Docker SwarmK

A unified management platform for Docker Swarm with real visibility, control, and automation

APRIL 20, 2026

THEDECIPHERIST.COM

TABLE OF CONTENTS

> ***SwarmK is currently in beta.*** *We're actively looking for teams who want to help us shape it into something truly excellent. If you run Docker Swarm in production, or you've been putting it off because managing it was too painful, we want your feedback. Every bug you find, every rough edge you hit, every feature you wish existed: that's exactly what we need. Join the beta, kick the tires hard, and help us build the thing we all actually want.*
> ***Want to join the beta?*** *Email me@thedecipherist.com to get access. We're especially looking for testers running servers on* ***Hetzner, Vultr, or AWS****, if that's you, jump in.*

There's a moment most developers know well. You've got a Docker Swarm cluster running, maybe two nodes, maybe ten, and something goes wrong at 2am. You SSH into each machine, grep through logs, tweak configs, restart services, and hope you didn't miss anything. If you're lucky you have some notes from the last time this happened. If you're not lucky, you're figuring it out fresh.

SwarmK exists because that moment shouldn't have to happen the way it does. It's a unified management platform for Docker Swarm, built to give you real visibility, real control, and real automation without requiring you to babysit shell sessions across your fleet.

But calling it a "management UI" undersells it. SwarmK is a complete infrastructure layer: networking policies, encryption tunnels, DNS management, a Web Application Firewall, load balancing, rate limiting, backups, webhooks, alerts, audit logs, and more. It runs an agent on each of your servers and coordinates everything from a single control plane.

And, this is the part that matters most, **you get your first agent completely free, forever**.

WHAT PROBLEM DOES SWARMK ACTUALLY SOLVE?

Docker Swarm is genuinely good software. It's simpler than Kubernetes, ships with Docker, and handles most production workloads without ceremony. The problem isn't the orchestrator, it's everything around it.

Managing a fleet of Swarm nodes without tooling means SSH into each server individually. It means writing scripts to coordinate deployments. It means piecing together visibility from multiple log sources. It means manually managing firewall rules that quietly drift from your intentions. It means hoping someone remembers what was changed and when.

Tools like Portainer give you a UI over the raw Docker API. That's useful. But it doesn't give you a network policy engine. It doesn't encrypt traffic between your services. It doesn't run a WAF, manage your DNS zones, or send alerts when a container starts crashing in a loop. And it doesn't get rid of the fundamental problem: you still need direct access to the server.

Dokploy is excellent for deployment automation, but it's focused on application delivery, not infrastructure networking and observability.

SwarmK is doing something different. It's not a thin wrapper over Docker, it's a full infrastructure platform that happens to orchestrate Docker Swarm underneath.

THE NO-SSH ARCHITECTURE

This deserves its own section because it's genuinely different from how most tools work.

SwarmK doesn't connect to your servers over SSH. Once you install the agent on a machine, which does use SSH one time, just for the initial install, the agent maintains a persistent, encrypted WebSocket connection back to the control plane. Every operation after that goes through that channel.

You want to restart a service? Command goes through the WebSocket. You want to inspect a container's logs? Same channel. Run a shell command, pull a new image, update firewall rules, all of it flows through the agent connection, not an SSH session.

Why does this matter? A few reasons:

Security. SSH is a powerful, dangerous protocol. An exposed SSH port with a weak key or compromised credential is a catastrophic failure mode. SwarmK agents don't expose any inbound ports. The connection goes outbound from the server to the control plane. You can lock down SSH entirely after the initial setup and never open it again.

Reliability. SSH sessions time out. Network interruptions break them. Managing a fleet over SSH at scale means managing connection state constantly. The WebSocket agent reconnects automatically with exponential backoff. A server that temporarily loses internet connectivity picks right back up when it comes back online.

Observability. When commands go through a structured API layer, you get something SSH doesn't give you: a full audit trail. Every command executed, every policy applied, every user who made a change, all of it is logged with timestamps, roles, and outcomes.

GETTING STARTED: ONE CLICK, FULLY RUNNING

One of the biggest friction points with infrastructure tools is getting from zero to actually managed. SwarmK removes that friction almost entirely.

If you're starting from a fresh VPS, SwarmK can provision it completely, one click. It connects via SSH one time, installs Docker, initializes Docker Swarm, and deploys the SwarmK agent. By the time you close the modal, your server is fully set up and reporting back to the dashboard. You don't touch the command line. You don't write a setup script. You don't follow a twelve-step guide. One click.

If you already have servers running, existing Docker setup, existing services, existing Swarm configuration, SwarmK works with what you have. Install the agent on your existing machine and SwarmK discovers what's already there. Your running services show up in the dashboard. Your existing Swarm nodes get mapped. Nothing needs to be redeployed or reconfigured inside SwarmK to start managing it.

This is a genuine difference from tools like Portainer and Dokploy, which require you to define and deploy your services through their interface to bring them under management. If you have an existing stack, you're either migrating to their model or

working outside it. SwarmK doesn't care how you got to where you are. It just works with what's there.

THE AGENT MODEL

SwarmK's architecture is built around agents. An agent is a lightweight Node.js process that runs on each of your servers alongside the Docker daemon. It handles local policy enforcement, collects metrics, executes commands from the control plane, ships logs, and keeps a heartbeat going.

When you provision a server through SwarmK, whether that's a fresh VPS or an existing machine, the agent gets installed. After that, the server shows up in your dashboard as a managed node.

Your first agent is free. Always. This isn't a trial. It's not time-limited. One agent per company is permanently free, which means you can run a single-node setup, a staging environment, a personal project, a homelab cluster, without paying anything. If you add more agents, you pay for the additional ones, but the first one stays free regardless.

Every agent reports back continuously: CPU, memory, disk, network throughput, load averages. That telemetry goes into time-series storage with a 30-day history, so you can see exactly what your server was doing last Tuesday at 3am when that spike happened.

THE DASHBOARD

The dashboard is a React application that gives you a live view of everything SwarmK is managing. It's built on TanStack Query for efficient data fetching and Zustand for state management, which means it stays snappy even when you're managing a large fleet.

The overview page shows aggregate stats: how many agents are online, how many network policies are active, how many tunnels are established, the current alert state. It's designed for the kind of at-a-glance situational awareness that tells you in three seconds whether something is wrong.

From there you can drill into anything. Agents, servers, services, databases, logs, traffic flows, alerts, all organized into a clean hierarchy that mirrors how you actually think about your infrastructure.

SERVER & VPS MANAGEMENT

SwarmK can provision and manage VPS servers across more than a dozen hosting providers: Hetzner, Vultr, DigitalOcean, Linode, Hostinger, Contabo, Scaleway, UpCloud, AWS EC2, OVHcloud, and more. You add your provider credentials once (encrypted at rest with AES-256-GCM), and then you can spin up, power-cycle, inspect, and tear down servers directly from the platform.

When you delete a server, SwarmK handles the cascade: the agent gets removed, API keys are revoked, metrics history is cleaned up, related services and DNS records and SSL certificates are updated. You don't have to manually trace through all the things that referenced that machine.

PROJECTS, ENVIRONMENTS, AND SERVICES

Applications in SwarmK are organized as Projects, which contain Environments (dev, staging, prod), which contain Services. Environments map to Docker Swarm stacks with overlay networks for isolation, your staging and production services don't share network space unless you explicitly connect them.

Service management is what you'd expect: deploy, scale, restart, remove. But it's all done through the agent command system, which means every operation is auditable, observable, and non-blocking. You trigger a deployment and get back a command ID. You can poll for completion or let it run in the background.

SwarmK can also discover services you deployed manually. Stack sync detects running services and brings them under management without requiring you to redeploy anything.

THE NETWORKING LAYER: KUBERNETES-GRADE, WITHOUT KUBERNETES

This is the part of SwarmK that has no real comparison in the Docker Swarm world. What's been built here isn't a management UI bolted onto Docker's networking. It's a complete network control plane, enforced at the kernel level, that adds capabilities Docker Swarm simply does not have natively. Load balancing algorithms. Stateful firewall policies. Per-service rate limiting. Bandwidth shaping. WireGuard encryption between service groups. Authoritative DNS with filtering rules. Real-time traffic flow visibility down to the byte.

To understand why this is different, you need to understand how it works underneath.

The Daemon: Enforcement at the Kernel Level

Every SwarmK agent runs alongside a daemon process on the VPS. The daemon is not a container. It's not a userland proxy. It's a dedicated enforcement engine running directly on the host, and it communicates with the agent through a JSON-line IPC channel over stdin/stdout, not MongoDB, not HTTP. When the API goes down, the daemon keeps enforcing. Rules don't disappear because the control plane is temporarily unreachable.

The daemon owns the entire kernel enforcement stack: nftables for firewall policies, rate limiting, and load balancing; Linux traffic control (`tc`) for bandwidth shaping; WireGuard for tunnel encryption; and BIND9 for DNS. Every 30 seconds it reads conntrack state and nftables counters to generate flow telemetry. Every 15 seconds it sends a heartbeat with operational stats. It watches the Docker event stream in real time and re-resolves group membership whenever a service is created, updated, or removed.

This architecture is what makes every networking feature in SwarmK genuinely real-time and genuinely kernel-level. Not a proxy hop. Not a sidecar. Not a software-defined abstraction layer with userland routing. Packets hit nftables rules inside the Linux kernel, at microsecond latency, on every node in your cluster.

Network Groups: Policies That Follow Your Services

Before you can understand SwarmK's policy engine, you need to understand groups, because groups are what make the policies useful.

A network group is a logical collection of services defined by match criteria: Docker service labels, explicit service names, CIDR blocks, FQDNs, or Docker network membership. You create a group called "api-tier" that matches any service with the label `tier=api`. The daemon on each node continuously resolves that group to real IPs, VIPs, and container addresses. When you deploy a new API service with that label, it joins the group automatically within the next resolution cycle. When a service is removed, it's gone from the group. No manual updates, no stale references.

This is the abstraction that makes your firewall policies durable. You write a policy that says "api-tier can talk to db-tier on port 5432." You don't write "10.0.1.5 can talk to 10.0.2.3 on port 5432." The policy means what you intended regardless of what IPs your containers land on.

Docker Swarm has no equivalent concept. Kubernetes has NetworkPolicy with pod selectors that works similarly, but only if you're using a CNI plugin that enforces it. SwarmK does this for Docker Swarm natively, through the daemon, with no additional infrastructure required.

Network Policies: A Real Firewall Engine

SwarmK's policy engine compiles your rules into six nftables chains, running in the Linux kernel on every managed node: `admin_ingress`, `admin_egress`, `service_ingress`, `service_egress`, `base_ingress`, `base_egress`. Three tiers, enforced separately.

The tier system matters. Admin-tier policies are set by platform administrators and cannot be overridden by anyone below them. Service-tier policies are managed by your teams for service-level access control. Base-tier policies are platform defaults. Within each tier, rules are evaluated by priority, first match wins. The combined effect is a proper layered firewall: your infrastructure security team sets non-negotiable baselines at the top, individual teams manage their own service boundaries in the middle, and sensible defaults protect everything at the bottom.

Every rule targets source and destination groups, CIDRs, or FQDNs. Actions are allow, deny, log, or pass. Protocols and port ranges are configurable. Every rule is wrapped with a kernel counter so SwarmK can track exactly how much traffic each rule is matching. Every policy change is recorded in an immutable history collection.

There is no native Docker Swarm equivalent. Docker's networking is open by default and provides no built-in mechanism to restrict traffic between services. You either implement this yourself with host firewall rules (fragile, manual, not topology-aware) or you don't have it at all. SwarmK gives you the full policy engine with a GUI, dynamic group resolution, and per-rule observability.

Traffic Flow Monitoring: See What Your Services Are Actually Doing

Every 30 seconds, the daemon on each node reads the Linux conntrack table and queries nftables counters. It converts that raw data into structured flow log entries: source service, destination service, port, protocol, bytes, packets, and the policy action that was applied. Those entries are upserted into MongoDB with 7-day retention, bucketed by minute.

The result is a real-time view of exactly how your services are communicating, which policies are allowing or denying which flows, and how much traffic is crossing each connection. The topology graph view takes this data and draws your actual service dependency graph, not the one you wrote in documentation three months ago, the one observed from live traffic.

This capability has enormous practical value. When something breaks, you see immediately whether traffic is being blocked by a policy. When you're doing a security audit, you see exactly what's talking to what. When you're trying to understand why latency is up, you can see which service pair is pushing unusual traffic volumes. And when you're planning a refactor, you can check whether the service boundaries you think exist actually exist.

None of this exists in Docker Swarm natively. It's a capability that typically requires a full service mesh like Istio or Linkerd to achieve in Kubernetes environments.

WireGuard Encryption: Opt-In, Per-Group, Fully Managed

Docker Swarm has optional overlay network encryption. It's automatic, it applies to the whole overlay, and it's relatively opaque. SwarmK takes a different approach: explicit, opt-in WireGuard tunnels between specific service groups.

When you create an encryption tunnel between two groups, the daemon on the relevant node generates a WireGuard keypair, creates a WireGuard interface inside the container's network namespace, assigns an overlay IP from a /30 subnet (`10.99.x.0/30`), and sets up the peer configuration. The tunnel goes from pending to active within seconds. You see the public keys, the interface name, the overlay subnet, and the establishment timestamp in the dashboard. You control which service groups are encrypted and which aren't.

This granularity is the point. Not everything needs encryption. Encrypting your database-to-cache link and your external-API-to-backend link while leaving internal gossip traffic unencrypted is a reasonable architecture. SwarmK lets you express that intent precisely. Docker's overlay encryption is all-or-nothing. WireGuard tunnels in SwarmK are per-group-pair, created and destroyed on demand.

Load Balancing: Algorithms Docker Swarm Doesn't Have

Docker Swarm's native load balancing is IPVS in VIP mode: round-robin across healthy replicas, managed by Docker. It works well for most use cases. But it's not configurable. You can't choose the algorithm. You can't weight replicas differently. You can't enforce session affinity. You can't route by least active connections.

SwarmK's load balancer adds six algorithms on top of Docker's networking, enforced via nftables DNAT rules for services in DNS round-robin mode: round-robin, least-conn (routes to the replica with the fewest active connections), ip-hash (consistent routing per client IP), random, weighted (per-replica weights you define), and sticky (session affinity with a configurable TTL). Health checks with configurable path, interval, timeout, and failure threshold protect against sending traffic to unhealthy replicas.

For services using Docker's native VIP mode, SwarmK respects Docker's IPVS and reports the enforcement status as "VIP mode" in the dashboard. For services using DNS round-robin, SwarmK steps in with its own kernel-level enforcement. The

enforcement status is tracked per service: enforced, pending, skipped, no replicas, no agents online. You always know whether load balancing is actually active.

This is the kind of capability that exists in Kubernetes via services, ingress controllers, and service meshes, each requiring separate configuration and understanding. In SwarmK it's a single config per service, set through the dashboard.

Rate Limiting: Per-Service, Per-Source, Kernel-Level

Docker Swarm has no rate limiting. You implement it in your application code, or you don't have it.

SwarmK's rate limiter generates nftables rules using the kernel's token bucket algorithm. You configure rate (requests per second, minute, or hour), burst allowance, and scope: global (the service as a whole is limited) or per-source (each client IP gets its own independent limit). The enforcement is at the packet level, in the kernel, before the request ever reaches your application. A service configured for 100 requests per second with burst 20 will never see more than 120 requests per second regardless of what's hitting it, and each limit is applied independently for per-source mode.

Bandwidth Shaping: Traffic Control Per Service

Linux's `tc` subsystem is one of the most powerful and least-used features on most servers. SwarmK exposes it through the dashboard. You configure egress rate, egress ceiling, ingress rate, and ingress burst per service. The daemon translates those settings into `tc qdisc` and `tc class` commands, applied to the network interfaces within each container's namespace.

This matters most in multi-tenant environments where one service should never be able to saturate the uplink at the expense of everyone else on the node. Give your log aggregation service a 50mbit egress ceiling and it cannot physically exceed that, no matter how much data it tries to flush.

Authoritative DNS with BIND9

Docker's internal DNS resolver (the one at `127.0.0.11:53`) does service discovery within overlay networks. It works fine for resolving service names within the same overlay network. But it's completely uncontrollable: no custom zones, no filtering

rules, no records you define, and no visibility into what other machines on your network are reachable at.

SwarmK installs BIND9 directly on the host machine, not in a Docker container. This is a deliberate architectural choice. When BIND9 runs on the host itself, it can resolve both Docker services and anything else reachable on the server's internal network: other VPS machines, databases on bare metal, internal APIs on private subnets. Docker containers reach it via the host's Docker bridge IP (`172.17.0.1`). When SwarmK deploys BIND9, it also reconfigures the Docker daemon to use that address as its DNS resolver, so every container on the node automatically gets BIND9 as its resolver with no changes to your services.

One click in the dashboard triggers the full deployment: install BIND9 on the host, write the initial `named.conf` and zone files, reload the service, and reconfigure Docker. After that you have full authoritative DNS. Create zones with all eight record types (A, AAAA, CNAME, MX, TXT, SRV, NS, PTR), configure SOA parameters and TTLs, and have every change automatically trigger config regeneration and a live reload. Zone serial numbers are auto-incremented on every mutation so secondary servers always detect changes.

This is entirely optional. If you don't need custom DNS, you don't deploy it. But for teams running databases on separate machines, needing stable internal hostnames across services, or wanting DNS-level filtering, it solves a real problem that Docker Swarm has no answer for on its own.

DNS filtering rules add a service-aware layer on top. A rule blocking `*.analytics-provider.com` for your API services, or redirecting `legacy-api.internal` to `api-v2.internal` globally, is a dropdown and a text field in the dashboard. The daemon's DNS resolver intercepts queries and applies allow/block/redirect actions based on glob-style pattern matching, evaluated by priority order.

Import Your DNS Records Directly From Your Provider

Connecting your domain's DNS records to services usually means a lot of manual work: looking up your existing records in Cloudflare or Route53, typing them in somewhere else, hoping you got every one. SwarmK eliminates that entirely.

You connect your DNS provider once, entering your API credentials (stored encrypted with AES-256-GCM). SwarmK supports fifteen providers: Cloudflare, Route53, Hetzner, DigitalOcean, Hostinger, Linode, Vultr, GoDaddy, Namecheap, OVHcloud, Gandi, Google Cloud DNS, Azure DNS, Scaleway, and custom. After connecting, you trigger a sync. SwarmK calls the provider's API, fetches all your domains, and imports them into the platform as managed domains. Every domain you own shows up in your dashboard automatically.

From there you can map any domain or subdomain directly to a Docker Swarm service through the proxy domain system, set up SSL, configure routing rules, and enable WAF, all without typing a single record name. The link between "this DNS record" and "this service" is made in the UI, not reconstructed from memory.

This is the kind of workflow that saves a surprising amount of time in practice, especially when you're managing more than a handful of services across multiple domains.

What This Means Compared to Kubernetes

Kubernetes is often chosen over Docker Swarm specifically because of networking. NetworkPolicy for traffic control. Service mesh (Istio, Linkerd, Cilium) for observability, encryption, and fine-grained routing. Custom CNI plugins for policy enforcement. ExternalDNS for DNS management. Separate load balancer controllers. The combined result is powerful, and the combined complexity is enormous.

SwarmK delivers the equivalent capabilities on Docker Swarm, without the operational overhead of running Kubernetes itself. Network policies with dynamic group resolution. Traffic flow observability at the connection level. WireGuard encryption between service groups. Configurable load balancing algorithms. Kernel-level rate limiting and bandwidth shaping. Authoritative DNS with filtering.

All of it configured through a single dashboard. All of it enforced by the daemon running on each node. All of it active within seconds of configuration. And all of it running on infrastructure that's an order of magnitude simpler to operate than a production Kubernetes cluster.

TOTAL NGINX CONTROL, THROUGH THE GUI

Nobody has done what SwarmK does with Nginx. That's not marketing language, it's just accurate. Every other tool that puts a UI in front of Nginx either exposes a handful of settings and hides the rest, or eventually tells you to go edit the config file yourself. SwarmK does neither. You can configure every meaningful Nginx behavior, from global server settings down to individual location blocks on individual routes, through the dashboard. No config files. No CLI. No SSH.

Here's how the system is structured.

Web Servers

A web server in SwarmK is a managed Nginx instance deployed as a Docker Swarm service. You control its deployment mode: global (one Nginx process per node in your cluster) or replicated (a fixed number of replicas). You control how it binds ports: host mode preserves the real client IP, ingress mode enables Docker's built-in load balancing across replicas. You control which overlay networks it attaches to, which volumes it mounts, and how many replicas it runs. All of this is configured through the dashboard and applied via agent command, no terminal required.

Each web server has its own Nginx config that SwarmK generates and manages. When you make a change, SwarmK regenerates the config, creates a new Docker config object with a version number, and performs a rolling update of the Nginx service. Your proxy stays running throughout.

Domains and Routing

Proxy domains map hostnames to upstream Docker Swarm services. You define routes: path prefixes, target services, ports. Multiple routes per domain let you run multi-service routing on a single hostname, `/api` to your backend, `/` to your frontend, all in one binding. Each domain can be assigned its own WAF profile, its own proxy policy, and its own SSL certificate independently of every other domain on the same web server.

SSL Certificates

Let's Encrypt certificates are provisioned automatically. SwarmK monitors expiry across all your domains and renews 30 days before expiration without any manual intervention. Custom certificates are also supported: upload your PEM cert and key once, SwarmK stores the key encrypted and strips it from all API responses. Private keys never leave the platform in plain form.

Proxy Policies: Server-Level Nginx Configuration

This is where SwarmK starts doing things no other tool does. Proxy policies are reusable bundles of Nginx server-block configuration that you create once and apply to any web server or any individual domain.

A policy controls:

Security headers. X-Frame-Options, HSTS, Content-Security-Policy, Permissions-Policy, Referrer-Policy, X-XSS-Protection. Set them once in a policy and every domain using that policy gets them applied automatically in the generated Nginx config.

Compression. Three compression methods, all independently configurable: gzip (levels 1-9), Brotli (levels 1-11), and Zstd (levels 1-22). You choose which MIME types to compress and the minimum response size that triggers compression. The static assets policy, for example, runs Brotli at level 11 and Zstd at level 6 for maximum compression on assets that never change.

Rate limiting. Per-path rate limit zones with configurable keys (`$binary_remote_addr` for per-IP, or custom keys), rates (requests per second, minute, or hour), burst allowances, and nodelay settings. The strict security policy ships with two zones: 5 requests per second globally and 3 per minute specifically on `/login`.

Client limits. Maximum body size (`client_max_body_size`), body read timeout, header timeout, send timeout. The upload-heavy policy sets this to 500MB with a 300-second body timeout. The API backend sets it to 50MB with tighter timeouts.

Custom directives. Any Nginx configuration that isn't covered by the above can be injected as raw directives. This is the escape hatch that ensures there is no Nginx setting you cannot configure. If SwarmK's abstractions don't expose what you need, you write it directly and it lands in the generated config.

SwarmK ships six built-in policies tuned for common application types: Default Web, API Backend, Static/CDN, SPA Frontend, Upload Heavy, and Strict Security. Each has sensible defaults for its use case. You can use them as-is, customize the parts that accept overrides, or create entirely custom policies from scratch.

Location Profiles: Per-Route Nginx Configuration

Proxy policies operate at the server block level. Location profiles go deeper: they control the settings inside individual Nginx location blocks, meaning you can configure different Nginx behavior for different routes on the same domain.

A location profile controls:

- Proxy buffering settings and upstream timeouts
- Proxy cookie handling and `next_upstream` behavior for failover
- Header manipulation: add, remove, or override request and response headers
- Upstream SSL/TLS verification behavior
- Access logging: on or off, with format selection
- CORS: allowed origins, methods, headers, `max_age`
- Cache-Control directives
- Proxy caching: cache keys, TTL, lock behavior, bypass conditions
- Static file serving: root or alias directives
- HTTP Basic Authentication
- IP allow/deny rules
- Redirects and rewrites
- WAF toggle: enable or disable ModSecurity per location
- Subrequest authentication via `auth_request`
- gRPC proxy settings
- Response body substitution
- Custom error pages
- Response rate limiting
- Custom directives, same escape hatch as policies

Three built-in location profiles cover the common cases: Empty (pure passthrough, the default), Static Site (24-hour cache, gzip, access logging off), and API Backend (CORS all origins, rate limit burst 20, WAF on, no-cache, 50MB body). You can create custom profiles for any specific routing requirement.

How Config Generation Works

When you deploy or update a web server, SwarmK iterates through all proxy domains assigned to it, reads their routes, pulls in the applicable proxy policies, location profiles, WAF settings, SSL certificates, and CSP headers, and generates a complete `nginx.conf`. The config is deterministic: same settings always produce the same output.

In Docker Swarm mode, SwarmK creates a Docker config object with a versioned name, dispatches an `nginx.update-config` agent command to the server, and the Nginx service performs a rolling update with zero downtime. In standalone mode, the config is written to a volume and the container is reloaded. The dashboard tracks deployment state the whole time: deploying, updating, running, degraded, error.

SwarmK also selects the right Nginx image tier automatically. If your configuration enables WAF, uses custom SSL certificates, or turns on Brotli or Zstd compression, SwarmK pulls the full image (ModSecurity, certbot, brotli, zstd all compiled in). If none of those are needed, it uses the lighter image. You never think about which image to use.

The Result

The sum of all of this is that SwarmK gives you complete, GUI-driven control over your entire Nginx layer. Security headers, compression, rate limiting, client limits, routing, SSL, WAF, CSP, caching, CORS, auth, rewrites, custom directives, all of it, configured through the dashboard, version-tracked, and deployed automatically. The generated config is authoritative. You never touch it directly.

Traefik, Caddy, and Nginx Ingress controllers expose subsets of this. None of them expose all of it. None of them bundle WAF, CSP scanning, and per-location profile overrides in the same system. None of them auto-select image tiers or handle rolling

config updates automatically. SwarmK is the first tool that treats Nginx configuration as a fully managed, fully observable, fully GUI-controlled layer of your infrastructure.

WEB APPLICATION FIREWALL

The WAF is ModSecurity with OWASP Core Rule Set running at the Nginx layer. SwarmK ships six built-in profiles tuned for different application types: Node.js APIs, PHP/Apache, SQL backends, NoSQL backends, generic web apps, and static assets.

Paranoia levels run from 1 (permissive) to 4 (paranoid). Rule groups can be selectively enabled or disabled, if you need SQL injection protection but XSS rules are causing false positives for your specific application, you can tune that without touching a config file.

The WAF events log stores ModSecurity events for 30 days with full detail: transaction ID, rule ID, severity (critical/error/warning/notice), matched data, and the source IP. The dashboard aggregates this into a view showing your top triggered rules, severity breakdown, and most active offending IPs.

AUTOMATIC CSP GENERATION

Content Security Policy headers are one of those things every security-conscious team knows they should have and almost nobody gets right. Writing a CSP manually means auditing every external script, stylesheet, font, image, and API your app loads, figuring out which CSP directive each one belongs to, keeping that list up to date as your app changes, and testing that you haven't accidentally blocked something legitimate. It's tedious, it's error-prone, and it almost always gets skipped.

SwarmK automates the entire thing.

When you trigger a CSP scan on a domain, SwarmK sends an agent command to the server running that service. The agent scans the container's filesystem directly, walking through your built application files: HTML, CSS, JavaScript, static assets. It extracts every external URL referenced via `src` and `href` attributes, CSS `url()` and

`@import` declarations, and `https://` references in JavaScript. It also detects whether your application uses inline scripts or inline styles, both of which affect CSP policy.

After collection, SwarmK classifies every discovered URL into the correct CSP directive automatically. Well-known services get mapped precisely: `fonts.googleapis.com` goes to `style-src`, `fonts.gstatic.com` to `font-src`, `cdn.jsdelivr.net` to `script-src`, Sentry to `connect-src`, Stripe and YouTube to `frame-src`. File extensions handle the rest: `.js` files become `script-src`, `.css` becomes `style-src`, `.woff2` becomes `font-src`, and so on. API paths and WebSocket endpoints get routed to `connect-src`.

The result is a complete, generated CSP header that accurately reflects what your application actually loads.

From there you can review the scan in the dashboard, add manual overrides for anything the scanner missed (additive, not replacing the auto-detected values), approve the result, and choose your enforcement mode: `report_only` (CSP violations are logged but not blocked, good for initial rollout), or `enforce` (violations are actively blocked). The moment you approve and set a mode, SwarmK regenerates the Nginx config and pushes it live. The CSP header appears in your responses automatically. No config files to edit, no Nginx restarts to coordinate, no YAML to write.

If your app ships a new version that pulls in a new CDN or adds a new third-party script, you re-run the scan, review the diff, and approve. Two clicks and your CSP is current again.

This is one of those features that sounds simple until you've tried to maintain CSP headers on a real production app. SwarmK makes it something you can actually keep up with.

LOAD BALANCING

Six algorithms: round-robin, least-conn (routes to the replica with fewest active connections), ip-hash (consistent routing per client IP), random, weighted (assign weights per replica), and sticky (session affinity). Health checks monitor each upstream

with configurable path, interval, timeout, and failure threshold before marking a replica unhealthy.

All of this is enforced at the nftables level, kernel-level packet routing, not application-level proxying. The performance difference matters at scale.

RATE LIMITING AND BANDWIDTH THROTTLING

Rate limiting is per-service, with global (all clients) or per-source (per-IP) scope. You configure the rate, unit (requests per second/minute/hour), and burst allowance. Enforcement is at the kernel level via nftables.

Bandwidth throttling uses Linux traffic control to shape egress and ingress per service. Set a rate and ceiling, and SwarmK enforces it. Useful for multi-tenant environments where one service should never be able to saturate your uplink at the expense of everything else.

DATABASES

SwarmK provisions and manages MongoDB and PostgreSQL instances as Docker Swarm services. Size presets (small, medium, large) configure appropriate memory, CPU, and cache settings for each engine.

Database operations, deploy, scale, stop, start, delete, all go through the agent command system. Backups are integrated: S3-compatible endpoints, configurable cron schedules, retention periods, and for MongoDB, oplog-based point-in-time recovery.

Restoring from a backup is a first-class operation, not a script you have to write yourself.

LOGS AND OBSERVABILITY

Every container, service, daemon, and system on every managed node ships logs back to SwarmK. They're stored with a 7-day retention window and indexed for full-text search. You can filter by log level (debug, info, warn, error, fatal with threshold semantics), source type, time range, and cursor-based pagination through large result sets.

Log archives (Cloud edition) let you export filtered log sets to S3 in NDJSON format for longer-term retention. The archive system has a proper state machine: pending → archiving → archived → retrieving → available.

This is a significant difference from tools that just help you see running containers. SwarmK gives you infrastructure-wide log access through a single interface, searchable, filterable, and retainable.

BACKUPS

The backup system supports nine destination types: NFS, SMB, SFTP, S3, local storage, Dropbox, Google Cloud Storage, Azure Blob, and Backblaze B2. Schedules use cron expressions. Retention is configurable. Compression options include gzip, zstd, and none.

Backup jobs track their execution: running, transferring, completed, failed, with size, duration, and error messages when things go wrong. Manual backups can be triggered on-demand alongside the scheduled runs.

ALERTS, WEBHOOKS, AND NOTIFICATIONS

The alert system monitors twelve metric types: traffic volume, latency, error rate, connection count, container crashes, service stop/start events, volume usage, node going offline, certificate expiry, backup failures, and image prune events. Operators, comparison operators (gt, gte, lt, lte, eq), thresholds, and cooldown periods are all configurable.

When an alert fires, it dispatches to notification channels: email (severity-colored HTML), Slack (Block Kit formatted), and webhooks. Alerts move through states: open → acknowledged → resolved. Each transition fires a webhook.

The webhook system itself is full-featured: HMAC-SHA256 payload signing, three retries with exponential backoff, auto-disable after five consecutive failures, and 7-day delivery records. Format options include JSON, Slack, Discord, and PagerDuty.

AUDIT LOGS

Every write operation in SwarmK, every policy change, every service deployment, every user added or removed, is recorded in the audit log. Append-only. Never modified. Never deleted via the API.

The audit trail includes the action, the resource type and ID, the user who made the change, and a timestamp. You can query it by action, resource type, user, or time range.

For teams operating under compliance requirements, or for anyone who's ever had to answer "who changed that firewall rule on Thursday?", this is not a nice-to-have.

USER MANAGEMENT AND ACCESS CONTROL

SwarmK uses four roles: Owner (full control), Admin (create and delete resources), Operator (manage running resources), Viewer (read-only).

Operator scoping is a standout feature: operators can be assigned to specific network groups, which limits their management scope to only the policies and groups that apply to their assigned resources. This enables genuinely decentralized infrastructure management in multi-team organizations, teams self-govern their own network zone without being able to touch anything else.

The Cloud edition adds user groups for RBAC at scale, OIDC integration for external identity providers, and LDAP support for enterprise directories.

ENVIRONMENT VARIABLES DONE RIGHT

Environment variables are one of those things that sounds trivial and becomes a genuine operational headache at scale. You end up with the same `DATABASE_URL`, `REDIS_HOST`, and `NODE_ENV` copy-pasted into a dozen services. Then someone changes the database password and has to hunt through every service config to update it. Or a new developer deploys a service and forgets three variables because they weren't documented anywhere obvious.

SwarmK solves this with environment variable sets. A set is a named, reusable collection of key-value pairs. You create a set called "production-database" with your database connection variables. You create another called "shared-config" with your application-wide settings. When you deploy a service, you select which sets to apply to it.

The merge system is where it gets genuinely useful. You can attach multiple sets to a single service, and SwarmK merges them left-to-right: if two sets define the same key, the later one wins. On top of that you can add ad-hoc overrides for anything service-specific. Before you deploy, the merge preview shows you the exact final environment the service will receive, variable by variable, with no guessing about which value came from which set.

When you update a set, every service using it picks up the change on its next deployment. You maintain variables in one place. You don't hunt for copies. You don't miss one. This is available in both the OSS and Cloud editions.

For secrets that genuinely need to stay out of environment variables, SwarmK manages Docker secrets and configs through the agent command system. Operations target swarm manager nodes, return command IDs for async tracking, and are fully auditable through the audit log.

All provider credentials (hosting APIs, DNS providers, container registry tokens) are encrypted at rest with AES-256-GCM and masked in every API response. They go in once and never appear in plain text again.

CONFIG IMPORT AND EXPORT

Network policies, groups, load balancer configs, DNS rules, rate limit configs, and bandwidth configs can be exported as a JSON snapshot and imported to another SwarmK instance. This enables staging-to-production config promotion, disaster recovery workflows, and template-based infrastructure provisioning.

THE GLOBAL SEARCH

A small feature that saves real time: full-text search across all major collections, agents, services, policies, groups, domains, servers, databases, from a single search bar. No more navigating through nested menus trying to remember where that one thing was.

IP BLOCKLIST AND THREAT INTELLIGENCE

Most infrastructure runs on the open internet. Bots, scrapers, and automated attack tools are a constant background noise. Managing blocklists manually is a losing game: by the time you've added an IP, the attacker has moved to another one.

SwarmK connects to the IPsum threat intelligence feed, a daily-updated database that scores IP addresses from 1 to 10 based on observed malicious behavior across the internet. You configure a threshold score. Any IP that meets or exceeds that threshold gets blocked automatically, enforced at the nftables level, before a request ever reaches your application.

This is not a proxy-based solution. The blocking happens in the Linux kernel, on every managed node, using native nftables set operations. The performance overhead at scale is negligible.

You can also manage your own blocklist: individual IPs, CIDR ranges, or bulk imported lists. Manual entries sit alongside the automatic threat feed entries and are enforced the same way. Block durations are configurable, temporary for transient threats and permanent for known bad actors.

The dashboard shows the current state: total blocked IPs, how many were auto-blocked from the threat feed, how many were added manually, when the feed was last updated, and how many rule insertions the last update triggered. A complete,

automated IP reputation system running natively in your kernel with no additional services required.

ENTERPRISE SSO: OIDC AND LDAP

For teams operating under enterprise security requirements, SwarmK's authentication doesn't stop at username and password. The Cloud edition supports full enterprise single sign-on through two protocols.

OIDC integration connects SwarmK to any compatible identity provider: Okta, Auth0, Azure AD, Google Workspace, Keycloak, or any custom OIDC provider. The implementation uses PKCE for security, supports configurable claim fields for email, name, and role extraction, and can auto-provision new user accounts the first time someone authenticates. You configure a role mapping, which external role names map to which SwarmK roles, and users get the right permissions from the moment they log in, without any manual user creation.

LDAP integration adds support for Active Directory and other directory services. Group sync maps LDAP groups to SwarmK roles on every login, so when an employee's group membership changes in your directory, their SwarmK permissions update automatically. Just-in-time provisioning means you manage the user database in one place and SwarmK reflects it without manual intervention.

Both integrations are company-scoped, so your SSO configuration applies to your company's users without affecting anyone else. You can run multiple OIDC providers simultaneously, useful for organizations with separate identity providers for different teams.

CONTAINER CONSOLE: DIRECT SHELL ACCESS

Click the console icon on any running container row in the dashboard. A terminal panel slides up and you're inside a live shell session in that container. No SSH. No docker exec from a terminal. No port forwarding. Just a browser-based terminal connected directly to the container's shell.

The underlying implementation routes through SwarmK's WebSocket infrastructure. The dashboard runs xterm.js. A WebSocket connection goes to the WS Gateway, which relays to the agent on the relevant node, which creates a Docker exec instance with a TTY and stdin attached. Characters you type go upstream base64-encoded. Output comes back the same way. Terminal resize events get forwarded to Docker's exec resize API, so your terminal width is actually correct.

You can choose which shell to open: `/bin/sh` by default, `/bin/bash` if available, or a custom command. Sessions have a 30-minute inactivity timeout. Concurrent sessions are limited to five per company to prevent resource abuse.

One hard rule: this requires Admin role or higher. This is remote code execution and SwarmK treats it that way. Every session is audit-logged with the container, the user, and the timestamp.

For debugging, for one-off operations on a running container, for situations where you'd otherwise need to SSH into the host and run docker exec yourself, this removes several steps and the need for any terminal at all.

BUILT-IN DATABASE BROWSER

If you're running MongoDB instances through SwarmK, the database viewer lets you browse them directly from the dashboard.

Connect a database by adding its connection URI, stored encrypted with AES-256-GCM and masked in all API responses after initial entry. Once connected, the viewer shows the databases and collections inside, collection-level stats (document count, storage size, index count), and lets you query documents directly.

The viewer is deliberately read-only. You can inspect data, understand schema patterns, and verify what's actually in a collection, but you cannot write through it. For production database debugging, where the last thing you want is accidental write operations, this is the right boundary.

Database connections are company-scoped. Only admins and operators with the right permissions can add or access connections. The URI never appears in plain text after the initial save.

IMAGE MANAGEMENT WITH PROTECTION RULES

Docker images accumulate quietly. Pull a new version of a service, and the old image stays on the host. Over time, a node that started with plenty of disk space finds itself running low because dozens of previous image layers are sitting around unused.

SwarmK gives you image management from the dashboard. You can see all images on each agent: repository, tag, size, creation date, and which containers are currently using them. You can pull new images and remove unused ones without touching the command line.

The protection rule system is the part that matters most in practice. You define rules that prevent specific images from being pruned: by repository prefix, by tag pattern, by age threshold. An image matching a protection rule will never be removed by an auto-prune operation.

Auto-prune runs on a configurable schedule and removes unprotected images that meet your criteria. Age-based pruning removes images older than a threshold. Size-based pruning removes images to reclaim a target amount of disk space, oldest-first. Protection rules run first, so the images you care about are never touched.

On a fleet of ten nodes, unmanaged image accumulation is a real operational problem. Managed image pruning with protection rules makes it automatic and safe.

LVM AND STORAGE MANAGEMENT

No other Docker management GUI has this. Not Portainer. Not Dokploy. Not anything else in this space.

SwarmK gives you direct Logical Volume Manager access from the dashboard. For teams running databases, high-throughput services, or any workload where storage performance and flexibility matter, this is significant.

Through the SwarmK interface you can inspect the physical volumes, volume groups, and logical volumes on each managed node. You can create new logical volumes, allocate space from an existing volume group, and assign them as bind mounts to your

services. You can resize logical volumes to grow storage without stopping services. You can take snapshots, point-in-time copies of a logical volume that capture its state at that exact instant, useful for database backups, pre-migration safety copies, or staged testing with production data.

The practical impact: a database running on an LVM logical volume can be snapshotted before a migration, resized when it outgrows its allocation, and restored from snapshot if something goes wrong. All of this is done from the SwarmK dashboard, without SSH, without running lvcreate and lvextend by hand, and with a full audit trail of every storage operation.

LVM access is available on nodes where the SwarmK daemon has the necessary host privileges. For teams running serious stateful workloads, the combination of service management, database management, and direct LVM control in a single platform means you're not working around tooling gaps at every turn.

CRON JOBS

Docker Swarm has no native equivalent of a Kubernetes CronJob. If you want a scheduled task in a pure Swarm setup, you typically set up a cron entry on the host, write a shell script that runs a docker command, and hope it keeps working when the container gets replaced or the host is rebuilt.

SwarmK adds proper cron job management. You create a cron job with a standard cron expression, a Docker image, a command, environment variable sets, resource limits, and a concurrency policy: skip the run if the previous one is still executing, or replace it. SwarmK schedules runs, dispatches them as agent commands, captures exit codes and output, and stores the full execution history.

From the dashboard you can see every past run: what time it executed, how long it took, whether it succeeded or failed, and the output from the container if you need to debug a failure. You can trigger a manual run at any time without waiting for the schedule. You can pause a job without deleting it and resume it later.

The integration with environment variable sets is the same system used for services, so you can reference your shared secrets and configuration variables in cron jobs without duplicating them.

DOCKER COMPOSE IMPORT

Most teams have existing stacks defined as `docker-compose.yml` files. Migrating to a new management platform usually means either recreating everything in the new tool's format or managing things manually outside the platform.

SwarmK supports importing Compose files directly. Upload a `docker-compose.yml` and SwarmK parses it on the spot, returning an import preview showing exactly what it found: services, volumes, networks, environment variables. The file is not stored anywhere, it's a stateless parse-and-preview operation.

The interesting part is what happens to database services. SwarmK detects standard database images in your Compose file, PostgreSQL, MongoDB, Redis, and others, and offers to convert them to managed databases rather than raw services. A managed database gets full backup integration, monitoring, the database viewer, and all the operational tooling that comes with SwarmK's managed database system. If you prefer to keep them as raw services, that option is available too.

After reviewing the preview, you deploy to a specific environment. SwarmK creates the services, applies the right network configuration, and starts everything up. What was a manually managed Compose stack becomes a fully managed SwarmK deployment.

REGISTRY WEBHOOKS: PUSH-TO-DEPLOY

Connect your container registry and SwarmK can deploy automatically when you push a new image. Build your image in CI, push to your registry, and SwarmK handles the rest.

The integration uses inbound webhooks signed with HMAC. Your registry posts to SwarmK's webhook endpoint. SwarmK verifies the signature, matches the pushed image against your configured rules using micromatch glob pattern matching, and dispatches a deployment command if there's a match. Pattern flexibility means you can deploy on every push to `main-*` tags while ignoring feature branch builds, or trigger only when the `production` tag is pushed.

Each webhook rule specifies which service to update, which environment to target, and optionally which tag to pull. The deployment goes through the standard agent command system, so it's auditable and visible from the dashboard. You can see every push event, whether it matched a rule, and whether the resulting deployment succeeded or failed.

This is the deployment automation glue between your CI pipeline and your running infrastructure. Push code, build image, push to registry, SwarmK deploys. No additional CI steps, no deploy scripts, no manual action required.

PROMETHEUS METRICS EXPORT

If you're running Prometheus for metrics collection and alerting, SwarmK integrates without any configuration on your end.

The `/metrics` endpoint (at the root level, following Prometheus convention, not under `/api/v1/`) exposes your SwarmK cluster state in Prometheus text exposition format. Prometheus scrapers hit it on whatever interval you configure and collect everything.

What gets exported: per-node daemon status and uptime, per-agent CPU/memory/disk/network metrics using the most recent reading per hostname, cluster-level counts for agents online, services, databases, and SSL certificates, alert counts by severity and state, WAF event counts for the last 24 hours broken down by severity, blocked request totals, and network flow traffic bytes and packets by service pair for the last 5 minutes.

Authentication uses your SwarmK API tokens, not JWTs. Prometheus scrapers need static credentials and API tokens are the right mechanism. The endpoint accepts a Bearer token header or a `?token=` query parameter. Results are cached for 15 seconds per company so parallel scrape requests don't trigger redundant database queries.

This is a Cloud edition feature. If your organization is already running Prometheus and Grafana, SwarmK slots right in.

LIVE ROLLOUT STATUS

When you deploy a service, scale it up, or trigger a rolling update, SwarmK tracks the rollout in real time.

After the Docker API call returns, SwarmK polls the task list for that service every 2 seconds, counting running replicas, starting replicas, and failed replicas. It calculates progress as a percentage and sends updates through the existing WebSocket infrastructure. The dashboard shows a live progress bar on the service card: "2/3 replicas ready," updating in real time until convergence.

Failure detection works correctly. A single replica entering failed state while others are still starting is not treated as a failure, Swarm will retry. But if replicas are failing and nothing is starting, the rollout is marked failed with a clear message.

Rollouts have a 5-minute timeout. If all replicas haven't converged in that window, the operation is marked "completed with warning" rather than failed, because the Docker operation itself succeeded and convergence is just taking longer than expected. The result data records the `timed_out` flag so the distinction is clear.

When everything converges, the progress indicator disappears automatically. Global-mode services skip rollout tracking entirely since they deploy one replica per node with no convergence concept.

The ability to watch a deployment converge in real time, to see "0/3 replicas ready" become "3/3 replicas ready," to know immediately when something goes wrong rather than discovering it by refreshing the services page, changes how you think about deployments.

CONTROL SWARMK FROM CLAUDE: THE MCP SERVER

The `swarmkmcp` package is a Model Context Protocol server that exposes your entire SwarmK infrastructure as tools Claude can call directly. If you use Claude, through the desktop app, VS Code extension, or any MCP-compatible client, you can manage your infrastructure in natural language without ever opening a browser.

What That Actually Looks Like

You open Claude and say: "Deploy the `api` service in the `production` project with the new image I just pushed." Claude calls `swarmk_deploy_service` with the right parameters, gets confirmation from your SwarmK instance, and reports back. You don't navigate to the dashboard. You don't remember which project the service lives in. You describe what you want and it happens.

Or: "What are the last 100 lines of logs from the `worker` service?" Claude calls `swarmk_get_service_logs`, formats the response, and you're reading logs in the same conversation where you were just debugging.

Or: "Spin up a new project called `staging-v2` on the `eu-west-1` server." One tool call. Done.

This is the full developer deployment workflow, provision, deploy, redeploy, inspect, debug, from a conversation.

Setup

`swarmkmcp` authenticates with a SwarmK API token, the same kind of token you'd use for the Prometheus scraper or any other programmatic access. Create one in SwarmK Settings → API Tokens, then add the MCP server to Claude:

```
claude mcp add swarmkmcp -- npx -y swarmkmcp@latest
```

Set two environment variables:

```
SWARMK_API_TOKEN=your_token_here
SWARMK_API_URL=https://your-swarmk-instance.com  # optional, defaults to https:
```

That's the entire setup. No SSH keys to manage, no config files to maintain, no separate service to run. `npx` handles the install on demand.

27 Tools Across the Full Platform

The MCP server exposes 27 tools covering the deployment workflow:

Servers, list, get details, provision new servers, delete servers.

Projects and Services, list projects, get project details, create projects, list services within a project, get service details, create services, deploy or redeploy a service, start and stop services, delete services.

Environment Variables, list env var sets, get the contents of a set (including all key-value pairs), create new sets, update variables in an existing set.

Logs, fetch recent log output from any service, with optional line count and timestamp filters.

Domains and SSL, list domains, add a domain to a service, remove a domain, list SSL certificates, issue a new SSL cert for a domain.

Docker and Nodes, list Docker images on a server, list containers, list Swarm nodes per server.

Every tool returns plain text that Claude presents directly in the conversation. Error messages are specific: a bad API token gets "Authentication failed. Check your SWARMK_API_TOKEN." A missing service gets the actual error from the SwarmK API. A rate limit gets a clear message asking you to wait. Nothing swallows errors silently.

Why This Matters

Most infrastructure operations require you to context-switch. You're in your editor, something needs deploying, you open a browser, navigate to the right page, click the right buttons, go back to your editor. Each switch is friction.

The MCP server eliminates that context switch entirely. If Claude is already in your workflow, reviewing code, helping debug, explaining a stack trace, it can also trigger the deployment. The same conversation that diagnosed the problem can fix it.

It also makes SwarmK accessible to developers who aren't infrastructure specialists. Someone on the team who knows their service name and what they want to deploy doesn't need to understand the full SwarmK dashboard hierarchy to do it. They describe it to Claude and the right API calls happen.

For teams already using AI coding assistants, `swarmkmcp` means your infrastructure has the same conversational interface as your codebase.

HOW SWARMK COMPARES TO PORTAINER

Portainer is a Docker management UI. It's good at what it does: container lifecycle management, stack deployment, a visual interface over the Docker API. If that's all you need, Portainer does it well.

But there are two practical limitations that matter a lot for real-world usage. First, to manage services in Portainer you typically need to define and deploy them through Portainer itself. If you have an existing setup, stacks you deployed manually, services already running, you're working against the grain. SwarmK discovers what's already there and manages it immediately, no migration required.

Second, Portainer's feature scope stops at Docker operations. It doesn't have a network policy engine. It doesn't manage WireGuard tunnels or run a WAF or handle your DNS zones. It doesn't have an alert evaluator polling 12 metric types or a webhook system with HMAC signing. The audit log in Portainer covers Docker operations; SwarmK's audit log covers your entire infrastructure control plane.

If you're making a direct comparison: Portainer manages containers. SwarmK manages infrastructure, and works with the containers you already have.

HOW SWARMK COMPARES TO DOKPLOY

Dokploy is a deployment platform. It handles CI/CD workflows, manages application deployments, and has good developer experience for shipping code. It's excellent at application delivery. But like Portainer, Dokploy expects you to define your services

through its interface to bring them under management. Services deployed outside of Dokploy exist outside its model.

SwarmK doesn't have that boundary. Install the agent and SwarmK maps everything, services you deployed yesterday, stacks you've been running for a year, Swarm nodes you configured by hand. It meets your infrastructure where it already is.

SwarmK is also focused on a different layer: what happens after code is deployed. How services communicate with each other, what traffic policies govern them, how they're secured, how their infrastructure is monitored and maintained. The two tools solve adjacent problems, but SwarmK's agent model means you can lock down SSH completely after initial setup, something Dokploy still requires for most operations.

AUTOSCALING AND SERVER UPGRADES

This is another capability that simply does not exist in any other Docker Swarm GUI, and it's worth taking a moment to understand what it actually does.

SwarmK can automatically scale your infrastructure in response to real metrics. You define an autoscale policy: which service to watch, which metric to track (CPU or memory), what the thresholds are, and how to respond. When your service hits the scale-up threshold, SwarmK acts. When it drops back below the scale-down threshold, SwarmK scales back down. Cooldown windows prevent thrashing. Every decision is logged as an autoscale event with the metric value, the threshold that triggered it, and the exact action taken.

What makes SwarmK's autoscaling different is that it supports two fundamentally different types of scaling, and they can work together.

Horizontal scaling is the familiar kind: add more replicas when load increases, remove them when load drops. You set a minimum and maximum replica count and SwarmK handles the rest, dispatching scale commands to your agent and tracking the outcome.

Vertical scaling is where things get genuinely impressive. If your service hits maximum replicas and the metric is still above threshold, or if a CPU or memory spike is severe enough to warrant it independently, SwarmK can upgrade the underlying VPS itself. It

provisions a larger server through your hosting provider, migrates everything over, and optionally destroys the old machine when the transfer is complete. All automatically, all orchestrated by SwarmK, with no manual intervention.

You define the upgrade path explicitly: an ordered list of server plans to step through. SwarmK walks that path one step at a time as load demands it. If your provider supports it, the whole operation happens without you touching anything.

The Server Transfer Engine

Vertical scaling works through SwarmK's server transfer system, which is a full workload migration engine. A transfer moves everything from the source server to the target: web servers, Nginx configuration, databases (with backup, transfer, and restore), projects, DNS records, and credentials. Seventeen ordered steps, each tracked individually with status, timestamps, and error detail if something goes wrong.

The transfer system sets up a temporary DNS bridge during the migration so traffic keeps flowing while DNS propagation catches up. After the migration is complete and DNS has propagated, the old server is destroyed. Zero-downtime infrastructure upgrades, driven entirely from the SwarmK dashboard.

If a transfer fails mid-way, you don't start over. SwarmK retries from the exact step that failed, preserving all the work already completed. You can also cancel a transfer in progress and inspect the preflight inventory before any migration begins, which tells you exactly what would be moved: web servers, databases, Nginx state, DNS state, and projects.

This is the kind of operational capability that normally requires a dedicated platform engineering team to build and maintain. In SwarmK it's a configured autoscale policy, or a few clicks in the dashboard.

PRICING

Your first agent is free, forever. No trial period, no expiry. If you're running a single server, a homelab, or just want to evaluate SwarmK properly in a real environment, you never pay anything.

Each additional agent is $9.99 per month. That's it.

THE BETA

SwarmK's feature set is deep. This article has covered the major systems, but there's more underneath: CSP scanning, proxy policies, location profiles, container console access, Docker image management, volume browsing with LVM support, cron job management, Compose import, registry webhooks, Prometheus export endpoints, rollout status tracking, server templates for repeatable VPS provisioning, detached resource cleanup, server transfers between providers.

This is a lot of surface area. Which means there are rough edges. We know it. That's what a beta is for.

What we're looking for are teams willing to actually run SwarmK on real infrastructure and tell us where it breaks, what doesn't make sense, what they need that isn't there. Not theoretical feedback, real feedback from real usage.

If you run Docker Swarm in production, or you've been meaning to but the operational complexity kept you away, SwarmK is worth trying. Your first agent is free. The control plane is easy to spin up. And the team is actively watching for feedback.

We built SwarmK because we wanted this tool and it didn't exist. We think a lot of you have been in the same situation. Help us make it the thing we all actually want.

SwarmK is available now in open beta. First agent free, no time limit.