01 HOW TO MOVE FROM DOCKER SWARM TO KUBERNETES

The migration you control.

The state of Swarm in 2026, how Swarm concepts map to Kubernetes, and the bridge that makes a planned migration survivable.

PART ONE · 0:00 – 0:08

01

Why migrate now?

The state of Swarm in 2026, the ecosystem signals you're seeing, and the business case for a planned migration.

Lead the migration, or react to it.

PLANNED · YOUR TERMS

You choose the timeline.

- Sequence workloads in order of risk
- Train your team progressively
- Validate each move before the next
- You decide when Swarm retires

REACTIVE · FORCED BY CIRCUMSTANCE

A production incident decides for you.

- Migrate under pressure with no time to test
- People learn Kubernetes fighting fires in it
- No room to sequence - everything at once
- Swarm decides when Swarm retires

Swarm hasn't stopped working. The ground around it has shifted.

Many clusters continue to run stable workloads. Service scheduling, overlay networking, rolling updates... none of that has been removed. Pinned, well-managed clusters are running fine right now. The change is around Swarm, not inside it.

The commercial guardian of Swarm just pivoted away.

WHAT CHANGED · 5 MAY 2026

IREN acquires Mirantis · $625M.

- Mirantis has been the principal commercial steward of the Docker / Swarm enterprise stack since 2019
- Acquired by IREN - a former bitcoin miner turned AI-cloud operator - for $625M in stock
- The new strategic focus is AI infrastructure: k0rdent, GPU orchestration, neocloud
- Swarm is not the story their new owner is telling investors

WHAT IT MEANS FOR SWARM OPERATORS

The future is now even less certain.

- No commitment - public or private - on Swarm's long-term roadmap under new ownership
- Commercial support runway is unclear; community-only support is the realistic planning assumption
- Plugin and integration vendors track where the money is - and the money just moved
- The ground around Swarm shifts again

This is not a doomsday signal - Swarm clusters still run fine today. It's a planning signal. If you were waiting for a stronger reason to put a Kubernetes migration on the roadmap, this is it.

Four issues. Disproportionately affect Swarm.

ISSUE · 01 · CRITICAL

API minimum raised to v1.44

Any tool, plugin, or integration compiled against an older API is now rejected outright - no fallback. Cluster-wide plugins fail on upgrade.

ISSUE · 02 · CRITICAL

V1 storage plugins rejected

Legacy NFS, iSCSI, SAN/NAS plugins no longer load. Volumes appear inaccessible even though the data is intact. Caused at least one major site-wide outage.

ISSUE · 03 · HIGH

Mixed-version overlay silently breaks

In clusters with v29 and older nodes, encrypted overlay traffic stops between nodes. Services look healthy but become unreachable. No gradual upgrade path.

ISSUE · 04 · HIGH

Standalone containers hang on overlay

Long-standing bug: containers outside Swarm services on overlay networks hang on cross-node traffic. Workaround: deploy as services, not standalone.

CVE-2025-68121 · CVSS 10.0

Critical Go `crypto/tls` flaw. Every Docker Engine below 29.3.0 is affected - including the entire 28.x branch with no backported fix. The remediation is to cross the v29 boundary, which means inheriting the four breakages above.

Mar 2026

Architecture, not features.

a

nftables incompatibility

Swarm cannot run cleanly on hosts using nftables - the default on every modern Linux distro. `iptables-nft` compatibility shims keep most clusters running, but the shim is a fragile layer below your network: rule ordering and counter accuracy are not guaranteed, and silent traffic drops are the typical failure mode.

b

Overlay scale ceiling

VXLAN-based overlays become unstable above ~1,000 containers. Kernel-level constraint, not a configuration knob.

c

VXLAN port 4789 collisions

Swarm and VMware both default to UDP 4789. In virtualized environments, requires manual reconfiguration and ongoing vigilance.

d

Ecosystem contraction

Fewer tools, plugins, and integrations being maintained for Swarm. The surrounding ecosystem is shrinking even as the core continues to function.

Most teams end up doing two and three.

OPTION 01

Keep running Swarm.

Deliberate lifecycle management: pin versions, audit plugins, document recovery procedures. Manage the exit horizon explicitly.

BEST FIT · STABLE, LOW-CHANGE WORKLOADS · DOCUMENTED END-OF-LIFE

OPTION 02

Run both, in parallel.

Introduce Kubernetes for new workloads. Keep Swarm for existing services. Manage both platforms from a single layer.

BEST FIT · PROGRESSIVE ADOPTION · RISK-AVERSE MIGRATION TEAMS

OPTION 03

Phased migration off Swarm.

Inventory workloads, take the low-risk services first, translate stack files, validate on Kubernetes, retire Swarm clusters in order.

BEST FIT · CLEAR MODERNIZATION MANDATE · CAPACITY TO INVEST

The complexity is real. It's also worth it.

01

Ecosystem momentum

New tooling, security capabilities, observability, CI/CD - built for Kubernetes first, Swarm rarely.

02

Support trajectory

Active development. Regular releases. Large community. A defined long-term support path.

03

Operational capability

HPA, namespace isolation, advanced scheduling, native secrets at scale - first-class, not bolt-on.

04

Tooling integration

CI/CD, monitoring, policy engines, security scanners - deeper, more current Kubernetes integration.

PART TWO · 0:08 – 0:20

02

Swarm vs Kubernetes.
A reliable mental map.

Not a Kubernetes intro. A translation layer for what you already know.

Integrated engine vs. distributed control plane.

DOCKER SWARM

Integrated. Opinionated.

- Docker Engine = runtime + orchestrator
- Manager nodes form a Raft quorum
- Workers run Docker Engine directly
- Flat node model with differentiated roles
- Designed for simplicity, minimal configuration

KUBERNETES

Separated. Composable.

- Runtime (containerd, CRI-O) decoupled from orchestrator
- Control plane = discrete, replaceable services
- Workers run a kubelet agent against the API
- Rich taxonomy: taints, tolerations, labels, selectors
- Designed for extensibility and scale

Your mental model translates.

Almost every Swarm primitive has a Kubernetes counterpart. Some are direct one-to-one. Some take two Kubernetes objects where Swarm had one. A few require new thinking. The terminology changes, the model survives.

Don't memorize. Recognize.

SWARM		KUBERNETES	WHAT TO KNOW
Service	→	`Deployment + Service`	Scheduling and networking become separate, explicit objects.
Task (replica)	→	`Pod`	One or more containers sharing network + storage. Direct match.
Stack (compose)	→	`Helm chart · Kustomize · raw manifests`	d2k bridges this - stack files keep working during migration.
Overlay network	→	`CNI plugin + Service + Ingress`	Pod-to-pod is automatic. Service discovery via CoreDNS.
Secret · Config	→	`Secret · ConfigMap`	Direct equivalents. Richer ecosystem available afterward.
Named volume	→	`PersistentVolumeClaim + StorageClass`	Dynamic provisioning via CSI drivers - replaces v1 plugins.
Global mode service	→	`DaemonSet`	One pod per node. Agents, log shippers, CNI components.
Placement constraint	→	`nodeSelector · affinity · taints & tolerations`	More expressive. Larger configuration surface.
HEALTHCHECK	→	`Liveness · Readiness · Startup probes`	Three probes, three intents - finer control.

Overlay mesh becomes explicit primitives.

DOCKER SWARM

Overlay-first, manual attachment

- Create overlay networks. Attach services to them.
- Discovery by name within the attached network
- Ingress routing mesh - every node serves every port
- VXLAN on UDP 4789 - watch VMware conflicts
- No native east-west traffic policy

KUBERNETES

Pod-first, declarative wiring

- Every pod gets an IP - pod-to-pod just works
- `Service` · stable name + ClusterIP via CoreDNS
- `Ingress` · HTTP routing from outside
- `CNI plugin` · Calico, Cilium, Flannel…
- `NetworkPolicy` · east-west firewall

`ports: ["80:80"]` is doing more work than you remember.

DOCKER SWARM

The routing mesh is included.

- Publish port 80 - every node serves it
- Low ports, privileged ports - all fine out of the box
- Built-in L4 load balancing across replicas
- No extra component to install · no extra component to maintain

KUBERNETES

You wire it up. Three options, three trade-offs.

- `NodePort` · works everywhere, but ports 30000–32767 only - rarely what users type
- `LoadBalancer` · the clean answer · cloud-native; on-prem needs MetalLB / kube-vip
- `Ingress controller` · HTTP/HTTPS routing on 80/443 · still requires an LB or hostNetwork underneath

Reaching the same outcome as `ports: ["80:80"]` on Kubernetes means picking, installing, and operating one more component. The capability is there - it's just not free. Plan for this in the migration estimate.

Volumes become a contract.

SWARM · THE CAFETERIA	The party walks in, finds an empty table, sits down. Named volumes are mounted directly by Docker on whichever node the task happens to land on. No request, no catalog, no binding - just a mount.

Kubernetes makes the same need explicit. A pod asks for storage with specific requirements. The cluster consults what kinds of storage it has. A specific disk gets bound to the request - the way a maître d seats arriving guests.

THE GUESTS

PersistentVolumeClaim

“Table for 4, ReadWriteMany, please.”
→ 20 GiB, fast SSD.

→

THE SEATING PLAN

StorageClass

Booths, patio, bar. Each section has different capabilities and provisioners.

→

THE ASSIGNED TABLE

PersistentVolume

A specific disk, bound to the claim. Spun up on demand by the CSI driver.

Migration note: if your Swarm relies on V1 Docker volume plugins for NFS, iSCSI, SAN/NAS - moving to Kubernetes CSI drivers is not just a feature win. It's the fix for the v29 plugin-rejection issue causing production risk today.

Your NFS server doesn't move.

OPTION A · STATIC PV

Closest to the Swarm model · fastest path

```yaml
# PersistentVolume — points at the NFS export
apiVersion: v1
kind: PersistentVolume
metadata:
  name: app-data-pv
spec:
  capacity: { storage: 10Gi }
  accessModes: [ ReadWriteMany ]
  nfs:
    server: 192.168.1.50
    path:   /exports/app_data
  persistentVolumeReclaimPolicy: Retain
```

OPTION B · NFS CSI DRIVER

Dynamic provisioning · production-grade

```yaml
# StorageClass backed by NFS CSI driver
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: nfs-storage
provisioner: nfs.csi.k8s.io
parameters:
  server: 192.168.1.50
  share:  /exports
reclaimPolicy: Retain
volumeBindingMode: Immediate
```

RWX vs RWO matters. If multiple Swarm services mounted the same NFS export across nodes, you need ReadWriteMany in Kubernetes - choosing the wrong access mode is the most common migration mistake. Keep reclaimPolicy: Retain during migration so an accidental PVC delete doesn't take the data with it.

Where Kubernetes becomes a platform.

01
Workload shapes
Deployment · DaemonSet · StatefulSet · Job · CronJob - one tool per workload pattern.

02
Namespaces
Logical isolation. Per-team or per-app scope. The building block of a governable platform.

03
Fine-grained RBAC
Per resource, per verb, per namespace. Mapped to corporate identity. Auditable.

04
Autoscaling
HPA on CPU, memory or custom metrics. VPA. Cluster Autoscaler. None of which Swarm has.

Implicit trust becomes explicit grants.

SWARM TODAY

If you can reach the socket, you can do anything.

- No first-class user model inside the cluster
- Access is binary - you have the Docker socket, or you don't
- Teams operate on the assumption that everyone can see everything
- Auditability and "who did what" live outside Docker, if at all

KUBERNETES

No grant, no action.

- Every API call is authenticated and authorized through RBAC
- Access is scoped per resource, per verb, per namespace
- Mapped to corporate identity · audited · revocable
- The platform becomes governable - that's the point

This is a people problem more than a technology problem. Teams used to broad access will need explicit role definitions, and someone has to own that conversation. Day-1 surprise: a developer's first `kubectl` command fails with `Forbidden`. That's the system working - but it doesn't feel that way until the RBAC mental model is in place.

Harder in places. Better in others.

WHERE KUBERNETES IS HARDER

More surface, more deliberation.

- One service = many objects to manage
- RBAC, NetworkPolicy, admission controllers - all new
- Control plane HA needs deliberate design
- Upgrades, certs, etcd backups become your problem

WHERE KUBERNETES IS BETTER

Operational signal & recovery.

- Self-healing is more reliable: probes distinguish "starting" from "unhealthy"
- Rollout & rollback are first-class: kubectl rollout undo
- Events model, structured logging, metrics-server
- Deep tooling: k9s, Lens, Portainer, Datadog, Dynatrace

Before manifests, inventory.

THE 95%

95%

Migrate easily.

- Standard stateless & simple stateful services
- Container images don't change · app code doesn't change
- Stack files keep deploying via d2k while you migrate manifests
- This is the population the rest of the talk is sized for

THE 5%

5%

Need design work.

- Use Swarm features without a clean Kubernetes equivalent
- Macvlan, bind mounts, device passthrough, DNSRR…
- Sequence last, redesign, or leave on Swarm deliberately
- The audit's job is to tell you which apps are in this column

An audit is the cheapest insurance in a migration. Inventory services, volumes, networks and dependencies up front - so the 5% gets a plan instead of becoming a surprise mid-cutover.

What the audit is looking for.

NETWORKING · 01

Overlay semantics

Swarm overlay is `NAT-based`. Kubernetes pod networks are `routed`. Apps depending on source-NAT behaviour need review.

NETWORKING · 02

Macvlan

No native Kubernetes equivalent. Workloads needing MAC-level network presence: redesign, Multus CNI, or KubeVirt.

NETWORKING · 03

DNSRR endpoint mode

Kubernetes Services are VIP-based. DNS round-robin requires a `headless Service` or manual external wiring.

STORAGE · 04

Bind mounts

Largely unsupported. `hostPath` exists but is discouraged - rework as PVCs, ConfigMaps, or downward-API mounts.

HARDWARE · 05

Device passthrough

GPU and accelerators via `Device Plugins`. Full hardware passthrough = KubeVirt territory.

IDENTITY · 06

Swarm permission model

Some Swarm role primitives don't map one-to-one to Kubernetes RBAC. Access policy gets redesigned, not translated.

None of these are blockers - they're items that need a deliberate design decision before they cross. The audit's value is naming them before the cutover, not during it.

One stack entry. Four or five Kubernetes objects.

SWARM STACK · ONE BLOCK

docker-compose.yml

```
services:
  web:
    image: myapp:1.4.2
    ports: [ "8080:80" ]
    deploy:
      replicas: 3
      update_config:
        parallelism: 1
    secrets:   [ db_password ]
    volumes:   [ "app_data:/var/data" ]
```

→

01
Deployment
Pod template + rollout strategy.

02
Service
Stable network endpoint.

03
Secret
Mounted as file or env var.

04
PersistentVolumeClaim
Bound via StorageClass.

05 · OPTIONAL
Ingress
HTTP routing: when you need hostname / path rules instead of raw ports.

Move the platform first.
Apps follow at your pace.

DAY 1

You today, on Swarm.

- Stack files in production
- Existing CI/CD pipelines
- Team trained on the Swarm model
- Real workloads, real risk

BRIDGE

d2k

TEMPORARY BY DESIGN

DURING

Kubernetes underneath. Stack files on top.

- Same stack files keep deploying
- Same pipelines keep working
- Real Kubernetes experience, no forced cutover
- Migrate apps one at a time · d2k comes out at the end

Honest about scope.

WORK THAT'S REAL

- Storage classes & PVCs for stateful workloads
- Ingress / LB for external traffic
- Secrets management setup
- RBAC + namespaces for multi-tenancy
- CI/CD pipeline updates - but app-by-app with d2k
- Monitoring & observability re-validation

OFTEN UNDERESTIMATED

- ▲ Stateful services with complex storage
- ▲ Network topology if you rely on overlay segmentation for security
- ▲ Operator knowledge ramp — budget time for the mental model
- ▲ Scripts & tooling that talk directly to Docker / Swarm API

OFTEN OVERESTIMATED

- Application code changes - typically zero
- Container images don't change
- The need to migrate everything at once - d2k removes that pressure
- Risk of "stuck" - the path is reversible during transition

26 END OF INTRODUCTION · 0:20

That covers the map. Now we build it.

Next: the demo. We'll connect to a Talos Kubernetes environment provisioned and managed through Portainer, install d2k, and show your existing Swarm workloads running on Kubernetes - with no changes to your manifests or your pipelines.

The bridge has three other destinations.

USE CASE · 01

Dev containers.

One Kubernetes namespace per developer. Point the local Docker CLI at d2k - every engineer gets a personal remote Docker host backed by the real cluster. Same isolation, same RBAC, no laptops melting.

OUTCOME · CONSISTENT DEV ENVIRONMENTS · GOVERNED CENTRALLY

USE CASE · 02

GPU development.

The same pattern, scheduled onto nodes with the NVIDIA operator. Data scientists keep typing `docker run` and get a real GPU on the other side - no hand-rolled manifests for every experiment.

OUTCOME · SHARED GPU POOL · FAMILIAR INTERFACE

USE CASE · 03

Edge deployments.

Pair d2k with `KubeSolo` - the single-node Kubernetes distribution - for a lightweight, GPU-capable Docker environment at the edge. Not stuck on Docker-CE or Podman.

OUTCOME · MODERN EDGE RUNTIME · ONE OPERATOR VIEW

d2k was built for the migration, but the shape it gives you - a Docker-style surface over a Kubernetes-style platform - is useful well after the last Swarm cluster is retired.

What d2k doesn't do - yet.*

01

No image build support

The build pipeline stays where it is today. d2k handles the run side - `docker build` is not in scope.

02

No global mode — coming soon

The DaemonSet-style "one task per node" equivalent isn't shipped yet. Workloads that depend on it stay on Swarm until it lands.

03

Docker networks ignored

Custom `networks:` blocks in stack files don't translate. Kubernetes CNI + Services take over the wiring underneath.

04

No direct device access

`--device`-style host passthrough isn't supported. GPUs go via the NVIDIA operator; arbitrary hardware needs a deliberate design.

05

No privileged containers

`privileged: true` is not honored. Workloads that need it should be redesigned around Kubernetes capabilities or stay on Swarm.

→

Active development

d2k is improving continuously. We don't publish a date-bound roadmap - but if a gap above blocks your migration, that's the input that shapes what lands next.

* "Yet" indicates active investigation, not a commitment. We won't guarantee that every item on this list ships - some may stay out of scope by design.

Your questions, on the record.

Q.

Is d2k meant to be used just temporarily during the migration - or do d2k namespaces and references stick around afterwards?

Temporary. Once your apps are live on Kubernetes they're Kubernetes-native and d2k can be removed. One caveat: the apps remain in whichever namespace d2k was deployed into - so name that namespace something you're happy to keep.

Q.

Does this only work for Swarm workloads, or can it also move non-Swarm Docker workloads? We have a mix.

Both are supported. d2k actually started as a Docker-only translation layer - Swarm support was added later. Plain Docker workloads and Swarm stacks are both first-class.

Q.

We run a mixed Swarm and RKE2 environment. Would d2k ever support RKE2 vs. upstream K8s?

Already supported. d2k interfaces directly with the standard Kubernetes API, so any conformant distribution works - RKE2, vanilla, managed cloud, all of it.

Q.

How seamless is the Portainer experience when pointed at a d2k-fronted environment, compared to the CLI?

Very seamless. Portainer sees a d2k-fronted environment as a Docker environment - just like any other Docker endpoint. No knowledge of the underlying Kubernetes infrastructure required.

How d2k behaves under the hood.

Q.

When we build a stack through d2k, does it essentially clone the Docker service into native Kubernetes parts? Could we then take down d2k once we have the manifest?

Correct. d2k does a live translation into native Kubernetes objects and deploys them on Kubernetes. Once you're done, you can instantly kill d2k - the apps keep running. It's the Swarm management/operations UX on top of Kubernetes, with d2k handling all the translation. (And no - this isn't Kompose underneath. We wrote our own translation layer with significantly more depth.)

Q.

Is there a way to just generate the YAML to files rather than firing at the K8s API? And does the tool warn about unsupported properties in stack files?

YAML export: not supported today - something we may look at in future. **Unsupported properties:** d2k won't silently proceed - it fails with an explicit error telling you exactly what isn't supported.

Q.

How would you go about deciding how to install Kubernetes? It's not as simple as Swarm being built into Docker.

Our recommendation: `Omni` from Sidero Labs to provision Talos Kubernetes nodes, via the Portainer integration. That said, d2k supports whatever Kubernetes you bring - the install method is your choice.

Q.

Any known issues or incompatible configurations to watch for? Or will it loudly fail and leave environments unchanged?

The primary limitations are on the previous slide. If a deployment tries to use a method d2k doesn't support, it fails up front - it won't half-apply or silently change your environment.