with Compose. It's quicker to spin up/down the whole application (`podman-compose up`, `podman-compose down`). This can be useful in development or small deployments.
- *Podman Compatibility:* The Podman team has worked to support Compose. Podman's Docker-compatible API means you can run `docker compose up` against Podman directly [16]. In fact, as of Podman 4.x, Docker Compose v2 (written in Go) works with Podman for most features [16]. Meanwhile, the community **podman-compose** tool is more tightly integrated with Podman's CLI and is often used for rootless setups [17].

- *Lightweight (No Daemon):* If using Podman Compose (the wrapper), it calls Podman commands directly, so no constantly running service is required [18]. The containers run under Podman's normal model (as child processes) and can be managed by systemd if needed (some users wrap `podman-compose up` in a systemd service to start all containers on boot).
- *Rootless friendly:* Podman Compose was designed with rootless in mind and naturally handles Podman pods and rootless networking better [17]. Docker Compose can also work in rootless mode via the Podman API socket, though it requires the Podman user service socket to be active.

Cons (Compose with Podman):

- *Single-node & No Native Scaling:* Like regular Docker Compose, it's meant for one host. There's no multi-node coordination or failover. All containers run on the same machine. Scaling (e.g., running 3 replicas) is possible only on that node. There is no knowledge of other nodes – to go multi-node you'd have to migrate to a different orchestrator or use Docker Swarm (which Podman does **not** support).
- *No Orchestrator Intelligence:* Compose doesn't automatically restart crashed containers unless you specify `restart:` policies in the YAML. It has basic dependency controls (`depends_on`) and can wait for healthchecks (Compose v3 allows waiting for a service to be "healthy"), but it lacks the sophisticated self-healing and scheduling logic of a true orchestrator. Essentially, it's a **one-shot** deployment tool – once containers are up, there's no active controller ensuring state (unless you run it attached, which isn't practical in prod).
- *External Management for Boot:* To run Compose apps at startup in an offline edge device, you'd likely still involve systemd (e.g., a systemd service that calls `podman-compose up -d` on boot). This adds a bit of complexity because now you manage the orchestrator (Compose) via systemd anyway. Alternatively, one could convert a Compose file to a Quadlet `.kube` or use `podman play kube` (see next section) for a more direct approach.
- *Feature Gaps and Stability:* The community Podman Compose might not implement every feature of the Compose spec, and it's an extra tool to install (which might be a challenge on an air-gapped network, though offline installers exist [19]). Docker Compose via Podman's API is fairly robust, but certain Docker-specific features (like BuildKit) may not be fully supported yet [16]. However, for most common use-cases it works.
- *Migration Path:* A Compose YAML can be a starting point, but migrating to multi-node (Kubernetes or Nomad) will require conversion. There are tools like Kompose to convert Docker Compose files to Kubernetes manifests, but it's not always flawless. If multi-node is a firm goal, you may prefer writing Kubernetes YAML from the start instead of Compose.

When to use: Compose with Podman is great for simplicity if you need a quick way to define a multi-container app on one node and you value familiarity. In production edge deployments, many teams have opted to skip pure Compose in favor of systemd/Quadlet or Kubernetes YAML, since Red Hat's direction is to focus on Podman's Kubernetes integration for better portability [20]. But Compose is still fully supported for those who prefer it, and Podman will continue to accommodate it [21] [20].

Podman “Kube” Mode (Kubernetes YAML with Podman)

Podman includes a powerful capability to work with **Kubernetes YAML** manifests on a single node. There are two key commands:

- `podman generate kube`: Export an existing Podman container or pod into a Kubernetes YAML definition.
- `podman play kube`: Launch containers on a Podman host based on a Kubernetes YAML file.

This effectively lets you use Kubernetes-style **declarative definitions** without running a Kubernetes cluster. For example, you can write a `myapp.yaml` that describes a Pod (or several Pods, Deployments, etc.), and Podman will create those containers on the local system.

Podman’s Kubernetes support has grown significantly. It can handle pods with multiple containers, volume mounts, environment configs, port mappings, and even **K8s init containers** and basic **PersistentVolumeClaim** definitions [22] [23]. Recent versions added support for K8s secrets and configmaps: you can create a K8s Secret YAML and do `podman play kube secret.yaml` to store it, then reference that secret in a Pod manifest – Podman will inject the secret value at runtime [24] [25]. Essentially, Podman aims to closely mimic Kubernetes Pod orchestration on one machine.

Pros (Podman Kube YAML):

- *Kubernetes Compatibility:* Perhaps the biggest advantage is **future-proofing**. By using Kubernetes YAML from the start, you ensure that your service definitions are ready to be deployed to a real Kubernetes or OpenShift cluster in the future. The Podman team explicitly encourages this approach, noting that containers run via `podman play kube` can be “easily moved onto an OpenShift (Kubernetes) cluster” when needed [20] [26]. This means your single-node deployment can serve as a stepping stone to multi-node Kubernetes.
- *Declarative and Portable:* A single YAML can describe complex apps (multiple pods, each with containers, volumes, etc.). This is portable and can be understood by any Kubernetes-aware tool or person. It’s also a **unified configuration** rather than separate unit files per service.
- *Podman Extensions for Edge:* Podman’s implementation extends K8s in some areas to suit single-node needs. For example, `podman play kube` can **build container images** on the fly if the image isn’t found but a matching `Containerfile` directory is present [27] [28] (something vanilla Kubernetes doesn’t do). This helps in offline scenarios where you might carry source and build locally.
- *Basic Orchestration Features:* You can specify a `restartPolicy` in the YAML (e.g. Always, OnFailure), and Podman will honor it by applying appropriate restart behavior to the container [23] [29]. Healthchecks defined in the Dockerfile should run as well (though Podman doesn’t have a controller to act on a failing healthcheck beyond logging it). You can also define dependencies implicitly by putting multiple containers in one Pod (they’ll share a pod network and start together), or separate pods can be started in sequence via an external script or systemd ordering if needed.
- *Secret Management:* As mentioned, Podman supports K8s secrets. In practice, you’d use `podman play kube` to create secrets, and then your pod YAML can reference them just like in Kubernetes [24] [25]. This avoids hard-coding sensitive data into configs and is done entirely offline. (Podman also has its own simpler `podman secret` for non-K8s usage [30] [31], but for the K8s YAML workflow, you use the K8s-style secrets.)

- *Integration with systemd:* You can combine this with Quadlet. As noted earlier, Quadlet supports `.kube` unit files that essentially run `podman play kube` on a given YAML [11]. This means systemd can manage your K8s-defined pod as one unit (auto-start on boot, restart on failure of the whole pod). This hybrid approach gives you the benefits of systemd's process supervision and Kubernetes config syntax. (Without Quadlet, you could still manually have a systemd service that calls `podman play kube myapp.yaml` at boot.)

Cons (Podman Kube YAML):

- *Still Single-Node:* It's important to note that `podman play kube` does **not** turn Podman into a multi-node orchestrator. There is no scheduling across machines or high availability beyond one node. If the node goes down, there's no controller moving those pods elsewhere (as Kubernetes would). You'd need to manually set up each node and perhaps use something like Ansible to coordinate multiple nodes with identical YAMLs (effectively treating each as a separate single-node cluster).
- *Limited Controllers:* Podman can interpret basic K8s objects (Pods, some Deployments, PVCs, etc.), but it isn't a full Kubernetes control plane. For example, a Kubernetes Deployment with replicas=3, if given to `podman play kube`, will start 3 pods on the same machine (that works), but if one of those containers exits, Podman doesn't have a Deployment controller to respawn it. In practice, Podman simply runs the containers as defined. If you require self-healing beyond restartPolicy or dynamic scaling of replicas, you don't have that with plain Podman. You'd need an external loop or cron to monitor and redeploy, or rely on systemd's restart for each container.
- *Overhead vs systemd:* Parsing and launching via YAML adds slight overhead compared to directly running containers or using Quadlet `.container` files. However, this overhead is minimal (just the Podman process reading YAML and creating containers). There's no heavy daemon – Podman still runs each container as a child process.
- *Learning Curve:* You need some knowledge of Kubernetes syntax. This might be overkill if your deployment is very simple. But given Kubernetes is widespread, this is often seen as an investment in the future.
- *Complexity in Debugging:* Without a Kubernetes dashboard or kubectl, debugging is a manual process (though one can use `podman ps`, `podman logs`, etc. to inspect the containers). Essentially, you have the complexity of Kubernetes objects, but not the rich ecosystem of tooling that usually comes with a Kubernetes cluster. For a small number of pods, this is typically fine.

HashiCorp Nomad (with Podman driver)

HashiCorp Nomad is a lightweight orchestrator that can manage containers (and other workloads) on one or many nodes. It's a single binary that serves as both server and client agent, making it appealing for edge or on-prem deployments where simplicity is key. Nomad does not require etcd or a complex control plane; it uses an internal consensus for the server cluster and is generally less resource-intensive than Kubernetes.

Nomad can run Docker containers by default, but importantly it has an official **Podman task driver plugin** [32]. This plugin allows Nomad to directly execute containers using Podman instead of Docker or containerd. In other words, Nomad + Podman can orchestrate containers across multiple nodes without needing any Docker daemon. The Nomad job specifications (HCL syntax) remain almost the same – you just specify `driver = "podman"` instead of `"docker"` for your tasks [33] [34]. Nomad will invoke Podman under the hood to run the container on each client node.

Pros (Nomad with Podman):

- *True Multi-Node Orchestration:* Nomad is a full-fledged orchestrator – it has a scheduler that can place containers on any node in a cluster, reschedule on failure, handle rolling updates, and scale out/in. If you foresee scaling to multiple edge nodes and want a single control plane, Nomad provides that. You can start with a single-node Nomad (server and client on one machine) and later join more client agents to the cluster seamlessly.
- *Lightweight & Offline:* Nomad's binary is relatively small and efficient. You can run the Nomad server on the same device (for a single node deployment) with little overhead. In an offline scenario, Nomad doesn't require cloud services – all cluster coordination is on-prem. As long as container images are available locally or in a local registry, Nomad doesn't need internet. (If using Nomad Enterprise features or cloud telemetry, those can be turned off for air-gapped use.)
- *Podman Integration:* Using the Podman driver means you keep the benefits of Podman (rootless containers, no Docker daemon). The driver is maintained by HashiCorp and community [35]. It supports key features like executing commands in containers, allocating ports, volumes, etc. – similar capability to the Docker driver [36]. This lets you remain "Podman-centric" even as you scale. Nomad essentially provides the high-level orchestration on top of Podman's runtime.
- *Flexibility (Mixed Workloads):* Nomad can orchestrate not just containers, but also VMs, batch jobs, or raw executables. For edge deployments that might be useful if you have some services not containerized. All can be managed under one Nomad cluster.
- *Declarative Job Definitions:* Nomad jobs are written in HCL, which is fairly straightforward. You can specify task groups, count (number of replicas), resource limits, dependencies, etc. Nomad will monitor the desired state and actual state, much like Kubernetes does, and attempt to keep them in sync (restarting failed tasks, etc.).
- *Built-in Features:* Nomad provides service discovery integration (with Consul, if desired), and health checking. For example, you can define a health check for a task that if fails can mark the task unhealthy. Nomad doesn't natively re-route traffic on health check failure (that's usually via Consul service registrations), but it will report health status. Nomad also has an event stream and UI/CLI for observability, and it can integrate with Vault for secrets – you can inject secrets (from HashiCorp Vault) into tasks without writing them to disk. Even without Vault, Nomad jobs can mark variables as sensitive to avoid logging them, and you could use environment variables or files for secrets in offline mode.
- *Security & Mode:* Nomad typically runs as a root daemon (the client agent needs privileges to spawn containers and allocate resources). However, the Podman driver **can run rootless containers** – this requires some extra configuration (cgroups v2, etc.) and Nomad running as root while dropping privileges for the task. The Nomad plugin documentation notes rootless is supported on systems with proper user namespace support [37]. This gives an option to run workloads rootless even under Nomad, though many Nomad users simply run containers as root and rely on AppArmor/SELinux for isolation.

Cons (Nomad with Podman):

- *Added Complexity (vs systemd):* Introducing Nomad means running at least one Nomad server process and a client agent on each node. In a single-node scenario, you might run both in one process (development mode) or as two processes. This is undoubtedly more moving parts than the systemd-only approach. It requires learning Nomad's concepts (jobs, task groups, allocations) and operational overhead (monitoring the Nomad agent, etc.). For a very simple deployment, Nomad might be overkill.

- *Memory/CPU Overhead:* While lightweight compared to Kubernetes, Nomad still uses some resources. The Nomad agent will consume a few tens of MB of RAM and some CPU, which on a small edge device might be noticeable but typically is acceptable. Systemd alone has near-zero overhead by comparison. So if every MB counts, consider this.
- *Web of Dependencies:* To unlock Nomad's full power (service discovery, health checks, secrets), you may need HashiCorp's ecosystem: **Consul** for service registrations/health checks and **Vault** for robust secret management. Both can be run in development modes or small clusters offline, but that again adds complexity. It's possible to use Nomad without them (Nomad can do basic health checks internally and you can provide secrets via environment variables), but many Nomad features assume their presence. In contrast, systemd/Podman approach would use simpler mechanisms for those concerns (or none at all).
- *Podman Driver Maturity:* The Podman driver for Nomad is relatively newer than the Docker driver. While it's supported, certain Docker-specific features may not work if the Podman API or CLI doesn't support them. For example, a StackOverflow discussion noted volume mounts in Nomad Podman driver lacked some support compared to Docker driver [38] . These gaps are closing over time, but it's something to verify in testing. Worst case, Nomad can still use the Docker driver if you run a Docker-compatible API (Podman's socket could potentially be pointed to by Nomad's Docker driver, though officially using the Podman plugin is the way to go).
- *No Kubernetes API/Ecosystem:* Nomad has its own API and tooling. If the wider community tools (like those built for Kubernetes) are important, Nomad would be a separate track. However, in an offline edge scenario, you might not need those cloud-focused tools anyway. Just note that Nomad's mindshare is smaller than Kubernetes', so community support and examples, while enthusiastic, are less abundant.

Rootless vs Rootful Orchestration

Rootless Podman allows running containers without root privileges on the host. This is a major security advantage for edge deployments, which might be unattended and in less secure environments. Rootless mode uses user namespaces and other kernel features to emulate root within the container while the host sees a normal user process [39] . All the methods discussed above can support rootless operation, but with varying effort and trade-offs:

- **Systemd/Quadlet (Rootless):** Podman can run as a user service. Each user on the system can have their own containers. Quadlet supports placing files in the user's config directory [40] . To have these start on boot without login, you'd enable lingering for that user (so their systemd --user instance starts at boot). The benefit is strong security isolation – even if a container escapes, it only gains the privileges of that unprivileged user. You can even run *different containers under different Unix users* for extra isolation (defense-in-depth) [41] [42] , though this becomes a management burden with file permissions, etc. (most deployments use one dedicated user for all containers). The drawback is certain capabilities (low ports, NFS mounts, raw sockets) might not work without additional configuration. Rootless networking uses slirp4net or rootless CNI, which has lower performance and doesn't allow incoming ICMP or certain protocols, though for many web apps this is fine. If your application needs to bind to, say, port 80, rootless Podman can use a higher port and you'd use firewall rules or a reverse proxy to forward, or use the `authbind` / `cap_net_bind_service` approach on the rootless network.
- **Systemd/Quadlet (Rootful):** Running Podman containers as root (e.g., system-level systemd units or Nomad agent as root) more closely mimics a Docker rootful setup. This allows full networking,

and easier volume sharing with host paths (no UID mapping issues). It's simpler to set up (no need for lingering user). SELinux can confine the container, and user namespaces can still be leveraged (Podman can run rootful containers with user namespaces too for extra safety). The obvious downside is if the container is compromised, it potentially has root on the host (although SELinux/AppArmor and dropping capabilities mitigate this significantly). Some practitioners choose rootful for production due to ease of use, especially if they trust their images or use additional hardening [43] [41]. It's a classic security vs convenience trade-off [44] [45].

- **Nomad (Rootless):** Nomad's Podman plugin can launch rootless containers, but Nomad itself typically runs as root. In practice, Nomad would call Podman's API or CLI as a certain user. Setting this up might involve running the Nomad client as the target user or configuring the plugin to drop privileges. It's doable on modern distros with cgroups v2 [37]. However, many Nomad deployments simply run containers rootfully for simplicity, especially if they already isolate at higher levels (Nomad can constrain CPU/mem per task).
- **Compose (Rootless):** Podman Compose and Docker Compose both work with rootless Podman. In rootless mode, a user can run `podman-compose up` without issue. The only caveat is services that need privileged ports or devices won't work unless the user has appropriate rights (for example, you can't easily use `device:/dev/ttyUSB0` in rootless unless that device's group permissions allow your user, etc.). For most app-level services, rootless is fine.
- **Kubernetes YAML via Podman (Rootless):** Also supported. The `podman play kube` will simply create rootless containers if run as a normal user. One thing to note is rootless Podman cannot create true "PersistentVolume" mounts that map to arbitrary host directories unless the user has access. Instead, rootless volumes reside in `~/.local/share/containers/storage/volumes` by default (within the user namespace). That means if you declare a host path in the YAML, it needs to be under your home or a path your user can access. In edge cases where data needs to persist to a specific host path owned by root, rootful might be needed or adjust permissions accordingly.

Summary of Rootless vs Rootful: Rootless is **more secure by default** (drops a layer of privilege), which is great for untrusted environments or multi-tenant scenarios on the edge. It integrates well with user-level systemd. The cost is some complexity (ensuring services run on boot, dealing with networking or filesystem permission quirks). Rootful is **simpler** and sometimes necessary for system-level services (and some container images that expect root). Many edge setups run a dedicated rootful Podman (possibly locked down via SELinux/AppArmor) for critical infrastructure, and use rootless containers for applications where possible. You can even mix: e.g., rootful Podman for a network-intensive proxy container, but rootless for less privileged apps. The right choice depends on your security requirements vs the convenience and features needed.

Comparison of Podman-Oriented Orchestration Options

The table below summarizes the key characteristics of each approach in the context of an offline, edge deployment:

Orchestration Method	Single vs Multi-Node	Podman Integration	Resource Footprint	Offline Friendly	Rootless Support	Features & Notes
Systemd Units (no Quadlet)	Single-node only	Podman as process executed via systemd (using `podman run` or generated unit files)	Minimal overhead (systemd is already running)	Yes – fully self-hosted	Yes (via systemd --user) or rootful	+ Rock-solid process management (restart, dependencies) [12]. + Easiest to bootstrap (comes with OS). - No awareness beyond one node. - Large manual unit files if not using Quadlet.
Systemd + Quadlet	Single-node only	Podman-managed by systemd using declarative `.container` / `.kube` files [10]	Minimal (same as above)	Yes	Yes (Quadlet works in user mode)	+ Simplified config ("compose-style" systemd files) [6]. + Explicitly designed for edge/offline use [2]. + Supports grouping containers (pods) via `.kube` files. - Still single-node. - Must learn Quadlet syntax (small learning curve).

Orchestration Method	Single vs Multi-Node	Podman Integration	Resource Footprint	Offline Friendly	Rootless Support	Features & Notes
Compose (Podman or Docker Compose)	Single-node only (no native multi-node)	Podman can run Docker Compose YAML via API [16] or use Podman-Compose tool	Low (CLI tool invocation; no daemon)	Yes (needs local images or registry)	Yes (supported in Podman)	+ Familiar Compose YAML format. + Good for dev/test and simple deployments. - Not an active orchestrator (no auto healing without restart policies). - Requires external trigger (systemd or manual) to start on boot. - Migration to cluster requires conversion (e.g., to K8s manifests).

Orchestration Method	Single vs Multi-Node	Podman Integration	Resource Footprint	Offline Friendly	Rootless Support	Features & Notes
Podman “Kube” YAML	Single-node (YAML portable to K8s multi-node)	Podman parses K8s YAML and creates local containers [46] [47]	Low (Podman CLI does the work, no persistent daemon)	Yes (no external dependency)	Yes (if run as user)	+ Declarative, Kubernetes-consistent definitions. + Easy path to Kubernetes later (reuse same YAML) [20] . + Supports pods, init containers, secrets, etc., on one node. - Lacks full K8s controllers (no scheduling or automatic rescheduling). - Single-node only; manual replication for multi-node.

Orchestration Method	Single vs Multi-Node	Podman Integration	Resource Footprint	Offline Friendly	Rootless Support	Features & Notes
Nomad + Podman driver	Multi-node capable (cluster or single)	Nomad orchestrates, uses Podman to run containers [32] [34]	Moderate (Nomad agent runs constantly; lightweight vs K8s)	Yes (self-hosted, no internet needed)	Yes (supported with config) but Nomad runs as root	+ True orchestration: scheduling, load balancing, rolling updates. + Scales from 1 to many nodes seamlessly. + Additional workload types (cron, VMs) in one system. - Additional components to run (Nomad, possibly Consul for service discovery). - Smaller community than Kubernetes; requires learning Nomad's HCL. - Podman driver may have minor feature lag vs Docker driver.

Sources: Podman/systemd integration [14] [12], Quadlet features [48] [10], Compose vs Podman details [49] [20], Podman kube YAML capabilities [46] [47], Nomad Podman driver info [32] [34].

Recommendations for Initial Single-Node Deployment

For an **offline, single-node edge** deployment starting out, the consensus among these options is to **leverage Podman's native systemd integration, using Quadlet for simplicity**. This approach aligns with the intended use-case of Podman on edge and provides a good balance of reliability and low complexity:

- **Use Quadlet (.container and .kube files)** to define your services declaratively. For each containerized service, write a `.container` file with the image, env vars, volumes, etc. If your application is composed of multiple containers that work together (e.g., a web app and a database), consider defining a Pod in a `.kube` file so they can share a network/pod and start together. Place these in `/etc/containers/systemd/` and do a `systemctl daemon-reload`. This gives you a one-step deployment per service and easy editing of configs. Quadlet will ensure that systemd knows about dependencies (it automatically creates `Requires=` and `After=` relations if you reference a `.network` or `.volume` file, etc. [50] [51]).
- **Keep it rootless if feasible.** Unless your containers need special privileges or ports, running them as a non-root user via systemd --user provides a safer default. Enable linger for that user so the services start at boot. Test that everything (network ports, file access) works in rootless mode. If you hit a limitation that is non-trivial to solve (for instance, needing to bind privileged ports or access USB devices), you could either run a specific container rootful or decide to run the Podman service as root on that device. Security is important for edge, but so is stability – so choose what makes maintenance easier and consider using SELinux (with `:Z` flags on volumes, etc.) to confine rootful containers if you go that route.
- **Leverage Podman's features:** Use `--healthcheck` in your container images (Podman will execute these, though you might need an external watcher to act on failures), use Podman secrets for any sensitive data (Quadlet can consume Podman secrets directly as shown by the `Secret=` key support from Podman 4.5+ [52]). Also, consider using the Podman auto-update feature if you have a way to provide updated images offline (e.g., via USB or local registry): by labeling containers with `io.containers.autoupdate=registry` and using systemd timers, Podman can periodically check and apply image updates [14] – this can be part of your maintenance routine on the edge device.
- **Observability on single-node:** Rely on journald logs (you can set up `journald` forwarding to a file or central syslog when connectivity is available). For metrics, you might not run a full Prometheus on the device, but you can use lightweight tools or even Nomad's fire-and-forget batch jobs if you incorporate Nomad later. On single-node, often simple scripts or systemd timers checking container status (via `podman inspect` or `podman healthcheck run`) can alert or take action. Ensure you have some way (even if it's just a cron that restarts the container or reboots the device on certain failures) to recover from common issues.

Why not Compose initially? Docker/Podman Compose is certainly an option for single-node and might be suitable if you already have a docker-compose.yml. However, given the offline, no-SaaS requirement and future scalability, the Quadlet approach covers the same single-node needs in a more integrated way. It also forces you to consider service dependencies explicitly via systemd, which is good practice for reliability. Compose would add an extra layer that ultimately systemd would have to manage anyway for auto-start, so it could be seen as redundant.

Alternate initial route: If your team is already familiar with Kubernetes manifests, you could opt to write a Kubernetes YAML for your application and use `podman play kube` (via a Quadlet .kube unit for auto-

start). This is slightly more complex at first but pays off later if migrating to Kubernetes. Quadlet will treat that like just another service unit.

Migration Roadmap to Multi-Node (Scalability-Ready Design)

Even though you start single-node, it's wise to design with **multi-node readiness** in mind. Here's a possible roadmap and considerations for evolving the orchestration:

1. **Phase 1 – Single Node (Podman + systemd):** Implement as above. Ensure stateless services and stateful data are separated (e.g., data stored in volumes or on the host, so that if you need to migrate a container to another node later, you can move its data). Use container images from a source you control (since edge is offline, you might run a local registry or periodically sneakernet updates). Document the setup with either Quadlet files or Kubernetes YAML so that it's clear what each service does.

2. **Phase 2 – Introduction of Orchestrator (if needed for growth):** As you add a second node, decide on the orchestrator:

3. If you want to remain **Kubernetes-compatible**, a natural progression is to move to a lightweight Kubernetes distro. **K3s** (Rancher's K3s) is a good candidate for edge – it's a single binary Kubernetes that can run on low-power hardware. K3s uses containerd by default (not Podman), but since your app is already described in Kubernetes manifests, you can deploy those manifests to a K3s cluster with minimal changes. Podman's YAML should be mostly compatible (you might need to wrap Pods into Deployments or StatefulSets for production use). Alternatively, **MicroShift** (a minimal OpenShift for edge) could be used – it's designed for offline and small environments, and uses CRI-O which shares heritage with Podman. Both K3s and MicroShift can operate in an air-gapped mode (you preload images).
4. If you prefer to stick with **Podman on each node without full Kubernetes**, you could treat each node like the single-node setup (with systemd/Quadlet) and use a configuration management tool to deploy services to multiple nodes. For example, Ansible could roll out Quadlet files to each new node. This isn't a true clustered orchestrator, but for a small number of nodes with fixed roles (e.g., each node runs a specific set of containers), it's a viable simplistic approach. It doesn't give you dynamic scheduling or failover though.
5. If you want a **simple, single control-plane** without jumping to Kubernetes complexity, consider adopting **Nomad** at this stage. You can start Nomad in parallel with your existing setup and gradually move workloads to it. For instance, set up a Nomad server on one of the nodes and Nomad clients on each node. Using the same container images, you can write Nomad job files that match what your Quadlet files did. You might initially run Nomad in "developer mode" on one node to experiment. Once comfortable, run Nomad in multi-node mode. Nomad can then take over the responsibility of keeping containers running, and you can remove/disable the systemd units for those services. The migration can be service-by-service to minimize disruption. One advantage: Nomad can pull from the same local registry or even reuse local images if you point it to Podman's image store (Nomad Podman driver will use the Podman engine which knows what images are present).

6. There is also a middle-ground tool called **Podman orchestrator** in development discussions (sometimes mentioned as “podman swarm” but nothing production-ready exists as of now). So the realistic choices boil down to Nomad vs Kubernetes when scaling out.

7. **Phase 3 – Multi-Node Operations:** After migrating, adjust your approach to fit the orchestrator:

8. With Nomad: You will now use Nomad’s CLI/UI or API to deploy updates. You get the benefit of Nomad’s scheduling – e.g., you can specify a job to run 1 instance on each of 3 edge nodes, Nomad will place them accordingly. If one node goes down, you can decide whether Nomad should reschedule that instance on another node (requires spreading data or having a stateless service). For stateful edge services, you might pin each service to a specific node using Nomad constraints (similar to Kubernetes nodeSelectors) – effectively treating Nomad more like an orchestrated init system.

9. With Kubernetes: If using K3s or another distro, you’ll start using `kubectl` to manage deployments. Ensure that the cluster can operate offline – for example, K3s can be started with registries configured to not hit the internet, and any Helm charts or operators you use are sourced from local files. At this point, Podman itself might be used only for development or troubleshooting (you could still use Podman on your laptop to test a pod then export it to YAML). On the nodes themselves, Podman can coexist with K3s, but K3s will use containerd for the actual runtime. If staying with Podman as runtime is critical, note that Kubernetes doesn’t directly support Podman, but uses CRI-O which is designed by the same team for Kubernetes – effectively giving equivalent capabilities.

10. **Phase 4 – Advanced Edge Scenarios:** As the deployment grows, consider **observability and management**:

11. For logs and metrics in a multi-node environment, a centralized approach is needed. With Nomad, you could integrate Consul for service discovery and health, and perhaps a lightweight aggregator for logs (Nomad can forward logs to its own UI, but long-term storage is up to you). With Kubernetes, you might deploy the usual EFK (Elasticsearch/Fluentd/Kibana) stack or a simpler Fluent Bit + local storage on each node depending on how much data you have. On edge, sometimes logs are just rotated locally and only pulled on demand to diagnose issues due to bandwidth constraints.
12. **Secret management:** In multi-node, it’s worth deploying a proper secrets store. If you went the Nomad route, HashiCorp Vault integrates neatly (and can run in HA with Raft backend in-cluster). Vault would let you securely distribute secrets to containers without ever exposing them in plaintext on disk. In Kubernetes, you could either use its built-in secrets (base64 encoded, not super secure by default but can be coupled with Vault or sealed-secrets for encryption) or also hook Vault in via a CSI driver or operator.
13. **Updates and Fault tolerance:** With multiple nodes, you can do rolling updates: Nomad has rolling update strategies (canary, blue-green) you can define in job specs; Kubernetes has Deployment rolling updates by default. Design your edge applications to tolerate restarts and node outages (e.g., use quorum or leader election if applicable, or at least auto-reconnect logic). If an edge device goes down, the orchestrator can start its workload elsewhere (if you have spare capacity and if the data is accessible on the other node, which might not be the case without a shared store or replication). Often in edge scenarios, each node might be pinned to a location or function, so “rescheduling” might mean waiting until the node is back up (because other nodes might not have connectivity to

the same devices or data). In such cases, orchestrators are mainly there for **uniform deployment and monitoring** rather than cloud-like elasticity.

14. **Resource constraints:** Edge hardware can be limited (CPU, memory). Nomad and Kubernetes both allow you to set resource limits/requests per container which is good to avoid one container starving others. Podman/systemd alone doesn't automatically limit resources unless you manually set cgroup limits in the unit file. In your initial single-node, consider cgroup limits via systemd's `CPUQuota`, `MemoryMax` directives or Podman's flags, so that as you move to a cluster, you have an idea of the resource needs of each service.

15. **Phase 5 – Long Term Maintenance:** Continue to favor tools that do not assume constant internet. For example, container image updates should be done via controlled batch jobs (maybe using `podman auto-update` with a local mirror, or manually loading images). Monitoring updates for orchestrator itself (Nomad or K3s) should be handled by a central management (could be an Ansible playbook run from HQ to update all edge nodes when needed, or over-the-air updates if available).

In summary, design your single-node deployment in a **declarative, orchestrator-agnostic** way as much as possible. This means using configuration files (Quadlet or K8s YAML) rather than imperative scripts, and containerizing all tasks. This makes it much easier to plug into an orchestrator later. The path of least resistance given the criteria is:

- **Start with Podman + systemd (Quadlet)** for a solid single-node foundation [2] [12].
- **Use Kubernetes YAML with Podman** or **Nomad** later to achieve multi-node, depending on your team's skills and the scale needed. If Kubernetes expertise is available or you want to leverage its ecosystem, transition your Quadlet configs (which can be partly generated into YAML) into a K3s or MicroShift cluster. If simplicity and low overhead are paramount, bring in Nomad to orchestrate Podman containers across nodes.
- Both paths keep you in control (no cloud dependency) and build on what you did in the single-node phase, rather than throwing it away.

Finally, ensure that whatever approach, you maintain **documentation and config as code**. Edge deployments can be hard to troubleshoot on-site, so having a clear declarative spec and using robust tools like systemd or Nomad means you can recover a node by reapplying the config. The choice of orchestration might evolve, but by focusing on Podman (OCI containers) at the core, you retain flexibility – the workloads (containers) remain the same, only the management layer swaps out when scaling.

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [14] [40] [48] Make systemd better for Podman with Quadlet
https://www.redhat.com/en/blog/quadlet-podman

[10] [11] [50] [51] [52] Deploying a multi-container application using Podman and Quadlet
https://www.redhat.com/en/blog/multi-container-application-podman-quadlet

[12] [13] Lightweight container orchestration in SUSE Linux Micro | SUSE Communities
https://www.suse.com/c/lightweight-container-orchestration-in-suse-linux-micro/

[15] [16] [17] [18] [20] [21] [26] [49] Podman Compose or Docker Compose: Which should you use in Podman?
https://www.redhat.com/en/blog/podman-compose-docker-compose

19 How to install podman-compose in an air-gapped network? #924
https://github.com/containers/podman-compose/issues/924

22 23 27 28 29 46 47 Build Kubernetes pods with Podman play kube
https://www.redhat.com/en/blog/podman-play-kube-updates

24 25 30 31 Storing sensitive data using Podman secrets: Which method should you use?
https://www.redhat.com/en/blog/podman-kubernetes-secrets

32 33 34 35 36 37 Podman task driver plugin | Nomad | HashiCorp Developer
https://developer.hashicorp.com/nomad/plugins/drivers/podman

38 Support for `volume_mount` in Nomad Podman task driver?
https://stackoverflow.com/questions/69000610/support-for-volume-mount-in-nomad-podman-task-driver

39 Adventures with rootless Podman containers - kcore.org
https://kcore.org/2023/12/13/adventures-with-rootless-containers/

41 42 43 44 45 Should I run podman-based systemd services as root? - Podman - Podman List Archives
https://lists.podman.io/archives/list/podman@lists.podman.io/thread/UXOGMIFKBVJHY3TEVZY7X2TZUV6BNO4K/