Docker Swarm vs Kubernetes in 2026

10 Years, 24 Containers, Two Continents, Zero Crashes

FEBRUARY 15, 2026

THEDECIPHERIST.COM

TABLE OF CONTENTS

10 YEARS, 24 CONTAINERS, TWO CONTINENTS, ZERO CRASHES — AND WHY THE INDUSTRY GOT IT WRONG

Join the discussion at r/docker_dev

TL;DR: I've run Docker Swarm in production for 10 years — starting with multi-node clusters running 4-6 replicas per service, optimized down to 24 containers across two continents, zero crashes, $166/year total. Kubernetes solves real problems for the 1% who need it. The other 99% are paying a massive complexity tax for capabilities they never use, while 87% of their provisioned CPU sits idle. The only feature K8s has that Swarm genuinely lacks is autoscaling — and half of K8s users don't even use it. This article includes a working autoscaler script that's actually smarter than K8s HPA, side-by-side YAML comparisons (27 lines vs 170+), and a cost breakdown that should make any CTO uncomfortable.

I wrote this article because I'm genuinely baffled by what I see online. Every week on Reddit, Hacker News, and dev forums, someone asks about Docker Swarm and the responses are the same: "Swarm is dead." "Just use K8s." "Nobody runs Swarm in production." "You'll have to migrate eventually." These aren't informed opinions — they're reflexes. People who've never run Swarm in production confidently telling others to avoid it, while recommending a system that wastes 87% of its provisioned CPU and costs 10-100x more to operate.

I've run Swarm in production for a decade. Not as a side project on a single VPS — as a real multi-node cluster. Three servers per cluster, manager redundancy, services running 4-6 replicas, proper rolling deployments, the full setup. When I pushed a new image, Swarm rolled it out in sets of two — updating two replicas at a time while the others kept serving traffic. If the new containers failed their healthchecks, Swarm automatically rolled back to the previous version. Customers never experienced downtime. No blue-green deployment tooling. No Argo Rollouts. Just Swarm's built-in `update_config` and `rollback_config` doing exactly what they're supposed to do. Over the years I optimized the architecture and code to the point where the entire platform now runs on two $83/year VPS instances across two continents — not because Swarm can't handle more, but because efficient code doesn't need more. And I'm tired of watching developers get talked into complexity they don't need by people who profit from that complexity.

This is the article I wish existed when I made my choice ten years ago.

Let me be clear upfront: Kubernetes is not a bad system. It is an incredibly powerful, well-engineered piece of software built by Google to solve Google-scale problems. If you need granular control over every tiny aspect of your container orchestration — network policies, pod scheduling, resource quotas, multi-tenant isolation, custom admission controllers, autoscaling on custom metrics — Kubernetes gives you knobs for all of it.

The problem is that 99% of teams don't need any of those knobs. And the 1% that do are paying a staggering tax for the privilege.

Ten years ago, I evaluated both Kubernetes and Docker Swarm, chose Swarm, and never migrated. Today I run a 24-container, dual-continent production infrastructure — including a live SaaS platform processing constant real-time data — on two $83/year VPS instances. Zero container crashes. Zero data loss. Zero security breaches. Disaster recovery in under 10 minutes. Server CPU at 0.3%.

This isn't a theoretical comparison. This is a decade of production receipts versus an industry that collectively chose the spaceship to go to the grocery store.

VHS VS BETA: THE PERFECT ANALOGY

In the 1980s, Sony's Betamax was technically superior to VHS in almost every way — better picture quality, better sound, better build quality. Beta lost the format war for one practical reason: VHS could record longer, which mattered to Americans who wanted to tape a three-hour football game with commercials. That single practical feature — recording length — outweighed every technical advantage Beta had.

Kubernetes vs Docker Swarm is the same story, except even more absurd.

K8s won the orchestration market not because 92% of teams needed it, but because Google open-sourced it, every cloud provider built a managed service around it (recurring revenue), a certification industry emerged, and suddenly it appeared on every job description. The ecosystem made money off the complexity.

And just like VHS vs Beta, most users can't tell the difference in their actual daily usage. The team running 10 services on K8s has the same outcome as me running 24 containers on Swarm. Containers serve traffic. Users don't know or care what orchestrator is behind it.

But here's where the analogy gets even better: VHS and Beta were at least different formats. K8s and Swarm are orchestrating the exact same Docker containers. It's like if VHS and Beta both played the same tape, but Beta required you to buy a separate $200 tape-loading robot that needed its own power supply, firmware updates, and a technician on call — while VHS just played the tape when you pushed it in.

WAIT — DOCKER AND KUBERNETES AREN'T EVEN COMPETITORS

Before we go further, let's clear up the most common confusion in the industry.

Docker owns the container *creation* market — about 88% market share. It builds images, creates containers, runs them. When someone says "I use Docker," they mean they package and run apps in containers.

Kubernetes owns the container *orchestration* market — about 92% share. It doesn't build containers or run them. It tells Docker (or containerd) where and when to run them across a cluster.

Docker Swarm is Docker's built-in orchestration layer. So the real competition isn't "Docker vs Kubernetes." It's "Docker Swarm vs Kubernetes." Both use Docker containers underneath.

When you choose Swarm, you're not rejecting Docker — you're running Docker either way. You're just choosing Docker's own orchestration over Google's. And the 92% of people using K8s for orchestration? They're also running Docker containers. They just added an entire separate system on top to manage them.

Which goes right back to the core problem: K8s rebuilt everything that already existed in Docker and Linux, then the industry charged you for the privilege of managing it.

THE 80% PROBLEM NOBODY TALKS ABOUT

If you already know Docker — and most developers do — you already know 80% of Swarm. Same compose files. Same networking concepts. Same CLI. Same images. Swarm is just Docker with a few extra commands. You write your `docker-compose.yml` for local development, and `docker stack deploy` runs essentially the same file in production. One config system. One mental model. One source of truth.

Kubernetes asks you to learn an entirely new mental model. Pods, Deployments, StatefulSets, DaemonSets, Ingress controllers, Helm charts, kustomize, etcd, kubectl — a whole new YAML dialect that somehow makes Docker's YAML look simple.

And at the end of all that learning curve, your containers still run the same way they did in Swarm.

With K8s, you're also maintaining two completely separate configuration systems: Docker Compose for local dev, Helm charts or Kubernetes manifests for production. That's double the surface area for bugs, drift, and misconfiguration. Every discrepancy between your local and production configs is a potential incident waiting to happen.

Don't take my word for it. Here's the same application — a Node.js API with MongoDB — deployed three ways.

Docker Compose (what you already know)

```yaml
services:
  api:
    image: myapp/api:latest
    ports:
      - "3000:3000"
    environment:
      - MONGO_URI=mongodb://mongo:27017/mydb
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:3000/health"]
      interval: 30s
      timeout: 10s
      retries: 3
    depends_on:
      - mongo
    networks:
      - app-network

  mongo:
    image: mongo:7
    volumes:
      - mongo-data:/data/db
    networks:
      - app-network

volumes:
  mongo-data:

networks:
  app-network:
```

27 lines. Zero new concepts. You wrote this.

Docker Swarm

```yaml
services:
  api:
    image: myapp/api:latest
    ports:
      - "3000:3000"
    environment:
      - MONGO_URI=mongodb://mongo:27017/mydb
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:3000/health"]
      interval: 30s
      timeout: 10s
      retries: 3
    depends_on:
      - mongo
    networks:
      - app-network
    deploy:                          # ← new
      replicas: 2                    # ← new
      resources:                     # ← new
        limits:                      # ← new
          memory: 128M               # ← new
      restart_policy:                # ← new
        condition: on-failure        # ← new
      update_config:                 # ← new
        parallelism: 1               # ← new
        delay: 10s                   # ← new

  mongo:
    image: mongo:7
    volumes:
      - mongo-data:/data/db
    networks:
      - app-network
    deploy:                          # ← new
      placement:                     # ← new
        constraints:                 # ← new
          - node.role == manager     # ← new

volumes:
  mongo-data:

networks:
  app-network:
    driver: overlay                  # ← new
```

42 lines. 5 new concepts — `deploy`, `replicas`, `resources`, `placement`, `overlay`. Everything else is identical to what you already write. Deploy with one command: `docker stack deploy -c docker-compose.yml myapp`.

Kubernetes

Same application. You now need a minimum of **4 separate files**.

File 1 — api-deployment.yaml:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api
  labels:
    app: api
spec:
  replicas: 2
  selector:
    matchLabels:
      app: api
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  template:
    metadata:
      labels:
        app: api
    spec:
      containers:
        - name: api
          image: myapp/api:latest
          ports:
            - containerPort: 3000
          env:
            - name: MONGO_URI
              value: "mongodb://mongo:27017/mydb"
          resources:
            requests:
              memory: "64Mi"
              cpu: "50m"
            limits:
              memory: "128Mi"
              cpu: "200m"
          livenessProbe:
            httpGet:
              path: /health
              port: 3000
            initialDelaySeconds: 15
            periodSeconds: 30
            timeoutSeconds: 10
            failureThreshold: 3
          readinessProbe:
            httpGet:
              path: /health
              port: 3000
            initialDelaySeconds: 5
            periodSeconds: 10
```

File 2 — api-service.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: api
spec:
  selector:
    app: api
  ports:
    - protocol: TCP
      port: 3000
      targetPort: 3000
  type: ClusterIP
```

File 3 — mongo-statefulset.yaml:

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mongo
spec:
  serviceName: "mongo"
  replicas: 1
  selector:
    matchLabels:
      app: mongo
  template:
    metadata:
      labels:
        app: mongo
    spec:
      containers:
        - name: mongo
          image: mongo:7
          ports:
            - containerPort: 27017
          volumeMounts:
            - name: mongo-data
              mountPath: /data/db
  volumeClaimTemplates:
    - metadata:
        name: mongo-data
      spec:
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: 10Gi
```

File 4 – mongo-service.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: mongo
spec:
  selector:
    app: mongo
  ports:
    - protocol: TCP
      port: 27017
      targetPort: 27017
  clusterIP: None
```

File 5 — ingress.yaml (if you want it accessible from the internet):

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: api-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
    - host: api.myapp.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: api
                port:
                  number: 3000
```

File 6 — hpa.yaml (if you want autoscaling):

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: api-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: api
  minReplicas: 1
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 70
```

170+ lines. 4–6 files. 25+ new concepts — Deployment, StatefulSet, Pod, Service, Ingress, HPA as resource types. A completely new YAML schema with `apiVersion`, `kind`, `metadata`, `spec`. Label-based wiring with `selector` and `matchLabels`. Two probe types instead of one healthcheck. `requests` vs `limits` for resources. `volumeClaimTemplates` for storage. Three different port concepts (`containerPort` vs `port` vs `targetPort`). Three service types (`ClusterIP`, `NodePort`, `LoadBalancer`). And deploy with six separate commands — or learn Helm, which is yet another templating language on top.

The Scorecard

	COMPOSE	SWARM	KUBERNETES
Files needed	1	1	4–6 minimum
Lines of YAML	27	42	170+
New concepts	0	5	25+
Deploy command	`docker compose up`	`docker stack deploy`	`kubectl apply -f` × 6
Learning time	—	An afternoon	Weeks to months
YAML structure	Same	Same	Completely new

The gap between Compose and Swarm is 15 lines of YAML and an afternoon. The gap between Compose and Kubernetes is 143+ lines, 4–6 files, 25+ new concepts, and weeks of study. If you already run Docker in production — and 88% of teams using containers do — you're 90% of the way to Swarm. You're 0% of the way to Kubernetes.

An Honest Note: What You Must Get Right in Swarm

There's a common assumption that Kubernetes is "smarter" about keeping your services alive. Let's test that.

Swarm's self-healing relies on two things you need to configure correctly: healthchecks and exit codes. These are the most important part of the 10% you need to learn. So the fair question is: does K8s solve these problems for you if you get them wrong?

SCENARIO	DOCKER SWARM	KUBERNETES	WHO WINS?
No healthcheck defined	Container hangs but doesn't crash. Swarm thinks it's fine. Traffic keeps routing to a dead service.	No liveness or readiness probe defined. K8s thinks it's fine. Traffic keeps routing to a dead pod.	**Tie.** Both are blind.
Bad healthcheck endpoint	Healthcheck hits `/health` but the endpoint always returns 200 even when the database is down. Swarm sees "healthy."	Liveness probe hits `/health` but the endpoint always returns 200 even when the database is down. K8s sees "healthy."	**Tie.** Both trust whatever you tell them.
Wrong exit code – `exit(0)` instead of `exit(1)`	App crashes but exits with code 0. Restart policy says "restart on failure." Code 0 = success. Swarm doesn't restart it. Service stays dead.	App crashes but exits with code 0. RestartPolicy says "OnFailure." Code 0 = success. K8s doesn't restart it. Pod stays dead.	**Tie.** Both follow POSIX. 0 = success, period.
App hangs (infinite loop, deadlock, memory leak)	Without a healthcheck, Swarm sees a running container and does nothing. With a proper healthcheck, Swarm kills and replaces it.	Without a liveness probe, K8s sees a running pod and does nothing. With a proper liveness probe, K8s kills and replaces it.	**Tie.** Both need you to define "healthy."
App is alive but not ready (cold cache, warming up)	No built-in concept of "alive but not ready." Healthcheck is pass/fail. You handle warm-up with `start_period`.	Readiness probe stops traffic routing until the pod is ready. Liveness probe handles crashes separately.	**K8s.** Separating "alive" from "ready" is a genuine advantage.
Container keeps crash-looping	Swarm keeps restarting it based on restart policy. No exponential backoff – it just keeps trying.	K8s uses CrashLoopBackOff – exponential delay between restarts (10s, 20s, 40s...) to prevent resource thrashing.	**K8s.** Backoff is smarter than infinite retry.

Score: K8s gets 2 out of 6. Readiness probes and CrashLoopBackOff are real advantages. But for the 4 scenarios that actually kill production services — missing healthchecks, bad healthcheck logic, wrong exit codes, and hanging processes — Kubernetes is exactly as blind as Swarm. Neither system saves you from not understanding your own application.

The difference is Swarm is honest about it: you write a healthcheck, you set your exit codes, and the system does what you tell it. K8s gives you more knobs for the same job — `livenessProbe`, `readinessProbe`, `startupProbe`, each with their own `initialDelaySeconds`, `periodSeconds`, `timeoutSeconds`, `failureThreshold`, `successThreshold` — but if you don't configure them, or configure them wrong, you get the same result.

And here's the uncomfortable truth: most K8s probe configurations in the wild are copy-pasted from StackOverflow or Helm chart defaults. The same developers who wouldn't write a proper healthcheck in Swarm are running `initialDelaySeconds: 15` and `failureThreshold: 3` in K8s because that's what the tutorial said, with no understanding of whether those values match their application's actual behavior. That's not self-healing — it's self-deception with more YAML.

TWO COMMANDS VS. THIRTY

Setting up a Swarm cluster:

```
docker swarm init          # on the manager
docker swarm join ...      # on the worker
```

Two commands. Done. You have a production cluster.

Setting up Kubernetes (self-managed): install kubelet, kubeadm, and kubectl on every node, disable swap (K8s doesn't play nice with it), configure the container runtime, run `kubeadm init` with a config file, set up your kubeconfig, install a CNI network plugin (Calico, Flannel, Cilium — pick one), then join worker nodes with tokens. That's easily 15-30 commands per node, plus config files, plus troubleshooting when something doesn't mesh.

And that's just getting the cluster running. You still need to install an ingress controller, set up persistent storage classes, configure RBAC, possibly set up Helm, and deploy a monitoring stack — because K8s doesn't come with one.

Most teams hit this wall and just pay for managed K8s (EKS, GKE, AKS), which is basically admitting the setup is too complex to do yourself. Then you're locked into their ecosystem and their pricing.

With Swarm, by the time a K8s admin has finished reading the prerequisites doc, you've already deployed your entire stack and gone to lunch.

THE OVERHEAD TAX: AN OPERATING SYSTEM ON TOP OF YOUR OPERATING SYSTEM

Before you deploy a single one of YOUR containers, Kubernetes is already running its own fleet: etcd (1-3 instances), kube-apiserver, kube-scheduler, kube-controller-manager, kube-proxy (on every node), CoreDNS (usually 2 replicas), and CNI plugin pods (on every node). Then most real deployments add an ingress controller, metrics-server, and cert-manager.

That's 10-15 system containers just to have a functioning cluster — before your first application container runs. Kubernetes' own documentation acknowledges that control plane services and system pods typically consume 5-15% of total cluster resources.

Swarm's overhead? Essentially zero. The orchestration is just the Docker daemon doing a little extra work. No separate database, no separate API server, no extra networking plugins. All of your containers are YOUR containers doing YOUR work.

It's like renting a house where K8s takes 5 of the 8 rooms for its own furniture and says "here, you can use what's left." Swarm hands you the keys and says "it's all yours."

And this is what struck me when I first evaluated both: whoever made K8s wanted to remake *everything* themselves. They rebuilt networking, storage, access control, service discovery, DNS, secrets management, config management — all things that already existed in Linux and Docker. They reimplemented the entire operating system inside containers. Which is powerful if you want one API for everything, but it means you're running an OS on top of your OS, and paying the resource tax for both.

THE ONE FEATURE K8S HAS THAT SWARM DOESN'T

Let's be honest about this: Swarm does not have autoscaling. Period.

Swarm will maintain your declared state — if you say 5 replicas and 2 crash, it brings them back to 5. That's self-healing, not autoscaling. Swarm won't watch your CPU hitting 90% and decide on its own to go from 5 replicas to 15.

Kubernetes has Horizontal Pod Autoscaling (HPA) that watches CPU, memory, or custom metrics and adjusts replica counts automatically. Vertical Pod Autoscaling (VPA) resizes resource allocations. Cluster Autoscaler adds or removes entire nodes when pods can't be scheduled. That's a real capability gap.

But here's where it gets interesting.

According to Datadog's 2023 Container Report, only about half of Kubernetes organizations have even adopted HPA. The "killer feature" that justifies all of K8s's complexity? Half the people using K8s don't use it.

And for the teams that don't need autoscaling, adding it to Swarm is a single script — watches CPU and memory, scales replicas, adapts its monitoring speed based on urgency, sends you a webhook when it acts, and handles downed services gracefully. Not as polished as HPA with Prometheus adapters, but for the 99% of teams who just need "scale up when load is high, scale down when it's not," it handles it.

So the only feature K8s has that Swarm genuinely lacks — half the K8s users don't even use. And for those who want it on Swarm, it's a weekend script.

But here's the thing — we can actually make it smarter than HPA. Kubernetes HPA polls metrics on a fixed interval, default every 15 seconds. It checks with the same frequency whether your service is at 2% CPU or 68% CPU. Between checks, it's blind. If your CPU spikes and your service crashes between polls, HPA was asleep.

That's a cron job with a nicer API.

What if instead of checking on a fixed cycle, the script adapted its urgency based on what it sees? Sleep 30 seconds when everything's calm. But when CPU or memory crosses 50% — wake up. Start checking every second. Watch the trajectory. If it hits the threshold, scale immediately. If it drops back down, stand down and go back to sleep. Urgency proportional to customer impact — scale up is urgent, scale down is housekeeping.

Here it is — with proper exit codes, signal handling, and a healthcheck. Because we practice what we preach.

```bash
#!/bin/bash
# Swarm Adaptive Autoscaler
# Smarter than HPA: monitors faster as danger approaches
# Watches CPU and memory — deployed as a proper Swarm service
# With healthchecks, exit codes, and signal handling. Obviously.

set -euo pipefail

SERVICES="${SERVICES:-api,dashboard,website}"
CPU_SCALE_UP="${CPU_SCALE_UP:-70}"
CPU_SCALE_DOWN="${CPU_SCALE_DOWN:-20}"
MEM_SCALE_UP="${MEM_SCALE_UP:-80}"
MEM_SCALE_DOWN="${MEM_SCALE_DOWN:-25}"
ALERT_THRESHOLD="${ALERT_THRESHOLD:-50}"
INTERVAL_NORMAL="${INTERVAL_NORMAL:-30}"
INTERVAL_ALERT="${INTERVAL_ALERT:-1}"
MIN_REPLICAS="${MIN_REPLICAS:-1}"
MAX_REPLICAS="${MAX_REPLICAS:-10}"
COOLDOWN="${COOLDOWN:-120}"
WEBHOOK_URL="${WEBHOOK_URL:-}"
HEALTH_FILE="/tmp/autoscaler-health"

STATE="normal"
RUNNING=true

# --- Signal handling: clean shutdown = exit 0 ---
cleanup() {
  echo "$(date '+%Y-%m-%d %H:%M:%S') Autoscaler shutting down cleanly"
  RUNNING=false
  exit 0
}
trap cleanup SIGTERM SIGINT

# --- Validate before starting: bad config = exit 1 ---
if [ -z "$SERVICES" ]; then
  echo "ERROR: No services configured" >&2
  exit 1
fi

if ! docker info > /dev/null 2>&1; then
  echo "ERROR: Cannot connect to Docker daemon" >&2
  exit 1
fi

notify() {
  local msg="[Autoscaler] $1"
  echo "$(date '+%Y-%m-%d %H:%M:%S') $msg"
  [ -n "$WEBHOOK_URL" ] && curl -sf -X POST "$WEBHOOK_URL" \
    -H "Content-Type: application/json" -d "{\"text\":\"$msg\"}" > /dev/null 2>
}

get_stats() {
  local svc=$1
  local containers=$(docker ps -q -f "name=${svc}")
  [ -z "$containers" ] && echo "0 0" && return
  docker stats --no-stream --format "{{.CPUPerc}} {{.MemPerc}}" $containers \
    | awk '{gsub(/%/,""); cpu+=$1; mem+=$2; n++} END {
        if(n>0) printf "%.1f %.1f", cpu/n, mem/n; else print "0 0"
      }'
}

check_and_scale() {
  local svc=$1
  local svc_down_file="/tmp/autoscale-down-${svc}"

  # Check if service exists and has running containers
  local containers=$(docker ps -q -f "name=${svc}")

  if [ -z "$containers" ]; then
    # Only notify on the transition to down
    if [ ! -f "$svc_down_file" ]; then
      notify "ALERT: service '${svc}' has 0 running containers"
      touch "$svc_down_file"
    fi
    return
  fi

  # Service is running — if it was down before, notify recovery
  if [ -f "$svc_down_file" ]; then
    notify "RECOVERED: service '${svc}' is back up"
    rm -f "$svc_down_file"
  fi

  local replicas=$(docker service ls -f "name=${svc}" --format "{{.Replicas}}"
    | head -1 | cut -d'/' -f1)
  [ -z "$replicas" ] && return

  # Cooldown check
  if [ -f "/tmp/autoscale-${svc}" ]; then
    local last=$(cat "/tmp/autoscale-${svc}")
    [ $(($(date +%s) - last)) -lt $COOLDOWN ] && return
  fi
  local stats=$(get_stats "$svc")
  local avg_cpu=$(echo "$stats" | awk '{print $1}')
  local avg_mem=$(echo "$stats" | awk '{print $2}')

  # Scale up — CPU or memory (either can cause customer impact)
  if (( $(echo "$avg_cpu > $CPU_SCALE_UP" | bc -l) )) || \
     (( $(echo "$avg_mem > $MEM_SCALE_UP" | bc -l) )); then
    if [ "$replicas" -lt "$MAX_REPLICAS" ]; then
      docker service scale "${svc}=$((replicas+1))" > /dev/null 2>&1
      date +%s > "/tmp/autoscale-${svc}"
      notify "SCALED UP $svc: $replicas → $((replicas+1)) (cpu:${avg_cpu}% mem:
    fi
  # Scale down — both must be low (no rush, this is housekeeping)
  elif (( $(echo "$avg_cpu < $CPU_SCALE_DOWN" | bc -l) )) && \
       (( $(echo "$avg_mem < $MEM_SCALE_DOWN" | bc -l) )); then
    if [ "$replicas" -gt "$MIN_REPLICAS" ]; then
      docker service scale "${svc}=$((replicas-1))" > /dev/null 2>&1
      date +%s > "/tmp/autoscale-${svc}"
      notify "SCALED DOWN $svc: $replicas → $((replicas-1)) (cpu:${avg_cpu}% me
    fi
  fi
}

echo "$(date '+%Y-%m-%d %H:%M:%S') Adaptive Autoscaler started"
echo "  Services: $SERVICES"
echo "  Normal: check every ${INTERVAL_NORMAL}s | Alert (>${ALERT_THRESHOLD}%):
echo "  Scale up: CPU>${CPU_SCALE_UP}% or MEM>${MEM_SCALE_UP}%"
echo "  Scale down: CPU<${CPU_SCALE_DOWN}% and MEM<${MEM_SCALE_DOWN}%"

while $RUNNING; do
  max_metric=0
  IFS=',' read -ra SVC_LIST <<< "$SERVICES"

  for svc in "${SVC_LIST[@]}"; do
    stats=$(get_stats "$svc")
    cpu=$(echo "$stats" | awk '{print $1}')
    mem=$(echo "$stats" | awk '{print $2}')
    check_and_scale "$svc"
    (( $(echo "$cpu > $max_metric" | bc -l) )) && max_metric=$cpu
    (( $(echo "$mem > $max_metric" | bc -l) )) && max_metric=$mem
  done

  # Write health file — healthcheck tests if this timestamp is recent
  date +%s > "$HEALTH_FILE"

  # Adaptive interval
  if (( $(echo "$max_metric >= $ALERT_THRESHOLD" | bc -l) )); then
```

```
    [ "$STATE" = "normal" ] && echo "$(date '+%Y-%m-%d %H:%M:%S') ALERT: ${max_
    STATE="alert"
    sleep $INTERVAL_ALERT
  else
    [ "$STATE" = "alert" ] && echo "$(date '+%Y-%m-%d %H:%M:%S') CLEAR: ${max_m
    STATE="normal"
    sleep $INTERVAL_NORMAL
  fi
done
```

And here's how you deploy it — as a proper Swarm service, in the same compose file as everything else:

```
services:
  autoscaler:
    image: docker:cli
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - ./autoscaler.sh:/autoscaler.sh:ro
    entrypoint: ["bash", "/autoscaler.sh"]
    environment:
      - SERVICES=api,dashboard,website
      - CPU_SCALE_UP=70
      - MEM_SCALE_UP=80
      - WEBHOOK_URL=https://hooks.slack.com/services/YOUR/WEBHOOK/URL
    healthcheck:
      test: ["CMD-SHELL", "[ -f /tmp/autoscaler-health ] && [ $$(( $$(date +%s)
      interval: 30s
      timeout: 5s
      retries: 3
      start_period: 10s
    deploy:
      replicas: 1
      resources:
        limits:
          memory: 32M
          cpus: '0.05'
      restart_policy:
        condition: on-failure
      placement:
        constraints:
          - node.role == manager
    networks:
      - app-network
```

That's everything K8s needs Prometheus, HPA, and AlertManager for — deployed as one service in your existing compose file. Proper exit codes: `exit 1` if it can't reach Docker or has no services configured, `exit 0` on clean shutdown via SIGTERM. Proper healthcheck: writes a timestamp every cycle, healthcheck fails if the timestamp goes stale (script hung or deadlocked), Swarm kills and restarts it. Service-down detection: if a watched service disappears or has zero running containers, it sends you a webhook alert and keeps monitoring everything else. Resource limits: 32MB memory, 5% CPU cap — it'll use a fraction of that. And configuration through environment variables, so you change thresholds without touching the script.

K8s needs three separate systems for this: Prometheus for metrics collection, HPA for scaling decisions, AlertManager for notifications. Three configs, three maintenance burdens, three things that can break. And even then, HPA polls on a fixed interval — equally lazy at 5% CPU as at 65% CPU. It can't adapt its own urgency. It's a cluster-wide setting that applies the same polling rate to every service regardless of load.

This script is smarter. It pays attention when it matters and sleeps when it doesn't. And it's deployed the same way as everything else — one service in your compose file, with the healthchecks and exit codes we told you to use. 32MB of memory. Practicing what we preach.

And what about disk? Neither Swarm nor K8s autoscale volumes natively. The difference is what happens when you need more space. In Swarm, volumes are directories on a Linux filesystem. Expand the disk in your VPS provider panel, run `resize2fs`, done. One native Linux command, no restart, no migration. K8s abstracted volumes behind PersistentVolumes, PersistentVolumeClaims, StorageClasses, and CSI drivers — and still can't autoscale them without third-party tools that require Prometheus. Scaling a volume *down* in K8s requires spinning up a temporary pod, attaching both volumes, and copying all data over. They took a one-line Linux command and turned it into a pipeline.

"BUT WHAT ABOUT...?" — DISMANTLING EVERY K8S ADVANTAGE

Let's go through every claimed K8s advantage and be honest about what's real and what's marketing.

"K8s has granular network policies." Docker gives you full control over networks. You create overlay networks, you decide which services attach to which networks, and you can map ports from and to. If a service isn't on the network, it can't reach it. Period. K8s adds the ability to say "pod A can only talk to pod B on port 443 using TCP" – essentially UFW but for K8s. Useful in theory, but I've never needed it in 10 years of production. Network-level isolation is enough for 99% of use cases.

"K8s has RBAC." It does – its own user auth system, built from scratch. Doesn't plug into Active Directory without middleware (you need OIDC plus something like Dex or Keycloak as a bridge). It exists because K8s clusters got so big that companies started sharing them across teams. If your team controls the server, you don't need another auth layer. For a 500-person org sharing one cluster? Useful. For everyone else? You have SSH and network access controls.

"K8s has Custom Resource Definitions and Operators." CRDs let you define new resource types and build controllers that automate complex operations – database failover, backup scheduling, certificate rotation. It's powerful, and Swarm doesn't have an equivalent API pattern. But let's be precise about what this actually is: it's many services watching and reacting to state changes. In Swarm, that's just containers running scripts. My MongoDB replica sets have been running flawlessly for a decade without an operator. A CRD is a service with a script wearing a fancier hat.

"K8s has superior persistent storage." K8s built StorageClasses, PersistentVolumes, PersistentVolumeClaims, and CSI drivers – an entire abstraction layer around what is fundamentally a Linux volume mount. That abstraction helps when 50 teams are dynamically provisioning storage from a cloud provider's API. On a VPS with a disk? You mount a volume. You can easily map S3, NFS, or anything else to a Docker container. It's Linux.

"K8s handles multi-cloud better." I'm literally running two continents right now on Swarm, isolated for GDPR compliance. Could connect them via VPN if I wanted a shared swarm, but I chose isolation by design. K8s "federation" solves the same problem with 10x the complexity. A VPN between two Swarm nodes is networking 101.

"K8s has a bigger ecosystem — Prometheus, Grafana, Istio, Helm." Every single one of those tools runs on Docker and Swarm. Prometheus scrapes metrics over HTTP — it doesn't care if those come from a K8s pod or a Swarm service. Grafana visualizes data from any source. These tools predate K8s or exist independently. The K8s ecosystem just packages them with Helm charts for easy deployment on K8s. But you can run all of them in Swarm containers.

"K8s has better self-healing." Swarm has healthchecks in compose files — interval, timeout, retries, start_period — and restart policies. If a container fails its healthcheck, Swarm kills it and starts a new one. K8s adds readiness probes that control whether traffic gets routed to a pod that's alive but not ready yet. That's a nice-to-have, not a dealbreaker.

"K8s scales better." This is the laziest claim of all. Swarm has `docker service scale`, replica counts in compose files, placement constraints, placement preferences, node labels, global vs replicated mode, rolling updates with configurable parallelism, and resource constraints. You can pin services to specific nodes, spread across availability zones, and scale any service in seconds. K8s can manage more nodes from a single control plane — thousands of nodes. But application scaling? Swarm does it the same way. The difference is autoscaling (covered above), not scaling itself.

"K8s has a management dashboard." So does Swarm — and it's free and runs on the same server. Portainer Community Edition is a single container that gives you a full GUI: service management, container logs, console access, resource monitoring, stack deployment from YAML, network and volume management. One command to install: `docker run -d -p 9443:9443 --name portainer --restart=always -v /var/run/docker.sock:/var/run/docker.sock portainer/portainer-ce:latest`. Dokploy is another free option — full deployment management with built-in Traefik reverse proxy, automatic SSL, GitHub/GitLab CI/CD integration, also one command to install. Both run as containers alongside your application. No extra machines. No extra cost. They sit on the same $83/year VPS that runs your entire production stack. Kubernetes dashboard options? The built-in K8s Dashboard needs RBAC configuration and a service account token just to log in. Lens limits its free tier — paid is $199/year. Rancher needs its own dedicated server. OpenShift Console requires enterprise licensing. The Swarm management tools run *on* your server. The Kubernetes management tools need their *own* server.

And if you don't want to use Portainer or Dokploy? Build your own. I built a custom management panel that monitors services, restarts containers, scales replicas, removes unused images/volumes/networks, manages deployments, and controls both my US and EU servers from a single interface — in 5 days. Every card and table has a pin function to keep key metrics visible, and an alert function where you set a threshold — "if CPU on this service exceeds X%, alert me." Custom per-service monitoring with user-defined thresholds, built in. The Docker management piece specifically took about a day. Because a Docker dashboard is just a UI calling the Docker API through the socket. Everything Docker can do from the CLI, your app can do through the socket. The entire service runs at 58 MB. Dokploy's management stack uses 539 MB to do less. Datadog charges per host per month to give you alerting. This is 58 MB and free.

When you line it all up, the only things K8s has that you can't easily replicate are RBAC for massive multi-tenant clusters and the CRD/Operator pattern for codifying automation into the API. Everything else either already exists in Swarm or is solvable with basic Linux, networking, and scripting.

THE AUTOSCALING INDUSTRY IS A BILLION-DOLLAR BAND-AID

Now let's talk about why autoscaling even exists, because this is where the story gets uncomfortable for the K8s ecosystem.

CAST AI's 2024 Kubernetes Cost Benchmark Report found that only 13% of provisioned CPUs and 20% of memory were actually utilized in K8s clusters. Thirteen percent. That means 87% of the CPU companies are paying for is sitting there doing nothing.

A January 2026 study analyzing 3,042 production clusters across 600+ companies found that 68% of pods waste 3-8x more memory than they actually use. One company was hemorrhaging $2.1 million annually on unused resources. The estimated annual waste per cluster is $50,000-$500,000.

And 99.94% of clusters analyzed were overprovisioned. Not most clusters. Not many clusters. Virtually all of them.

Why? Because the system is so complex that people overprovision out of fear. The study traced 73% of inflated memory configs back to three sources: the official Kubernetes docs examples (which use arbitrary values), popular StackOverflow answers from 2019-2021, and Helm charts with "safe" defaults. After experiencing a single OOMKilled incident, 64% of teams admitted to adding 2-4x headroom "just to be safe."

So the industry built an entire autoscaling ecosystem — HPA, VPA, Cluster Autoscaler, KEDA, plus dozens of FinOps platforms selling cost optimization tools — to manage the waste that the complexity created in the first place.

It's like inventing a more powerful engine instead of taking the bricks out of the trunk.

THE REAL SCALING PROBLEM ISN'T ORCHESTRATION — IT'S CODE

Here's the question nobody asks at K8s conference talks: why does the code need that many resources in the first place?

A typical Node.js container ships at 200-300MB. My containers average 40MB. Same language, same runtime, same kind of workload. The difference is 5-7x.

That gap doesn't come from orchestration magic. It comes from lazy defaults. A standard Node container pulls a full base image with hundreds of packages you'll never use, installs every npm dependency including dev dependencies, doesn't optimize the build, and ships with debugging tools, documentation files, and dead code baked in. 300MB before your first line of business logic even runs.

An efficient container uses Alpine, installs only production dependencies, multi-stage builds, and ships nothing that isn't needed at runtime. 40MB and it does the same job.

When your containers are 7x bigger than they need to be, of course you need autoscaling. You're burning through resources because every instance is carrying 200MB of dead weight. Then you need HPA to spin up more bloated containers, then you need Cluster Autoscaler to add more nodes to hold the bloated containers, then you need a FinOps team to figure out why your bill is $50k/month.

Or you could just write efficient code and run 24 containers on $83/year.

The entire autoscaling ecosystem is a billion-dollar band-aid for bad architecture.

ARCHITECTURE BEATS ORCHESTRATION EVERY TIME

My infrastructure follows one principle: each container has one job.

API – accepts incoming data and writes it. That's it. No computation on the request path. Data comes in, goes into the collection, response goes back. Milliseconds. 38 MB. 0.3% CPU.

Watcher – separate containers that watch collections and aggregate data natively. When new data lands, they aggregate it into ready-to-read documents. Runs at its own pace, completely decoupled from the API request cycle.

Dashboard – serves pre-aggregated documents. One read from MongoDB, render, done. It doesn't query and re-query. It doesn't compute anything on page load. The data is already shaped and waiting. 43 MB. 0.0% CPU.

Monitor – serves metrics. One job. 47 MB. 0.8% CPU.

Four containers. Four jobs. Zero overlap. If the watcher crashes, the API keeps ingesting. If the dashboard goes down, data keeps flowing and aggregating. Nothing is coupled. And because each container only does one thing, each one is tiny – no dead code, no unused dependencies, no bloat. That's how you get to 40MB average.

Compare that to what most teams build: one Express app with 200 npm packages that handles API routes, serves the frontend, runs background jobs, computes aggregations on every request, manages websockets, and has a scheduler tacked on. One container doing five jobs, carrying every dependency for all five, using 300MB. When traffic spikes, the background aggregation fights the API for CPU cycles. So they autoscale, and now they have three copies of that bloated container all fighting each other.

My ten containers averaging 38.5 MB each: 385 MB total for an entire SaaS platform. Their one container: 300 MB doing the same work worse. Then they need K8s to autoscale the 300MB monolith. Then they need FinOps to manage the bill. Then they wonder why their $200k/year infrastructure does what mine does for $83.

The irony is perfect: my architecture IS the microservices pattern that K8s advocates preach about. Small, single-responsibility containers that do one thing efficiently. I'm actually doing microservices correctly. I just don't need K8s to run them because when each container is doing one thing well, the orchestration overhead is trivial. Swarm handles it fine.

The K8s crowd evangelizes microservices architecture but then builds monolithic API containers that do everything, throws K8s autoscaling at the problem, and calls it cloud-native.

HOW 26 MB SERVES A LIVE API

People will look at those numbers and ask how. The answer isn't complicated — it's just discipline that most teams skip.

Every container runs the same lean stack: Node.js with the native MongoDB driver. No Mongoose. No ORMs. No abstraction layers adding memory overhead on every operation. The native driver sends a command and returns a result. That's it.

The aggregation framework pushes computation to the database — where it runs at the C++ level using indexes — instead of pulling raw data into Node and processing it in JavaScript. BulkWrite batches operations into single round trips. Proper indexing means queries take microseconds instead of full collection scans. Pipeline stage ordering matters — always `$limit` before `$lookup`, so you're joining 10 documents instead of 10,000.

NGINX and ModSecurity sit in front of Node as a reverse proxy — and this is where most teams get the architecture completely wrong. Rate limiting, gzip compression, SSL termination, WAF protection, IP jailing, request filtering, bot blocking — none of that touches Node. Ever. That's the proxy's job. It runs in compiled C code, handles hundreds of thousands of malicious requests daily, and Node never sees any of it.

Most teams do the opposite. They install `express-rate-limit`, `helmet`, `compression`, `cors` middleware — all running in JavaScript, on every single request, inside the application process. Every middleware layer adds memory, adds latency, and adds dependencies. Their Node process is doing security work, compression work, and SSL work on top of business logic. Then they wonder why their container needs 300 MB and 200m CPU.

My Node containers receive only clean, pre-filtered, already-decompressed traffic from NGINX. They do one thing: business logic. That's why they're 26-38 MB. The security, compression, and traffic management happen in a purpose-built C proxy that's been doing this job for decades, not in JavaScript middleware that reinvents the wheel on every request.

As a general rule: never publish Node directly to the internet. Always put a reverse proxy in front. NGINX, Caddy, Traefik — pick one. Node is single-threaded. When you expose it directly to the web, that one thread is handling SSL handshakes, gzip compression, rate limit calculations, bot detection, and your actual business logic — all on the same event loop. One slowloris attack holding connections open, one malformed payload that takes too long to parse, and the entire event loop blocks. Every user waits. NGINX handles thousands of concurrent connections across multiple worker processes in compiled C. It was built to sit on the internet and take abuse. Node was not. Let compiled code handle the internet. Let Node handle your app.

None of this is exotic. It's fundamentals. But most teams skip them, ship 300 MB containers full of middleware and abstraction layers they don't need, and then solve the resulting performance problems with more infrastructure instead of better code.

There's no orchestration tool in the world that compensates for getting these fundamentals wrong. And when you get them right, you don't need the orchestration tool.

THE COST COMPARISON THAT ENDS THE ARGUMENT

My setup:

- 2 VPS instances × $83/year = **$166/year**
- 24 containers across two continents
- Live SaaS platform receiving constant real-time data
- Zero container crashes, zero data loss, zero security breaches
- MongoDB replica sets running flawlessly
- Hundreds of thousands of WAF-blocked attacks daily
- Disaster recovery in 5-10 minutes
- Server CPU at 0.00%

A typical small Kubernetes production setup:

- EKS control plane alone: $72/month ($864/year) — just for the orchestration brain
- 3 worker nodes minimum for HA: $20-40/month each ($720-1,440/year)
- Total minimum infrastructure: **$1,584-2,304/year**
- Plus a DevOps engineer who understands K8s: $150k+/year
- Plus monitoring tools, FinOps platforms, cost optimization subscriptions
- Plus 87% of provisioned CPU sitting idle
- Plus $50,000-$500,000/year in estimated waste per cluster
- And they still get crashes

My entire annual infrastructure cost is less than what most teams spend on their Kubernetes monitoring tools in a single month.

THE ADOPTION NUMBERS TELL THE STORY

Kubernetes holds 92% of the container orchestration market. Docker Swarm sits at roughly 2.5-5%. On the surface, that looks like a settled debate.

But look deeper.

91% of Kubernetes users are organizations with 1,000+ employees. It's overwhelmingly a big-enterprise tool being adopted by teams that don't operate at enterprise scale.

Docker Compose/Swarm usage among PHP developers rose from 17% in 2024 to 24% in 2025 — growing — while Kubernetes fell by approximately 1%. Swarm is gaining ground among working developers who ship products instead of managing platforms.

A 2024 comparative analysis found Swarm achieving similar application response times with 40-60% lower resource consumption for clusters under 20 nodes.

Mirantis committed to long-term Swarm support through at least 2030, with significant adoption across manufacturing, financial services, energy, and defense — industries where operational simplicity and low overhead matter more than features you'll never use.

Swarm isn't dead. It's quietly thriving among teams that figured out they were paying the K8s complexity tax for capabilities they never needed.

THE ADOPTION CYCLE

There's a self-reinforcing cycle worth understanding. More people learn K8s, so more tutorials exist, so more companies adopt it, so hiring managers require it on resumes, so more people learn it. Cloud providers sell managed K8s. Vendors sell monitoring tools. Consultancies sell migration services. Training companies sell certifications. FinOps platforms sell cost optimization for the waste K8s creates.

None of that means K8s is bad — it means the ecosystem has momentum that goes beyond pure technical merit. Nobody has an economic incentive to tell you that Docker Swarm, which is free and built into Docker, might be enough for your use case.

That's not a conspiracy. It's just how markets work. The more complex solution creates more jobs, more tooling, more consulting hours. The simple solution that just works doesn't generate the same economic activity. Nobody can sell you a platform for "write efficient code and use the orchestrator that's already built into Docker."

But there's no vendor revenue in that answer.

THE BOTTOM LINE

Kubernetes is a powerful, well-engineered system that solves real problems for the teams that genuinely need it. If you need autoscaling on custom metrics for unpredictable 100x traffic spikes, multi-tenant RBAC for hundreds of developers sharing a cluster, CRDs for extending the orchestrator, or you're managing thousands of nodes — K8s is the right tool. Full stop.

But most teams don't need any of that. And the data proves it: 87% idle CPU, $50k-500k annual waste per cluster, 68% of pods overprovisioned by 3-8x, half of K8s users not even using the one feature that differentiates it from Swarm.

$166/year. 24 containers. Two continents. Live SaaS data pipeline. Zero crashes. Ten years. 0.3% CPU.

Those numbers aren't an argument that K8s is bad. They're evidence that for 99% of teams, the real problem was never orchestration. It was architecture. It was code efficiency. It was the discipline to make each container do one thing well at 40MB instead of doing five things badly at 300MB.

Fix those, and you won't need a $200k/year infrastructure to do what $166/year handles without breaking a sweat.

Tim Carter Clausen is a Danish-American full-stack architect and cryptographic researcher who writes at thedecipherist.com. His Docker Swarm Production Guide ranks #1 on Google, ahead of Docker's official documentation. He runs a live, dual-continent SaaS platform on $166/year with zero crashes.