Bare-metal platform redesign for the Jajiga production stack — physical & network layer, virtualization, Kubernetes cluster architecture, external stateful services, observability, and a phased zero-downtime migration plan.

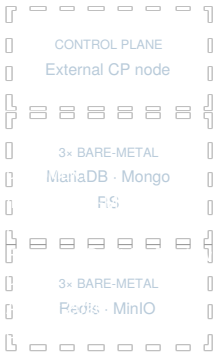


Table of Contents

00 — OVERVIEW

Executive Summary

This document proposes the redesign and migration of Jajiga's production platform from a three-node Docker Swarm cluster to a purpose-built, on-premise Kubernetes platform — covering everything from physical server provisioning and switching to cluster architecture, stateful-service placement, observability, and a phased zero-downtime cutover.

CURRENT PLATFORM

Docker Swarm

3 on-prem nodes, LAN-connected

TARGET PLATFORM

Kubernetes

Kubespray · Cilium · Traefik

EST. DURATION

≈27 weeks

~6.5 months, single DevOps engineer

Why migrate

The current Swarm stack — reviewed directly from the production `docker-compose.yml` — runs roughly 15 services with hardcoded database credentials inside the compose file, host ports exposed directly on the LAN interface, no ingress/TLS layer, and a mix of pinned and floating (`:latest`) image tags. These are workable trade-offs for a 3-node Swarm cluster, but they become liabilities as the platform grows: no native rolling-update safety net, no policy-based network segmentation, no secret encryption at rest in Git, and a single orchestrator with materially weaker self-healing and scheduling guarantees than Kubernetes.

What this proposal covers

- **Physical & network layer** — server roles, LAN/switching topology, out-of-band management.
- **Virtualization layer** — VMware ESXi/vCenter design carved out of two 72-core hosts.
- **Kubernetes cluster** — Kubespray-provisioned cluster, 3 control-plane nodes (1 off-host for quorum resilience), 4 worker nodes.
- **Cluster networking & add-ons** — Cilium (CNI), Traefik (ingress), cert-manager, Longhorn (storage), Sealed Secrets.
- **External stateful services** — MariaDB, MongoDB (new 3-node replica set), Redis, MinIO kept outside the cluster on 3 dedicated bare-metal/VM hosts.
- **Delivery pipeline** — a single generic Helm chart, packaged and versioned through a Nexus OCI registry, deployed from GitLab CI.
- **Observability** — Prometheus, Grafana, Loki, carried over and extended from the exporters already running today.
- **Migration & cutover plan** — parallel-cluster strategy, stateless-first rollout, stateful data migration, traffic cutover, and rollback plan.
- **Project roadmap** — a 13-phase execution plan mapped one-to-one to the Jira epic already opened for

this initiative (TECH-1792 → TECH-1804).

OPERATING CONSTRAINTS DRIVING EVERY DECISION BELOW

- Effectively **zero tolerance for unplanned downtime** on a production service with continuous real-user traffic.
- A **single DevOps engineer** owns build, run, and migration — every architectural choice below is biased toward operational simplicity over theoretical maximum scale.
- A **ten-person development team with no prior Kubernetes exposure** — the platform must stay approachable, well-labeled, and predictable.
- Two physical servers must survive the **loss of either one** without violating the availability constraint above.

01 — ASSESSMENT

Current State Assessment

The current staging/production stack was reviewed directly from its Docker Swarm compose definition. It is a monolithic `services:` block of roughly 15 containers sharing one Docker network, with data persistence handled by named local volumes and bind mounts on a single host's filesystem.

Service inventory

Fourteen services were verifiable directly from the reviewed `docker-compose.yml`. Four more — `frontend`, `kootam`, `search`, and `minio` — are known to exist in production from earlier discussion but were not present in the specific compose file reviewed for this document; they're listed separately below so the gap is explicit rather than guessed at.

api	Backend API	STATELESS	Depends on MariaDB & Redis; hardcoded `extra_hosts` IPs for the three Mongo nodes
dev	Backend API (develop branch)	STATELESS	Separate image/tag from `api`, same dependencies, own env file
kartlang	Search-adjacent service	STATELESS	Depends on PostgreSQL (`pgsql`); image tag driven by `KARTLANG_IMAGE_TAG`
cachedony	Internal cache service	STATELESS	Single port, env-file driven — used as the reference Helm chart template in Chapter 08
shorjiga	Short-link maker	STATELESS	Talks to MariaDB and MongoDB directly; Sentry DSN and DB credentials embedded in plain env vars
soketi	WebSocket server (Pusher-compatible)	STATELESS	Two ports exposed (9601 HTTP API, 6001 WebSocket)
blog (WordPress)	Marketing blog	STATELESS*	*Uses a **dedicated** MariaDB instance (not `jajiga_db`) and an external S3 bucket for media — carries no local disk state
adminer	Database admin UI	STATELESS	Dev/ops utility; low migration priority, candidate for dropping entirely in favor of kubectl port-forward
mariadb	Primary relational DB (`jajiga_db`)	STATEFUL	Root & app passwords hardcoded in compose; config and logs bind-mounted from host paths
pgsql	PostgreSQL 14 (kartlang)	STATEFUL	Password hardcoded; already has a `pg_isready` healthcheck defined
mongodb	Document store	STATEFUL	Image pinned to `mongo:latest` (floating tag); three nodes reachable via hardcoded IPs, but **replication status unconfirmed** — see Chapter 07

redis	Cache / queues	STATEFUL	No password configured; has a `redis-cli ping` healthcheck already defined
node-exporter, mysql-exporter, promtail	Observability agents	STATELESS	Already running today with custom scrape paths and basic-auth — a real head start on Chapter 09

KNOWN SERVICES NOT PRESENT IN THE REVIEWED COMPOSE FILE

frontend	Web frontend	Current image/registry path, build source, and whether it's stateless (expected) or holds any local cache/upload state
kootam	Admin panel	Current image/registry path and its database/API dependencies
search	Search service	Current image/registry path and what it indexes (MariaDB, MongoDB, or a dedicated search engine)
minio	Object storage	Confirmation of standalone vs. distributed mode today, and current bucket/data volume so Chapter 07's `mc mirror` migration can be sized correctly

Findings that directly shape this proposal

ISSUES TO RESOLVE DURING MIGRATION, NOT CARRY FORWARD

- **Plaintext credentials in compose files** — MariaDB root/app passwords, MongoDB root password, and PostgreSQL password are committed in clear text. Resolved by Sealed Secrets (Ch. 08).
- **Host ports published directly** (`3306:3306` , `5432:5432` , `57327:27017`) — replaced by ClusterIP Services and, for the external databases, LAN-only firewall rules (Ch. 04, 08).
- **Floating image tags** — `mongo:latest` and `postgres:14` style tags risk an unplanned version bump on redeploy. All images get pinned digests in the Helm values before go-live.
- **Hardcoded peer IPs** via `extra_hosts` for the Mongo nodes and MinIO backup host — replaced by proper DNS/Service discovery inside the cluster (Ch. 08).
- **No ingress / TLS termination layer** currently defined (the `nginx` block is commented out) — replaced by Traefik + cert-manager (Ch. 07).
- **Unconfirmed MongoDB replica-set status** — three nodes are reachable, but replication has not been verified. Treated as a from-scratch replica-set build (Ch. 08).

Traffic & availability profile

AVERAGE LOAD

~10 req/s

Moderate, steady production traffic

DOWNTIME BUDGET

≈ Zero

No acceptable planned outage window

CI/CD TODAY

GitLab CI

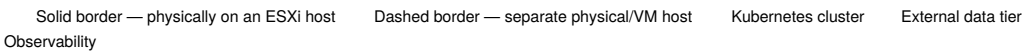
Carried forward, extended with Helm/OCI stages

02 — ARCHITECTURE

Target Architecture Overview

The target platform deliberately splits the world into two halves: a **stateless Kubernetes cluster** that only runs application workloads, and an **external stateful tier** of purpose-run database and object-storage hosts that the cluster talks to over the LAN. This is not a compromise forced by time constraints — for a single-operator team, it is the more reliable design: database failure domains stay outside the blast radius of cluster upgrades, CNI changes, or scheduler bugs, and existing backup/tuning tooling keeps working unmodified.

Figure 2.1 — Full request & data path: users enter through Traefik, land on worker VMs spread across two ESXi hosts, and reach the external data tier over the LAN; Observability scrapes both halves independently.



Layered view

Physical	2× 72c/128GB/1TB servers + 3× dedicated data hosts	Compute and storage hardware, LAN switching, out-of-band management
Virtualization	VMware ESXi + vCenter	Carves each physical host into control-plane and worker VMs with NUMA-aware sizing
Cluster bootstrap	Kubespray	Repeatable, from-scratch Kubernetes install with HA control plane
Networking (CNI)	Cilium (eBPF)	Pod networking, NetworkPolicy enforcement, Hubble observability
Ingress & TLS	Traefik + cert-manager	HTTP(S) routing into the cluster, automated Let's Encrypt certificates
Storage	Longhorn	Replicated block storage for in-cluster stateful add-ons (Prometheus, Grafana, Loki)
Secrets	Sealed Secrets	Git-safe, cluster-only-decryptable credentials
Packaging & delivery	Helm + Nexus (OCI)	One generic chart for all stateless services, versioned in the existing Nexus registry
Observability	Prometheus, Grafana, Loki/Promtail	Metrics, dashboards, alerting, and centralized logs — in-cluster and for the external data tier
External data tier	MariaDB, MongoDB RS, Redis, MinIO	All persistent state, isolated from cluster lifecycle events

DESIGN PRINCIPLE CARRIED THROUGH EVERY CHAPTER

Nothing in the cluster should require manual, tribal-knowledge recovery steps. Every stateful concern that would need that kind of care — database failover, replica-set elections, storage rebuilds — is deliberately pushed to systems designed for exactly that job, run outside Kubernetes, so that the one engineer operating this platform is never debugging two hard problems (orchestration and data durability) at the same time.

03 — LAYER 1

Physical & Network Layer

Hardware inventory

esxi-host-a	Kubernetes CP-1 + Worker-1 + Worker-2	72 cores	128 GB	1 TB
esxi-host-b	Kubernetes CP-2 + Worker-3 + Worker-4	72 cores	128 GB	1 TB
ext-data-01/02/03	MariaDB, MongoDB RS, Redis, MinIO (distributed across 3 hosts)	per current spec	per current spec	per current spec
cp-external	Kubernetes CP-3 (quorum node)	2–4 cores	4–8 GB	40–60 GB

LAN & switching topology

All hosts — the two ESXi servers, the external control-plane node, and the three data-tier hosts — sit on the **same flat LAN segment**, measured at ~1 Gbps between the ESXi hosts and the external control-plane node. That single fact removes the biggest usual risk for a 3-node etcd control plane (write latency), so no cross-site etcd tuning is required.

RECOMMENDED SWITCHING SETUP

- Use a **dedicated VLAN** for cluster/pod traffic, separate from management (ESXi/vCenter, iLO/iDRAC) traffic — even on a single physical switch, tag them into two VLANs.
- **Two physical NICs per ESXi host** in an active/active or active/standby team (LACP if the switch supports it), split across two switch ports at minimum — a single switch port failure should not take a whole host offline.
- Give the **external data-tier hosts a static IP plan** now (not DHCP) — every ExternalName Service and firewall rule in Chapter 07 depends on these addresses staying fixed.
- Reserve a small address block for **MetalLB-style or Traefik-facing VIPs** if a future requirement calls for a floating ingress IP; not required at this scale but cheap to reserve now.

Out-of-band management (iLO / iDRAC / IPMI)

Whatever the physical servers' vendor is (HPE iLO, Dell iDRAC, or generic IPMI), out-of-band management should be treated as production infrastructure in its own right:

- Put the **management interfaces on the isolated management VLAN** above — never expose iLO/iDRAC directly on the same broadcast domain as application or pod traffic.
- Change default credentials immediately and store them as a Sealed Secret / offline password-manager entry — not in any Git repository.
- Enable **virtual console + virtual media** so ESXi/OS reinstalls and BIOS-level troubleshooting never depend on physical site access.
- Configure **remote syslog / SNMP alerting** from iLO/iDRAC for hardware health events (disk predictive

failure, PSU, ECC memory errors) and route them into the same Loki/Alertmanager stack from Chapter 09 — hardware failures should surface in the same place as application alerts.

- Enable **scheduled firmware/BIOS update windows** as a recurring maintenance task, tracked alongside Kubernetes version upgrades.

ZERO-DOWNTIME IMPLICATION FOR THIS LAYER

Because the two ESXi hosts must each be able to lose the other and keep serving production traffic, **every NIC, switch path, and PDU/power feed should be reviewed for the same assumption**: nothing that is a single point of failure on Layer 1 should undermine the redundancy designed into Layer 3 and above. If both hosts currently share one switch or one power circuit, that is the highest-priority physical gap to close before go-live.

04 — VIRTUALIZATION

Virtualization Layer — ESXi & vCenter

Each 72-core / 128 GB physical host is carved into three VMs: one Kubernetes control-plane node and two worker nodes. vCenter manages both hosts as a single cluster, which unlocks DRS affinity rules — the actual mechanism that protects the two-server design from correlated failures.

VM layout per host

control-plane-N	4	8 GB	60 GB (thin, OS datastore)	Tainted — no application workload scheduled here
worker-N (×2 per host)	~34	~55 GB	80 GB OS + dedicated Longhorn disk	Sized to stay inside one NUMA node (see below)

Sizing & performance guidance

- **NUMA alignment** — with 72 physical cores typically split across 2 sockets (2 NUMA nodes), size each worker VM to fit inside a single socket's core count (e.g. ~34–36 vCPU on a 36-core-per-socket host) to avoid cross-NUMA memory access penalties. Confirm the exact per-socket core count in vCenter → Host → Hardware before finalizing VM sizes.
- **Network adapter type** — use VMXNET3 , not the legacy E1000 emulated adapter, for every VM's network interface to get near-native throughput.
- **DRS anti-affinity rules** — create an anti-affinity rule keeping the two control-plane VMs that live on the same physical estate apart from each other where possible, and a second rule keeping worker VM pairs spread so that a single host outage removes exactly one worker from each "pair," never two workers that back the same replica.
- **Dedicated Longhorn disk** — attach a separate VMDK (its own datastore or at least its own virtual disk, not sharing the OS disk) to each worker VM for Longhorn's block storage, so storage I/O contention never competes with the OS/container runtime disk.
- **Reserve, don't overcommit, control-plane resources** — set CPU/memory reservations on the three control-plane VMs so etcd write latency never degrades under host memory pressure from worker VMs.

CAPACITY MATH AGAINST THE ZERO-DOWNTIME CONSTRAINT

With 4 worker VMs total (2 per host) and a requirement to survive the loss of either physical host, **every Kubernetes resource request/limit decision in Chapter 05 and Chapter 08 must be validated against the capacity of only 2 worker VMs**, not 4 — this is already proven achievable: the same workload has been load-tested successfully on a single physical server of this spec, which is the strongest available evidence this design will hold under a real host loss.

05 — CLUSTER

Kubernetes Cluster Design

The cluster is bootstrapped from scratch with **Kubespray**, chosen over a bare `kubeadm` install for its built-in HA control-plane playbooks and CNI selection, and over a managed offering because the platform is fully on-premise/bare-metal.

Control plane — 3 nodes, 1 off-host

- **control-plane-1** (esxi-host-a), **control-plane-2** (esxi-host-b), **control-plane-3** (separate physical/VM host, same LAN, ~1 Gbps to the other two).
- This gives etcd real quorum resilience: losing either ESXi host still leaves 2 of 3 control-plane members alive, so the cluster keeps accepting writes.
- Control-plane nodes are tainted (`node-role.kubernetes.io/control-plane:NoSchedule`) so application pods never land there — with 8 GB reserved per CP VM this has ample headroom for etcd + API server + controller-manager + scheduler.

Worker pool — 4 nodes

- **worker-1, worker-2** on esxi-host-a; **worker-3, worker-4** on esxi-host-b.
- All application Deployments run here. Kubernetes' default scheduler, combined with pod anti-affinity rules on the Helm chart side, spreads replicas of the same service across both physical hosts automatically.
- **Sizing envelope:** total `requests` across all workloads must fit inside 2 worker VMs (the surviving half after a single host loss) — this is the binding constraint for every resource request/limit set in the Helm values.

Kubespray configuration highlights

Container runtime	containerd	Kubespray default, no Docker-shim overhead
kube-proxy mode	Disabled — replaced by Cilium	Cilium's eBPF datapath replaces kube-proxy entirely for lower latency
CNI	Cilium	See Chapter 06
etcd deployment	Stacked on control-plane nodes	Simpler ops for a 3-node CP; external etcd not justified at this scale
Kubernetes version	Latest stable at execution time, pinned explicitly	Avoids silent version drift between nodes
Inventory	Static Ansible inventory, kept in the same Git repo as the Helm chart	Cluster bootstrap becomes reproducible and reviewable

NODE ROLE SUMMARY

ROLE

control-plane ×3

ROLE

worker ×4

OUTSIDE CLUSTER

data tier ×3 hosts

Cluster Networking & Add-ons

Cilium — CNI

Cilium's eBPF datapath gives lower per-packet overhead than iptables-based CNIs, ships with native `NetworkPolicy` enforcement for segmenting namespaces (e.g. keeping `production-data` Secrets unreachable from pods that don't need them), and includes **Hubble** — a flow-level network observability UI that is disproportionately valuable for a solo operator troubleshooting connectivity without a networking team to lean on. Hubble UI is enabled from day one.

Traefik — Ingress & TLS

Traefik is deployed in the `ingress-system` namespace as the single entry point for all HTTP(S) traffic into the cluster, paired with **cert-manager** for automated Let's Encrypt certificate issuance and renewal. Compared to ingress-nginx, Traefik's CRD-based `IngressRoute` configuration and built-in dashboard are easier for a one-person team to reason about, and its native Kubernetes Gateway API support keeps the door open for a future migration path without another ingress-controller swap.

Longhorn — in-cluster block storage

With databases and MinIO living outside the cluster, Longhorn's remaining job is smaller but still necessary: persistent volumes for **Prometheus** (metrics history), **Grafana** (dashboards/config), and **Loki** (log chunks). Installed on a dedicated disk per worker VM (Chapter 04), with a replica count of 2–3 depending on final worker count, and snapshot/backup targets pointed at the external MinIO cluster.

Sealed Secrets

Every credential that would otherwise sit in plaintext in a Helm `values.yaml` — database connection strings, API keys, the Sentry DSN currently embedded in the Swarm compose file — is instead encrypted client-side with `kubeseal` against the cluster's public key before being committed to Git. Only the Sealed Secrets controller running inside the cluster can decrypt it. This was chosen over HashiCorp Vault deliberately: Vault's dynamic secrets, leasing, and multi-tenant ACL model solve problems this platform doesn't have yet, at the cost of another stateful service a one-person team would need to run, unseal, and back up.

Cilium	kube-system	CNI, NetworkPolicy, Hubble observability
Traefik	ingress-system	Ingress controller, TLS termination
cert-manager	cert-manager	Automated Let's Encrypt certificates
Longhorn	longhorn-system	Replicated block storage for in-cluster stateful add-ons
Sealed Secrets	sealed-secrets	Git-safe secret encryption

07 — DATA TIER

External Stateful Services Architecture

All persistent data services run outside Kubernetes, on 3 dedicated bare-metal/VM hosts on the same LAN. The cluster reaches them through Kubernetes `ExternalName` Services, so applications always resolve a stable in-cluster DNS name and never hardcode an IP.

EXAMPLE — EXTERNALNAME SERVICE PATTERN

```
apiVersion: v1
kind: Service
metadata:
  name: mariadb
  namespace: production-data
spec:
  type: ExternalName
  externalName: mariadb-vm1.internal.jajiga.com
```

MariaDB — two independent instances

MariaDB — primary (`jajiga_db`)	api, dev, shorjiga, kartlang*	*kartlang currently uses PostgreSQL; confirm during migration whether it stays on Postgres
MariaDB — WordPress (dedicated)	blog only	Fully isolated credentials and schema — WordPress never touches production data

Both instances are distributed across the 3 data-tier hosts (e.g. primary + one replica on two hosts, WordPress instance on the third) so that no single host loss removes both a MariaDB primary and its replica at once.

MongoDB — 3-node Replica Set (built from scratch)

The current 3 Mongo nodes are reachable but their replication status is unconfirmed, so this is treated as a clean build on **MongoDB 7.0.35** (the version already running today — pinned, not `:latest`):

1. Install MongoDB 7.0.35 independently on each of the 3 data-tier hosts.
2. Enable **keyfile authentication** for secure internal replica-set communication.
3. Run `rs.initiate()` to form the replica set (1 primary, 2 secondaries).
4. Migrate data: if the current instance is genuinely standalone, add the new nodes as secondaries for an online initial sync (lower downtime); otherwise use `mongodump`/`mongorestore`.
5. Update application connection strings to the multi-host replica-set form:
   `mongodb://user:pass@mongo1,mongo2,mongo3/db?replicaSet=rs0`

Redis

Deployed as a single primary today with no persistence configured; for the new platform, add **Redis Sentinel** across the 3 data-tier hosts for automatic failover, and set an authentication password (currently absent) via Sealed Secrets.

MinIO — distributed mode

Migrated from its current mode to **MinIO distributed mode** across the 3 data-tier hosts for erasure-coded redundancy. Data migration from the existing MinIO instance uses `mc mirror` — one of the lower-risk parts of this whole migration, since it can run incrementally ahead of cutover with a final delta sync at the end.

Placement matrix — surviving a single host loss

ext-data-01	Primary (jajiga_db)	Node 1	Sentinel + node	Erasure set member
ext-data-02	Replica (jajiga_db)	Node 2	Sentinel + node	Erasure set member
ext-data-03	WordPress instance	Node 3	Sentinel + node	Erasure set member

Network security for the data tier

Because Cilium NetworkPolicy has no effect on traffic to hosts outside the cluster, LAN-level firewall rules (iptables/nftables or VLAN isolation) must restrict inbound connections on the data-tier hosts to only the Kubernetes node IP range — closing the direct-internet-exposed ports seen in the current Swarm compose file.

08 — DELIVERY

Delivery Pipeline — Helm, Nexus & GitLab CI

One generic Helm chart, not fifteen

Rather than a dedicated chart per service, all stateless services are defined as entries in a single `values.yaml` map, rendered by one shared set of templates (`Deployment`, `Service`, optional `Ingress`). Adding a new service is a values-file change, not a new template — the right trade-off for a single maintainer supporting ten developers who don't yet read Kubernetes manifests themselves.

CHART STRUCTURE

```
jajiga-platform/
  Chart.yaml
  values.yaml          # one block per service
  templates/
    _helpers.tpl
    deployment.yaml     # ranges over .Values.services
    service.yaml
    ingress.yaml
  sealed-secrets/        # kubeseal output, safe to commit
```

ALREADY BUILT AS A WORKING STARTING POINT

The `cachedony` service has been implemented end-to-end in this chart structure as the reference template — every subsequent service follows the identical pattern.

Nexus as an OCI registry

Helm ≥3.8 speaks the OCI registry protocol natively, so the same Nexus instance already used for Docker images can host chart packages without a separate Helm-specific repository type:

```
helm package .
helm push jajiga-platform-0.1.0.tgz oci://nexus.jajiga.com:PORT/helm-charts
```

GitLab CI pipeline stages

build	Build & push the application image to Nexus/Harbor with an immutable tag (commit SHA), never `latest`
lint	`helm lint` and `helm template --dry-run` against the updated chart
package	`helm package` the chart and push to the Nexus OCI repo, versioned
deploy	`helm upgrade --install` against the target namespace, gated by manual approval for production

PRE-PRODUCTION CHECKLIST BAKED INTO THIS PIPELINE

- Image tags pinned to commit SHA — resolves the `:latest` issue found in Chapter 01.
- `envFromSecretName` in every service block references a Sealed Secret, never a plaintext value.
- Resource `requests`/`limits` reviewed against real Prometheus data before the service's first production deploy (Chapter 09 provides that data).

09 — OBSERVABILITY

Monitoring & Logging

The current Swarm stack already runs `node-exporter`, `mysql-exporter`, and `promtail` — a real head start. Observability is designed in from the start of the migration, not bolted on afterward, and it is the same stack that watches both the cluster and the external data tier.

Metrics — kube-prometheus-stack

- Deployed via the official `kube-prometheus-stack` Helm chart into `production-monitoring`: Prometheus, Grafana, and Alertmanager together, with a large library of ready-made dashboards.
- In-cluster services are scraped automatically via `ServiceMonitor` CRDs.
- The existing `node-exporter` and `mysql-exporter` continue running on the 3 external data-tier hosts, scraped through Prometheus' `additionalScrapeConfigs` — no rewrite needed, just a new scrape target.
- iLO/iDRAC hardware alerts (Chapter 03) and MongoDB/Redis-specific exporters are added to the same Prometheus instance so hardware, database, and application signals live in one place.

Logs — Loki + Promtail

- Loki deployed in-cluster, backed by a Longhorn-provisioned volume for chunk storage.
- The existing Promtail configuration is extended to also ship logs from the 3 external data-tier hosts, alongside in-cluster pod logs collected via a Promtail DaemonSet.
- Grafana becomes the single pane of glass for both metrics (Prometheus datasource) and logs (Loki datasource).

Why this is built before anything migrates

Monitoring and logging are Phases 07–08 on the roadmap in Chapter 11 — deliberately scheduled *before* Phase 09 (stateless service migration) starts. That ordering means every subsequent phase — stateless migration, validation, stateful migration, cutover — has real dashboards and alerting in place *from day one*, instead of retrofitting visibility onto an already-migrated platform where a regression would be much harder to catch.

10 — MIGRATION

Migration Strategy & Cutover Plan

Parallel-cluster strategy

The Kubernetes cluster is built and validated **alongside** the running Swarm cluster — never in place of it — until the very end. This is the single biggest lever for meeting the zero-downtime constraint: at every point before final cutover, the Swarm cluster remains the system of record and can absorb 100% of production traffic if anything in the new platform needs more time.

Phase order and why

STATELESS SERVICES FIRST

api, dev, frontend, kartlang, cachedony, shorjiga, soketi, kootam, search, blog — each has no local data to migrate, so each can be deployed to Kubernetes, smoke-tested against the (still-external, unchanged) databases, and validated independently, one at a time, with the Swarm version continuing to serve real traffic throughout.

STATEFUL SERVICES SECOND

MariaDB, MongoDB, Redis, and MinIO are migrated to their new dedicated hosts only after the application layer above them is already proven stable on Kubernetes — isolating the highest-risk data-migration work from application-layer risk, instead of changing both at once.

Cutover sequence

1. **DNS/load-balancer weighting** — shift a small percentage of read-only or low-risk traffic to the Kubernetes ingress first, watching Grafana dashboards built in Chapter 09.
2. **Full traffic switch** — once error rates and latency match or beat the Swarm baseline, move 100% of traffic to Traefik.
3. **Soak period** — keep the Swarm cluster warm (not decommissioned) for an observation window before touching it, per the roadmap in Chapter 11.
4. **Decommission** — only after the soak period passes cleanly, the legacy Swarm infrastructure is powered down and its hardware reclaimed (potentially into the data tier or as future cluster capacity).

Rollback plan

IF SOMETHING GOES WRONG AFTER CUTOVER

- **Before decommission:** DNS/load-balancer weight is simply reverted to the Swarm cluster — since it was kept warm and untouched, this is a fast, low-risk rollback.
- **Database changes:** stateful migration only proceeds one service at a time, with the previous host kept as a live read replica until the new host is proven — every stateful cutover step has its own independent rollback point, not just one big-bang switch.
- **Communication:** a rollback decision point and go/no-go criteria (error rate, latency, data-consistency checks) are agreed *before* each cutover step begins, not improvised during an incident.

11 — ROADMAP

Project Roadmap & Timeline

The 13 phases below map one-to-one to the Jira epic already opened for this initiative (TECH-1792 → TECH-1804), all owned by Milad Heidari. The original 8-month estimate has been re-planned to a tighter but still realistic **≈27 weeks (≈6.5 months)**, assuming this remains one engineer's primary but not sole responsibility. It includes explicit buffer around the highest-risk phases (stateful migration, cutover) rather than assuming best-case execution throughout.

ORIGINAL ESTIMATE

8 months

Provided estimate

PROPOSED ESTIMATE

≈27 weeks

≈6.5 months, with buffer

TARGET START → FINISH

Aug '26 → Feb '27

13 sequential/overlapping phases

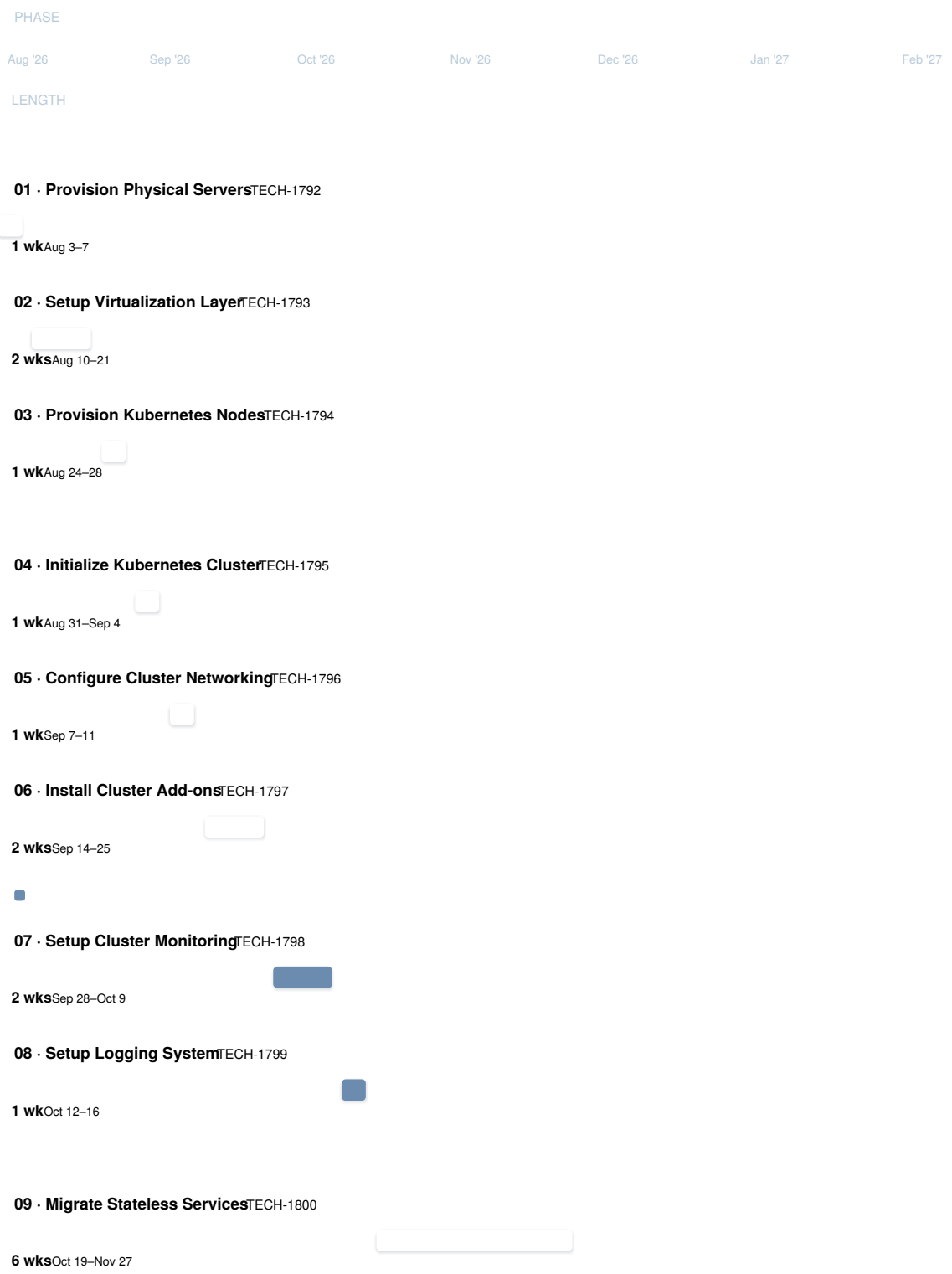
WHY NOT COMPRESS FURTHER

Stateless service migration (Phase 09) and stateful service migration (Phase 11) are intentionally the two longest phases — they involve, respectively, ten independent services each needing individual validation, and the highest-consequence, hardest-to-rollback work in the whole plan (live database replication, replica-set formation, distributed object storage). Compressing either would trade schedule for exactly the risk this plan exists to avoid.

11 — ROADMAP

Gantt Chart — 13-Phase Execution Plan

Phases are grouped into five stages that mirror the migration strategy from Chapter 10 — foundation work, cluster bootstrap, observability, migration & validation, and the highest-risk cutover work. Bar color encodes the group; exact dates and duration are printed for every phase so nothing depends on reading bar length alone.



10 · Validate Service Migration TECH-1801

2 wks Nov 30–Dec 11

11 · Migrate Stateful Services TECH-1802

6 wks Dec 14–Jan 22

12 · Switch Traffic to Kubernetes TECH-1803

1 wk Jan 25–29

13 · Remove Legacy Infrastructure TECH-1804

2 wks Feb 1–12

Foundation Cluster bootstrap Observability Migration & validation Cutover & closeout

Reference table

01	TECH-1792	Provision Physical Servers	Aug 3, 2026	Aug 7, 2026	1 wk
02	TECH-1793	Setup Virtualization Layer	Aug 10, 2026	Aug 21, 2026	2 wks
03	TECH-1794	Provision Kubernetes Nodes	Aug 24, 2026	Aug 28, 2026	1 wk
04	TECH-1795	Initialize Kubernetes Cluster	Aug 31, 2026	Sep 4, 2026	1 wk
05	TECH-1796	Configure Cluster Networking	Sep 7, 2026	Sep 11, 2026	1 wk
06	TECH-1797	Install Cluster Add-ons	Sep 14, 2026	Sep 25, 2026	2 wks
07	TECH-1798	Setup Cluster Monitoring	Sep 28, 2026	Oct 9, 2026	2 wks
08	TECH-1799	Setup Logging System	Oct 12, 2026	Oct 16, 2026	1 wk
09	TECH-1800	Migrate Stateless Services	Oct 19, 2026	Nov 27, 2026	6 wks
10	TECH-1801	Validate Service Migration	Nov 30, 2026	Dec 11, 2026	2 wks
11	TECH-1802	Migrate Stateful Services	Dec 14, 2026	Jan 22, 2027	6 wks
12	TECH-1803	Switch Traffic to Kubernetes	Jan 25, 2027	Jan 29, 2027	1 wk
13	TECH-1804	Remove Legacy Infrastructure	Feb 1, 2027	Feb 12, 2027	2 wks

12 — RISK

Risk Register

Loss of one physical host removes half of worker capacity	All (structural)	MEDIUM	Resource requests sized against 2-worker capacity (Ch. 04); DRS anti-affinity keeps replicas split across hosts
MongoDB replica set built from an unverified standalone source	11 · TECH-1802	HIGH	Treated as a from-scratch build with online secondary sync (Ch. 07); validated with test writes before any traffic depends on it
Data-loss window during stateful cutover (MariaDB, Mongo, Redis, MinIO)	11–12 · TECH-1802/1803	HIGH	Old host kept as live read replica until new host proven; per-service rollback points, not one big-bang switch (Ch. 10)
Single DevOps engineer is a bottleneck / single point of knowledge failure	All	MEDIUM	Everything captured as Helm values + Ansible inventory in Git, not tribal knowledge; this document itself is part of that mitigation
Developer team unfamiliar with Kubernetes slows incident response post-cutover	12–13 · TECH-1803/1804	MEDIUM	Grafana/Loki dashboards mirror familiar service names; basic runbook + short internal training session before cutover
Floating image tags reintroduce unpinned-version risk after migration	08 · Delivery	LOW	CI pipeline enforces commit-SHA image tags (Ch. 08)
Legacy Swarm decommissioned too early, removing the rollback safety net	13 · TECH-1804	MEDIUM	Mandatory soak period before decommission; explicit go/no-go criteria agreed in advance (Ch. 10)
Out-of-band management (iLO/iDRAC) left on a shared network segment	03 · Physical layer	LOW	Isolated management VLAN, credentials rotated and sealed (Ch. 03)

13 — OPERATIONS

Post-Migration Operations & Maintenance

Recurring operational tasks

Kubernetes minor version upgrades (via Kubespray)	Quarterly, off-peak window	DevOps
Helm chart / image dependency review	Monthly	DevOps
Sealed Secrets private key backup verification	Monthly	DevOps
MariaDB / MongoDB / Redis backup restore drills	Quarterly	DevOps
Longhorn volume & snapshot health check	Monthly	DevOps
iLO/iDRAC firmware & BIOS updates	Semi-annual, planned maintenance window	DevOps
cert-manager certificate renewal audit	Automated + monthly spot-check	DevOps
Resource request/limit tuning against Grafana trends	Monthly for first quarter, then quarterly	DevOps

Day-2 principles

- **Everything as code** — cluster bootstrap (Ansible/Kubespray inventory), workloads (Helm values), and secrets (sealed, encrypted) all live in Git; there should be no manual `kubectl edit` that isn't reflected back into a commit.
- **Alert before it's an incident** — Grafana/Alertmanager thresholds are set from the real baselines captured during Chapter 09's rollout, not guessed defaults.
- **Backups are only real if restored** — the quarterly restore drills above are non-negotiable; an untested backup is a hope, not a plan.
- **Capacity headroom review** — as more of the 10-person dev team's services land on the cluster, periodically re-validate the "survive one host loss" math from Chapter 04 against actual usage, not the original estimate.

HANDOVER ARTIFACT

This document, the Helm chart repository, and the Ansible/Kubespray inventory together form the operational handover package — sufficient for another engineer to pick up this platform without the original operator in the room.

APPENDIX A

Namespace & RBAC Reference

Only the production environment moves to Kubernetes at this stage — running separate dev/staging clusters is not justified for a one-person DevOps team, and namespace separation by **architectural layer** (rather than by dev team, which isn't yet Kubernetes-literate) keeps day-to-day operation simple while still giving real RBAC boundaries where they matter most.

production	api, dev, frontend, kartlang, cachedony, shorjiga, soketi, kootam, search, blog	Read-only: pods, logs, services
production-data	ExternalName Services, Sealed Secrets referencing the external data tier	No access by default
production-monitoring	Prometheus, Grafana, Loki, Alertmanager	Read-only: Grafana dashboards
ingress-system	Traefik	No access
cert-manager	cert-manager controller	No access
sealed-secrets	Sealed Secrets controller	No access
longhorn-system	Longhorn storage engine	No access

SAMPLE READ-ONLY DEVELOPER ROLE

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: production
  name: developer-readonly
rules:
- apiGroups: [""]
  resources: ["pods", "pods/log", "services"]
  verbs: ["get", "list", "watch"]
```

No equivalent Role is created for `production-data` — the absence of a binding is the access control. Database credentials never become visible to the development team through `kubectl`, regardless of what they can see in `production`.

APPENDIX B

Glossary

CNI	Container Network Interface — the plugin standard Kubernetes uses for pod networking. This proposal uses Cilium.
CRD	Custom Resource Definition — extends the Kubernetes API with new object types (e.g. Traefik's `IngressRoute`).
DRS	Distributed Resource Scheduler — vCenter's engine for placing and balancing VMs across ESXi hosts, including affinity/anti-affinity rules.
eBPF	Extended Berkeley Packet Filter — kernel technology Cilium uses for high-performance networking and observability without iptables.
etcd	The distributed key-value store backing the Kubernetes control plane's state; requires low-latency quorum between its nodes.
ExternalName Service	A Kubernetes Service type that resolves to an external DNS name instead of routing to pods — used here to reach the data tier.
Kubespray	An Ansible-based tool for installing production-ready Kubernetes clusters from scratch.
Longhorn	A distributed block-storage system for Kubernetes, used here only for in-cluster stateful add-ons.
NUMA	Non-Uniform Memory Access — a multi-socket CPU architecture where memory access speed depends on which socket's memory is being accessed.
Replica Set (MongoDB)	A group of MongoDB instances (one primary, N secondaries) that replicate the same data set for redundancy and failover.
Sealed Secrets	A Kubernetes controller and CLI tool (`kubeseal`) that lets Secrets be encrypted for safe storage in Git, decryptable only by the target cluster.
Sentinel (Redis)	Redis's built-in high-availability system for monitoring instances and performing automatic failover.
Taint / Toleration	A Kubernetes mechanism to repel pods from nodes (taint) unless they explicitly tolerate it — used to keep workloads off control-plane nodes.
vCenter	VMware's centralized management platform for ESXi hosts, clusters, and VMs.

End of document — INFRA-K8S-MIG-001, Rev. A Prepared for: Jajiga Engineering