NexGate Deployment Guide (New)

The paved road for shipping every NexGate service — architecture, flow, setup, and maintenance.

Owner: Kibuti · Organization: NexGate / nexgate-hq · Status: living document

---

How to read this

This is the single source of truth for how services are built, deployed, and operated across every NexGate service. It covers four things:

1. **Architecture** — what the pieces are and how they fit.
2. **Flow** — what happens when you push code, and how a change reaches production.
3. **Setup** — how to stand the whole thing up from zero.
4. **Maintenance** — how to run it day to day, add services, roll back, and scale.

If you only remember one sentence, remember this:

> **You write code, drop one folder, and register one secret role. An image is built once when you push, and pulled once per environment by the reconciler. Nothing else is touched.**

Table of contents

---

1. Core principles

The whole design turns on one shift away from the old Jenkins-driven pipeline:

Stop editing shared state imperatively. Declare desired state, and let a reconciler make reality match it.

Everything else follows from that:

- **Every service has the same shape** — a template so onboarding the 40th service is no harder than the 4th.
- **Each service is self-contained** — its own folder describes its containers, config, and secret permissions. One service's deploy can never touch another's.
- **Build once, promote by reference** — the image tested in staging is the exact artifact that runs in production. Prod never rebuilds.
- **Git is the control plane** — approvals are pull requests, history is the audit log, and rollback is a revert.
- **The pattern is substrate-agnostic** — the same repo and flow run on one VPS today and on many hosts later, without a rebuild.

---

2. Architecture

2.1 The three moving parts

Part	What it is	Who owns it
Service repos	Application code + a thin CI workflow (`nexgate-hq/<service>`)	One repo per service
`nexgate-infra`	Declarative desired state — what runs, where, at which version	One shared repo

Part	What it is	Who owns it
Reconciler (Komodo)	Watches `nexgate-infra` and applies changes on the host(s)	Installed once per host

There is **no Jenkins**. The reconciler is an off-the-shelf tool (Komodo, or Portainer as an alternative) — not code you write. It does exactly what Jenkins did on the deploy side (`docker compose pull` + `up -d`), driven by git instead of by a webhook.

2.2 Runtime layers

Each host runs two layers:

- **Base layer** — the always-on platform: Traefik, Vault Agent, PostgreSQL, Redis, RabbitMQ, MinIO. Rarely redeployed. Deploying an app service never touches it.
- **Service layer** — the application services (backend, notifications, File Thunder, `ai`, …), each deployed independently against the base layer.

2.3 Component roles

- **GHCR** (`ghcr.io/nexgate-hq/*`) — image registry. CI pushes here; the reconciler pulls from here.
- **Komodo** — the reconciler. Watches `nexgate-infra`, applies stack changes, provides a dashboard for logs/redeploy/rollback, and emits deploy notifications.
- **Vault** (`vault.qbitspark.com`) — secrets. Services authenticate via **AppRole**, scoped per service and per environment.
- **Traefik** — edge router. Auto-discovers services by container labels and serves `*.nexgate.co` over HTTPS (Let's Encrypt).

---

3. The repository layout

`nexgate-infra` is the heart of the system. Everything a service needs to run is described in its own folder — the same information that used to be crammed into two giant shared `docker-compose.yml` files, now cut apart so each service owns its slice.

```
nexgate-infra/
├── base/                          # the always-on platform layer
│   ├── traefik.yml
│   ├── vault-agent.yml
│   ├── postgres.yml               # optional shared Postgres for light services
│   ├── redis.yml
```

```
│   ├── rabbitmq.yml
│   └── minio.yml                  # always-on (File Thunder needs it live)
│
├── services/
│   ├── nexgate-backend/
│   │   ├── service.yml            # compose block: image, labels, depends_on
│   │   ├── config.env             # non-secret config + Vault paths
│   │   ├── vault-policy.hcl       # which Vault paths this service may read
│   │   └── tag.env                # IMAGE_TAG=...  ← the one line CI edits
│   ├── notification-server/
│   ├── file-thunder/
│   └── ai/
│
├── staging/
│   └── stack.yml                  # include: base + services (staging tags)
└── prod/
    └── stack.yml                  # include: base + services (prod tags)
```

Nothing in a service folder is a program. `service.yml` is the compose snippet you already write for a service. `tag.env` is one line. `vault-policy.hcl` is a few lines of permissions. The layout just guarantees that one service's deploy is isolated to its own files.

Example: a service folder

```
# services/ai/service.yml
services:
  ai:
    image: ghcr.io/nexgate-hq/ai-service:${AI_TAG:-latest}
    env_file: [config.env, tag.env]
    networks: [nexgate-prod, proxy]
    depends_on: [postgres, redis, rabbitmq]
    labels:
      - traefik.enable=true
      - traefik.http.routers.ai.rule=Host(`ai.nexgate.co`)
      - traefik.http.routers.ai.tls.certresolver=le
    restart: unless-stopped
```

```
# services/ai/tag.env      ← CI rewrites ONLY this file on each deploy
AI_TAG=0.1.0
```

Because `tag.env` is per-service, a deploy of `ai` cannot reset the backend's tag. The shared-`.env` clobber problem is structurally impossible.

---

4. The deploy flow

4.1 The one rule: built once, pulled per environment

- **Created** (built + pushed to GHCR): once, by CI, the moment you push code. Tied to the code, not the environment.
- **Pulled**: by the reconciler, once per environment, when that environment's tag changes to point at the image.

Staging pulls the image right after the change. Prod pulls the **same** image after approval. Prod never rebuilds.

4.2 End-to-end

```
git push (staging/master)
   │
   ▼
CI (GitHub Actions, reusable workflow)
   • build image via Buildx
   • push to GHCR (tagged version + latest)
   • bump tag in nexgate-infra (services/<svc>/tag.env)
   │
   ▼
nexgate-infra  ← declarative desired state
   │
   ▼  (prod only: approval gate — PR review / environment reviewer)
   │
   ▼
Komodo reconciler on the VPS
   • detects the tag change
   • docker compose pull <svc> && up -d <svc>   (only that service)
   │
   ▼
Service container up  →  Vault Agent injects secrets at boot
```

```
    |
    ▼
Traefik routes it over HTTPS (*.nexgate.co)
```

No SSH. No Jenkins job. Automated hops require zero server interaction.

4.3 Approvals

The gate lives in git, which is stronger than a button because it is reviewable and permanent.

- **Staging** — no gate. Merge to `staging`, it auto-syncs. Fast feedback.
- **Prod** — promotion is a **pull request** that bumps the prod tag (or merges `staging` → `master`). The PR is the approval: you see exactly which service and tag are moving. Merge = deploy.

Optional hard gates on top of the PR:

- A GitHub `production` environment with a **required reviewer** (may need a paid plan for private repos; if so, branch protection requiring a review before merge to `master` gives the same gate for free).
- Komodo's **manual-sync** for the prod stack — auto-sync staging, hold prod for a manual "deploy" click. This is the closest one-to-one replacement for the old Jenkins button.

Recommended: PR promotion + Komodo manual-sync for prod.

4.4 Notifications

Three taps, together richer than the old success/failure email:

- **CI** → a webhook step posts build + tag-bump status to Telegram (a bot is ideal — instant on mobile, free).
- **Komodo** → deploy success/failure to the same channel, reported *after* a health check rather than just "compose exited 0."
- **GitHub** → free email/mobile pings on workflow failures and pending PR approvals.

Post-deploy runtime health (the thing Jenkins never gave you) comes from Uptime Kuma or the Grafana stack — see §9.

---

5. Secrets model

5.1 Two tiers, per service

- `_infra` **tier** — raw container credentials (Postgres/Redis/RabbitMQ/MinIO passwords) needed to boot the base containers.
- **App tier** — the service's own secrets (JWT, DB creds, third-party API keys). The service reads these itself at boot.

5.2 AppRole, scoped per service and environment

Do **not** use one long-lived `VAULT_TOKEN` shared across environments (the old blast-radius problem). Instead:

- Each service has a Vault **AppRole** per environment (`ai_staging`, `ai_prod`, …) with a policy limited to its own paths.
- A **Vault Agent** authenticates with the AppRole, receives a short-TTL token, renders secrets to the service, and auto-renews.
- Staging credentials cannot reach prod paths. A leak in one environment is contained.

```
# services/ai/vault-policy.hcl
path "nexgate/data/ai_staging" { capabilities = ["read"] }
path "nexgate/data/ai_prod"    { capabilities = ["read"] }
```

---

6. Setup from scratch

Bootstrap order matters. Do these in sequence the first time.

Step 1 — Provision hosts

- Two Ubuntu VPS: `staging` and `prod` (separate boxes — see §10).
- Install Docker Engine + the Compose plugin.
- Lock down with UFW (allow 22, 80, 443; deny the rest).

Step 2 — Private networking (do it early)

- Set up a WireGuard mesh (or your provider's private network) between hosts, even if you start with one host each. This makes future host-splitting a one-line change instead of a migration.
- Services should address each other by name / internal DNS, never hardcoded `localhost`.

Step 3 — Docker networks

- One external proxy network for Traefik: `docker network create proxy`.
- One internal network per environment: `nexgate-staging`, `nexgate-prod`.

Step 4 — Deploy the base layer

On each host, bring up: Traefik (with the Let's Encrypt resolver `le`), Vault Agent, PostgreSQL, Redis, RabbitMQ, MinIO. MinIO is **always-on** from day one.

Step 5 — Configure Vault

- Enable the **AppRole** auth backend.
- For each service + environment, create a policy and a role.
- Write the service secrets to their app paths (`nexgate/<svc>_<env>`).

Step 6 — Create `nexgate-infra`

- Scaffold `base/`, `services/`, and `staging/stack.yml` + `prod/stack.yml`.
- Each stack `include`s the base plus the service folders for that environment.

Step 7 — Install and connect Komodo

- Install Komodo on each host (or a central Komodo managing per-host agents).
- Point it at `nexgate-infra`.
- Configure **staging = auto-sync**, **prod = manual-sync** (the gate).

Step 8 — Create `service-template` + reusable CI

- A `service-template` repo with the Dockerfile, the `workflow_call` caller, and `/health` + `/metrics` wired.
- A single reusable GitHub Actions workflow in the org that all services call.

Step 9 — Onboard the first service

- Run the add-a-new-service checklist end to end to validate the whole chain.

Step 10 — Wire notifications and observability

- Telegram webhook step in the reusable workflow.
- Komodo deploy alerts to the same channel.
- Stand up the **Grafana stack on a dedicated ops VPS** for runtime health — see §9 Monitoring.

---

7. Adding a new service

Assuming the base, Komodo, template, and reusable workflow already exist, onboarding a service (example: `ai`) is:

1. **Generate the repo** — `nexgate-hq/ai-service` from `service-template`. Dockerfile, CI caller, health/metrics already wired.
2. **Write the code** — your Spring Boot service. This is the only real work; everything below is config.
3. **First push** to `staging` → CI builds `ai:0.1.0` → GHCR. *(image created, first time)*
4. **Add one folder** — `services/ai/` with `service.yml`, `config.env`, `vault-policy.hcl`, `tag.env`.
5. **Register the Vault AppRole** for staging + prod, and write the app secrets.
6. **Add it to the staging stack** (or, if the stack globs `services/*`, Komodo picks it up automatically).
7. **Commit → Komodo syncs staging** → pulls `ai:0.1.0` → live at `ai.staging.nexgate.co`. *(pulled #1)* — validate.
8. **Promote to prod** — PR bumping the prod tag. Merge → Komodo syncs prod → pulls the same image → live at `ai.nexgate.co`. *(pulled #2, identical)*

The wiring (steps 4–5) is the entire cost — the old four-place scavenger hunt collapsed to **one folder + one AppRole**. Every change after this is just §8.

> **File Thunder note:** onboarding File Thunder additionally means uncommenting MinIO in `base/` (it's the service that finally needs it live) and declaring its own Postgres (on 5433) and a ClamAV container in its `service.yml`. Those are one-time, and they live entirely in File Thunder's folder + the base layer.

---

8. Deploying a change

For a service already onboarded:

1. Edit code, push to the `staging` branch.

2. CI builds `<svc>:<version>`, pushes to GHCR, bumps the staging tag. *(created)*
3. Komodo sees the staging tag change → pulls the image → restarts only that service in staging. *(pulled #1)* Automatic, no gate.
4. Validate in staging.
5. Open a PR bumping the prod tag (or merge `staging` → `master`). The PR is the gate.
6. Approve/merge → Komodo pulls the same image into prod. *(pulled #2, identical)*

Prod runs the byte-for-byte artifact you tested.

---

9. Maintenance and operations

Rollback

Revert the tag-bump commit in `nexgate-infra` (or use Komodo's "redeploy previous"). Because images are immutable and promoted by reference, rolling back is re-pointing the tag at the last-good image — fast and safe.

Rough edges (now fixed by design)

- **Shared `.env` clobber** → gone. Per-service `tag.env`.
- **One `VAULT_TOKEN` for both envs** → gone. AppRole per service/environment.
- **MinIO commented out** → gone. Always-on in `base/`.

Secret rotation

Rotate in Vault; the Vault Agent picks up the new value on renewal. No image rebuild, no redeploy needed for most secrets.

Backups

- **Postgres** — scheduled `pg_dump` per database (shared and per-service instances), shipped off-host.
- **MinIO** — bucket replication or scheduled sync to off-site storage.
- **Vault** — snapshot the storage backend regularly.
- **`nexgate-infra`** — it's git; it's already backed up and versioned.

Monitoring — the Grafana stack

Monitoring runs on a **dedicated ops VPS that sits outside dev, staging, and prod** — never on a box it watches, so it survives that box going down. **One ops host watches every environment**; targets are labelled by env (`env=dev|staging|prod`) and dashboards filter on that label. It joins the WireGuard mesh to reach services over internal IPs, and hits public endpoints through Traefik.

The stack:

Component	Question it answers
Prometheus	Metrics over time + alerting rules (scrapes each service's `/metrics`)
Loki	Log search (services log JSON to stdout)
Tempo	Distributed traces across services
Grafana	Dashboards and the single view across all of it
Alertmanager	Routes alerts → Telegram / email

Collectors that feed it:

- **node_exporter** — host CPU / RAM / disk, per VPS.
- **cAdvisor** — per-container resource use (spot a service starving the box before it takes others down).
- **blackbox_exporter** — uptime/health probes against `/health` and public URLs (this is the "is it up?" layer — no separate Uptime Kuma needed).
- **Grafana Alloy** (or Promtail) — ships stdout JSON logs into Loki.

Because every service already exposes `/metrics` and logs JSON to stdout (the service contract in §11), adding a new service to monitoring is just a scrape target + a health probe, both env-labelled.

Komodo vs Grafana — you need both

They answer different questions and neither covers the other:

	Komodo	Grafana stack
Watches	**Deployments** (control plane)	**Runtime health** (observability)
Tells you	Did it deploy, what tag runs where, deploy logs	Is it up, is it slow, error rate, resource use, log/metric history
Lets you	Redeploy, roll back	Alert, investigate — but *not* deploy

The gap is real: Komodo reports "deploy succeeded" the moment a container starts, even if it then crash-loops or serves 500s under load — Grafana is what catches that. Conversely, Grafana can tell you a service is unhealthy but can't redeploy or roll it back — that's Komodo. One tells you *something's wrong*; the other lets you *do something about it*.

Komodo does show basic per-host/container CPU/RAM/disk, so it doubles as light resource monitoring — but it has no metrics history, no log search, no dashboards, no tracing. The moment you need "why was it slow last Tuesday" or "alert me when p95 latency climbs," that's Grafana. Run both on the ops VPS; both alert to Telegram.

Upgrading base images

Postgres/Redis/etc. versions are pinned per service in `service.yml` (see §10.4). Bump the pin, test in staging, promote.

The single pane of glass

Komodo's GUI replaces the old Jenkins dashboard: see every service, view logs, redeploy, roll back. Choosing Komodo (or Portainer) is what fills that slot — it isn't optional flavor.

---

10. Scaling

10.1 Two kinds of scale

- **Many services** (control plane) — scales freely; that's the whole point of the paved road. The 40th service onboards exactly like the 4th.
- **Heavy load** (data plane) — single-host Compose has a ceiling. You climb a ladder, without rebuilding the pattern.

10.2 The ladder

Rung	Setup	When
1	Single VPS, Compose + Komodo	Now
2	Bigger VPS	Vertical scale buys runway
3	Services split across hosts (Komodo, multi-host)	When a service gets heavy
4	Swarm or K3s	Replicas + failover, much later

The repo and onboarding journey are **identical at every rung**. Moving up is a config change (which host a folder runs on), plus — at rung 3 — the private network from [§6 step 2].

10.3 VPS topology options

- **Split by environment (staging VPS + prod VPS)** — do this now. It's a safety baseline, not a scaling choice, and it enforces per-environment secret isolation.
- **Split by service** — a growth move. When File Thunder's transcoding starves the API, move it (and its workers) to its own host. One at a time, when load demands.
- **Every VPS runs everything** — careful. Fine for *stateless* services (horizontal replicas behind Traefik). A trap for *stateful* services: three MinIOs / Postgres / RabbitMQs are three diverging databases, not one system. This only works as true active-active HA with **clustered** datastores (Postgres replication, MinIO distributed mode, RabbitMQ quorum queues, Redis Sentinel) on Swarm/K3s — a distant milestone, never a switch you casually flip.

Sequence to commit to: **be on env-split today → grow into service-split as load appears → treat all-have-all as a distant HA project.**

10.4 Different versions per service

Postgres/Redis/MinIO are **upstream public images** you pin by tag — unrelated to the "build once" rule (which is only about your app image). Different services can pin different versions freely, because each service's datastore is a separate container in its own folder:

```
# services/file-thunder/service.yml
  ft-postgres:
    image: postgres:17
    volumes: [ft-pgdata:/var/lib/postgresql/data]
```

```
# services/backend/service.yml
  backend-postgres:
    image: postgres:15
    volumes: [backend-pgdata:/var/lib/postgresql/data]
```

They don't collide — different container names, volumes, and versions. **Shared vs dedicated is a per-service call:** a shared base Postgres saves RAM for light services (each gets a database inside it); a dedicated Postgres gives version/extension freedom and isolation (why File Thunder has its own). The Postgres version is a per-service decision in `service.yml`, never a base-layer constraint.

---

11. Conventions and invariants

Rules that keep the system coherent. Don't break these.

- **Everything runs in a container — no exceptions.** Every service (backend, frontend, AI, File Thunder) and every base component (Postgres, Redis, RabbitMQ, MinIO, Traefik) is

a container. Nothing is installed bare on a host. A static frontend still ships as a container (e.g. nginx serving built assets), not as a special case. This is what makes the whole model uniform: Komodo speaks Docker, the paved road builds images, and "add a service" always means "add a container."
- **One service = one repo = one folder** in `nexgate-infra`.
- **One deploy path for all service types.** Backend, frontend, and AI share the same tail (push GHCR → bump tag → Komodo deploys). Only the build *head* differs per language (Maven / npm / pip), as flavors of the same reusable workflow — never separate deploy systems. Isolation belongs at runtime (host targeting) and policy (approval gates), not in the pipeline.
- **CI writes only** `services/<svc>/tag.env` — never a shared file.
- **Prod is promoted by reference**, never rebuilt.
- **Secrets come from Vault via AppRole** — no long-lived tokens, no secrets in git.
- **Services address each other by name / internal DNS**, never hardcoded `localhost` (so host-splitting stays a one-line change).
- **Every service exposes** `/health` **+** `/metrics` **and logs JSON to stdout** (the service contract).
- **Staging auto-deploys; prod is gated by a PR** (plus optional manual-sync).
- **The base layer is stable** — deploying an app service never touches it.

---

12. Glossary

- **Reconciler** — the tool (Komodo) that watches `nexgate-infra` and applies changes on the hosts. Replaces Jenkins' deploy role. Not code you write.
- **Desired state** — what `nexgate-infra` declares should be running (services, versions, hosts).
- **Base layer** — the always-on platform containers (Traefik, Vault Agent, Postgres, Redis, RabbitMQ, MinIO).
- **Service layer** — the application services deployed against the base layer.
- **AppRole** — Vault auth method giving each service a scoped, short-lived credential per environment.
- **Promote by reference** — moving the exact same image artifact from staging to prod, rather than rebuilding.
- **Paved road** — the standardized template + workflow + folder pattern that makes onboarding a new service near-free.

This document is versioned in `nexgate-infra`*. Update it whenever the architecture changes — it is the guide the whole organization relies on.*

Revision #1
Created 6 July 2026 12:24:20 by Admin Qbit
Updated 6 July 2026 12:24:51 by Admin Qbit