Kneo Agent Platform

Operation Guide

Version 0.4.0

2026-05-22

Deployment shapes, TLS termination, security hardening, observability, backup and recovery, upgrade, incident response, troubleshooting, and the deployment smoke test — one printable operator guide.

Table of contents

Deployment

Source: *docs/user/deployment.md*

Reference for the supported deployment shapes. For a guided zero-to-running walkthrough on Docker Compose with PostgreSQL, see `tutorial_postgres_deployment.md`. For every environment variable referenced below, see `environment.md`.

The service supports three shapes:

- **Container** — a single `kneo-serv` image with a database you supply.
- **Compose** — the bundled stack that starts the API plus PostgreSQL.
- **Embedded** — `kneo_serv.service.app:create_app()` mounted in your own ASGI server (covered in `tutorial_custom_tool.md § 7`).

Container

Pull the published image from GitHub Container Registry:

```
docker pull ghcr.io/kneo-agent/kneo-serv:latest
```

Tag conventions: `<version>` (e.g. `0.4.0`), `<major>.<minor>` (`0.4`), and `latest`. amd64-only for the 0.4.x line; arm64 is deferred to 1.0. From 0.3.0 onward the image is keyless-signed via cosign and ships with a CycloneDX SBOM attestation; from 0.4.0 onward the release pipeline also runs a blocking Trivy CVE scan against the pushed digest under the CVSS≥7 policy (security_hardening.md § Image vulnerability scanning) — verification commands in `supply_chain_review.md § Verification commands`. The image installs the `kneo-serv[deploy]` extra (psycopg + SDK telemetry).

Run it against PostgreSQL:

```
docker run --rm -p 8000:8000 \
  -e KNEO_SERV_DATABASE_URL=postgresql://kneo_serv:change-
me@host.docker.internal:5432/kneo_serv \
  -e KNEO_SERV_AUTH_ENABLED=true \
  -e KNEO_SERV_API_KEYS='operator:replace-token:operator' \
  ghcr.io/kneo-agent/kneo-serv:latest
```

For local builds from a source checkout (contributor / pre-publish):

```
docker build -t kneo-serv:local .
```

Compose

The bundled Compose stack starts the API plus PostgreSQL:

```
cp deploy/production.env.example deploy/production.env
docker compose --env-file deploy/production.env up --build
```

Replace every placeholder token and database password before binding the service to a network. The stack defaults to port `8000`; set `KNEO_SERV_PORT` to change the host-side port.

For a staging rehearsal, use the staging env example:

```
cp deploy/staging.env.example deploy/staging.env
KNEO_SERV_ENV_FILE=./deploy/staging.env \
  docker compose --env-file deploy/staging.env up --build
```

`deploy/staging.env` is gitignored. Keep SDK telemetry argument/result capture (`KNEO_SERV_OTEL_RECORD_ARGUMENTS`, `KNEO_SERV_OTEL_RECORD_RESULTS`) disabled in staging unless the deployment's data classification has explicitly approved payload capture.

TLS and reverse proxy

The service speaks bare HTTP and does not terminate TLS itself. For any deployment exposed beyond `127.0.0.1`, place a reverse proxy (nginx, Caddy, AWS ALB, or similar) in front and terminate TLS there. See `tls_and_proxy.md` for topology, bind-address guidance, and trusted-proxy header handling.

Choosing a persistence backend

Backend	When to use
SQLite	Local dev or single-process service. Default when `KNEO_SERV_DATABASE_URL` is unset.
PostgreSQL	Any multi-process or production deployment. Set `KNEO_SERV_DATABASE_URL`. Requires `kneo-serv[postgres]` or `kneo-serv[deploy]`.

When `KNEO_SERV_DATABASE_URL` is set, the service uses PostgreSQL for run state, checkpoints, idempotency records, queue leases, locks, audit events, and workflow continuations. Without it, the service falls back to SQLite for state and file-backed continuations.

Multi-process SQLite is not a supported topology; see troubleshooting.md § 2.3.

Readiness and liveness

Wire these endpoints into your supervisor or load balancer:

```
GET /livez      # process liveness
GET /readyz     # readiness: all dependencies healthy
GET /healthz    # lightweight overall health
```

`/livez` and `/readyz` are intentionally unauthenticated for probe integration. `/readyz` returns `503` with a structured `not_ready` payload when any dependency check fails — see troubleshooting.md § 1.2 for the failure shape.

TLS and reverse proxy

Source: *docs/user/tls_and_proxy.md*

The Kneo Agent Platform service speaks plain HTTP. It does not terminate TLS, parse `X-Forwarded-*` headers itself, or rate-limit by IP. Any deployment that faces a network beyond `127.0.0.1` must run behind a reverse proxy that terminates TLS and shields the service.

For deployment shapes (Container, Compose, Embedded) and the choice of persistence backend, see `deployment.md`. For the hardening checklist that includes TLS, see `security_hardening.md`.

Topology

```
client ──HTTPS──► reverse proxy ──HTTP──► kneo-serv
                  (TLS termination,
                   request size limits,
                   rate limiting,
                   client-IP injection)
```

The proxy is responsible for:

- TLS termination and certificate management
- Request body size limits at the edge (defense in depth above `KNEO_SERV_MAX_BODY_BYTES`)
- IP-based rate limiting if you need it (the service has no built-in per-IP limiter)
- Forwarding the client IP for the service's structured logs

Run the proxy and `kneo-serv` on the same host (or in the same private network) so the unencrypted hop is not exposed.

Bind address

Topology	`--host` value	Rationale
Proxy + service on the same host	`127.0.0.1`	Service is unreachable except through the proxy.
Proxy + service in a shared private network	`0.0.0.0`	The network boundary is the proxy; firewall the service port.
Compose (`compose.yaml`)	`0.0.0.0` inside the container; only the proxy's port is published on the host.	The Compose stack's internal network already isolates the API service.

The Dockerfile defaults to `--host 0.0.0.0 --port 8000`. Override with `KNEO_SERV_PORT` for the published host-side port; the container port is fixed at `8000`.

Trusted-proxy headers

The service logs the immediate TCP peer as `client_ip` in its structured request logs (`observability.md`). When a proxy fronts the service, the immediate peer is the proxy, not the original client. To capture the real client IP in logs and traces, configure the proxy to write `X-Forwarded-For` upstream and ingest it at your log aggregator — the service itself does not rewrite `client_ip` from `X-Forwarded-For` (no implicit trust).

The service does honor `X-Request-ID` and echoes it back on the response. Proxies that already inject a request ID should pass it through; the service generates a UUID per request otherwise.

Reverse-proxy snippets

These are minimal examples. Production configurations should add timeouts, buffer sizing, and rate-limit zones; consult your proxy's docs.

nginx

```
server {
  listen 443 ssl http2;
  server_name kneo.example.com;

  ssl_certificate     /etc/letsencrypt/live/kneo.example.com/fullchain.pem;
  ssl_certificate_key /etc/letsencrypt/live/kneo.example.com/privkey.pem;

  client_max_body_size 2m;   # match or exceed KNEO_SERV_MAX_BODY_BYTES

  location / {
    proxy_pass         http://127.0.0.1:8000;
    proxy_http_version 1.1;
    proxy_set_header   Host              $host;
    proxy_set_header   X-Forwarded-For   $proxy_add_x_forwarded_for;
    proxy_set_header   X-Forwarded-Proto $scheme;
    proxy_set_header   X-Request-ID      $request_id;
    proxy_read_timeout 130s;             # exceed KNEO_SERV_CLIENT_TIMEOUT
  }
}
```

Caddy

```
kneo.example.com {
  reverse_proxy 127.0.0.1:8000 {
    header_up X-Forwarded-For {remote_host}
    header_up X-Request-ID    {http.request.uuid}
  }
  request_body {
    max_size 2MB
  }
}
```

AWS ALB / generic L7 load balancer

- Listener: HTTPS 443 with an ACM certificate; redirect 80 → 443.
- Target group: HTTP, port 8000, healthcheck `GET /readyz` (interval 30s, unhealthy threshold 3, success codes `200`). `/readyz` is unauthenticated by design.
- Idle timeout: ≥ `KNEO_SERV_CLIENT_TIMEOUT` (default 120s); set 130s for a safety margin.
- Body size: ALBs cap at 1 MiB by default — if you accept larger inline specs (`KNEO_SERV_MAX_INLINE_SPEC_BYTES` is 256 KiB by default), use a CloudFront or nginx tier in front and bypass the ALB cap accordingly.

Health-check endpoints behind the proxy

Expose `/livez` and `/readyz` directly to the proxy or load balancer. Both are unauthenticated to keep probe integration simple. Do **not** expose `/readyz` to the public internet — its `503 not_ready` payload includes internal check names and registry contents that should stay inside the operational perimeter.

For most setups: bind the proxy's probe routes to internal listeners only, or restrict the source IP range to your load-balancer subnet.

Verifying TLS is actually in front

```
# TLS terminates at the proxy, service is unreachable directly.
curl -sf https://kneo.example.com/readyz | jq '.metadata.ready'   # → true
curl -sf http://kneo.example.com/readyz                            # → connection refused / 301
curl -sf http://127.0.0.1:8000/readyz                              # → only succeeds from the
proxy host
```

If the third command succeeds from outside the proxy host, the service port is reachable from the public network and the bind address or firewall is misconfigured.

What `kneo-serv` does not provide

- **No built-in TLS.** Terminate at the proxy.
- **No `X-Forwarded-For` rewriting.** Capture client IPs at the proxy or in your log aggregator.
- **No per-IP rate limiting.** Use the proxy's rate-limit zone.
- **No mTLS to upstream providers.** Provider connections go out from the service host; lock down egress at the network layer.

See `security_hardening.md` for the full pre-launch checklist.

Security hardening

Source: *docs/user/security_hardening.md*

Pre-launch checklist for taking a `kneo-serv` deployment to production. Each item references the authoritative configuration doc; this page is the single sheet to walk before going live.

For the auth model itself (roles, scopes, route mapping), see `service_api.md § Authentication`. For audit-event details, see `service_api.md § Audit events`.

Pre-launch checklist

1. Enable authentication

- [ ] `KNEO_SERV_AUTH_ENABLED=true` is set (or `KNEO_SERV_API_KEYS` / `KNEO_SERV_ADMIN_API_KEY` are set, which enables auth implicitly).
- [ ] No API keys are committed to the repo, the Compose `.env` file, or example configs.
- [ ] Each consumer has its own key, named so audit events identify the caller (`KNEO_SERV_API_KEYS='ci:…:service;analyst:…:viewer'`).
- [ ] The admin key (`KNEO_SERV_ADMIN_API_KEY`) is issued separately and used only for break-glass operations.

2. Assign the narrowest role

Pick the narrowest built-in role that covers each caller's needs. The canonical role-to-scope mapping lives in `service_api.md § Authentication`; below is the operational guidance for choosing between them.

Role	Use for
`admin`	Break-glass operator key only
`operator`	Day-to-day operator console / CI deploy
`service`	Server-to-server callers that drive runs
`reviewer`	Human-in-the-loop approvers
`viewer`	Dashboards, read-only analytics

Custom scopes are allowed in the third field of `KNEO_SERV_API_KEYS` when no built-in role fits.

- [ ] No consumer is using `admin` for routine traffic.

- [ ] Read-only consumers are on `viewer`, not `operator`.

3. Rotate keys without downtime

`kneo-serv` has no in-place key rotation API in the 0.4.x line. Rotation is a config swap:

1. Add the new key to `KNEO_SERV_API_KEYS` alongside the old key (semicolon-separated entries; same `name:` is fine).
2. Restart the service. Both keys are now valid.
3. Roll callers over to the new key.
4. Remove the old entry from `KNEO_SERV_API_KEYS`.
5. Restart again.

- [ ] Rotation procedure is rehearsed in staging before production keys are issued.
- [ ] Old keys are revoked, not left "in case."

4. Sign spec bundles for production

For environments that block ad-hoc spec edits:

- [ ] `KNEO_SERV_SPEC_SIGNING_KEY` is set in CI and on the service hosts (different value than any API key).
- [ ] Production deploys use signed bundles only: `kneo spec bundle sign … --approved-by <name> --env prod` and `kneo spec bundle verify <bundle>` in the deploy pipeline.
- [ ] Project config declares `environments.prod.policy_enforcement` so CLI and API spec flows enforce policy after overlays (`project_config.md`).

5. Terminate TLS upstream

- [ ] A reverse proxy in front of the service terminates TLS; the service bind address is `127.0.0.1` or restricted to a private network. See `tls_and_proxy.md`.
- [ ] `/readyz` is exposed only to the load balancer or probe subnet (see `tls_and_proxy.md § Health-check endpoints behind the proxy`).
- [ ] The proxy enforces a request body size limit ≥ `KNEO_SERV_MAX_BODY_BYTES`.

6. Lock down container and host

The bundled Dockerfile already enforces a non-root user (`kneo`); you do not need to override it. (`Dockerfile`)

- [ ] Container runs as the non-root `kneo` user (default).
- [ ] Image is pulled from a trusted registry; tags are pinned by digest in production manifests.
- [ ] CI scans the image for known CVEs; releases blocked on HIGH/CRITICAL findings (§ Image vulnerability scanning).

- [ ] Host or Kubernetes drops Linux capabilities the service doesn't need (no `NET_ADMIN`, no `SYS_ADMIN`).
- [ ] Egress is restricted at the network layer to the provider, MCP, and observability endpoints the deployment actually uses.

The container's filesystem is **not** read-only — the service writes checkpoints, queue state, and optionally SQLite files. If you need read-only-root, mount writable volumes for `.kneo/` (SQLite + continuations) and the artifact paths declared in your spec.

7. Protect the audit trail

- [ ] Audit events are persisted in the same backend as run state (PostgreSQL in production); the DB is backed up per `backup_and_recovery.md`.
- [ ] `audit:read` is scoped to a small set of principals (compliance, on-call, incident-response).
- [ ] Audit-event retention is set deliberately. If `KNEO_SERV_RETENTION_RUNS_DAYS` is set, runs and their audit events age out together — confirm that aligns with your compliance window before enabling it (`environment.md § Retention`).

8. Keep redaction in place

`kneo-serv` redacts secrets, tokens, authorization headers, emails, and SSNs from responses, traces, checkpoints, and CLI JSON output by default (`service_api.md § Redaction`). The two escape hatches both default to off:

- [ ] `KNEO_SERV_OTEL_RECORD_ARGUMENTS=false` (unless tool arguments are classified safe to emit to your trace backend).
- [ ] `KNEO_SERV_OTEL_RECORD_RESULTS=false` (same rationale).
- [ ] Custom tools and middleware do not log user inputs or provider responses without redaction.

Image vulnerability scanning

The release pipeline scans every published GHCR image for known CVEs using Trivy. The scan runs against the **pushed image digest** (the same bytes cosign signed and SBOM attestation describes), so the four supply-chain artifacts — image, cosign signature, SBOM attestation, and scan report — all agree on what they describe.

Locked policy (0.4.0; recorded in plan/TODO-0.4.0.md):

- **Severity threshold:** CVSS≥7 (HIGH and CRITICAL findings).
- **Release-tag scans (`v<version>` / `v<version>rcN`):** blocking. The Trivy step in `release.yml` runs with `--exit-code 1`; HIGH/CRITICAL findings fail the step, preventing

the `Publish build artifact` and `Publish GitHub release` steps from running. The publish is the irreversible step, so the gate fires there.
- **PR-time scans:** report-only via `.github/workflows/image-scan.yml`. The same scanner version + severity threshold runs against a locally-built image but with `--exit-code 0`, so findings surface in the PR's check summary without blocking merges. Dev velocity isn't gated on un-fixable transient base-image CVEs; the release gate catches anything that matters before publish.
- **Scan report retention:** the JSON report is attached as a GitHub Actions artifact (`trivy-report-<version>`) on every release-tag build, retained for 90 days. The deployer can download it for audit.

Operator-side verification

Re-run the scan locally against any published tag:

```
trivy image \
  --severity HIGH,CRITICAL \
  --ignore-unfixed=false \
  ghcr.io/kneo-agent/kneo-serv:<tag>
```

Cross-check against the release-time scan output by downloading the `trivy-report-<version>` artifact from the GitHub Release.

Accepted findings

If an upstream CVE has no fix available, or the deployer's risk tolerance accepts a specific finding (e.g. low exploitability in your network posture), record the acceptance in `supply_chain_review.md § Current workspace result` using the same shape as the existing `pip-audit` remediation blocks. The release pipeline does not implement an inline-ignore mechanism — acceptances are deployer policy, not platform policy.

What `kneo-serv` deliberately does not provide

Operators sometimes go looking for these; document the gap rather than inventing it:

- **No built-in TLS.** Terminate at a reverse proxy (`tls_and_proxy.md`).
- **No per-IP rate limiting.** Use the reverse proxy's rate-limit zone.
- **No mTLS to upstream providers.** Provider HTTPS calls leave the service host; control with egress firewall rules.
- **No live key rotation API.** Keys are configured via env vars; rotate with a config swap and restart (§ 3).

- **No external secret-manager integration.** Secrets are injected via environment variables. Use your platform's secret store (Kubernetes Secrets, AWS Secrets Manager, Vault) to populate the env at startup.
- **No SCIM or directory integration.** API keys are flat-file in `KNEO_SERV_API_KEYS`; map them to identities in your audit log aggregator.

These are tracked in the roadmap, not bugs. See `docs/plan/roadmap.md`.

Observability

Source: `docs/user/observability.md`

Operator guide for wiring `kneo-serv`'s structured logs, request tracing, and OpenTelemetry exports into a production observability stack.

This page is the setup view. For symptoms and recovery when observability itself misbehaves, see `troubleshooting.md § 7`. For the full env-var list, see `environment.md § Observability`.

Three signals, three surfaces

Signal	What it is	Where it comes from
Structured request logs	One JSON record per HTTP request, redacted	`RequestLoggingMiddleware` (always on by default)
Service-side trace events	Per-run trace and checkpoint records, queryable via the API	`TracingMiddleware`, exposed at `/v1/runs/{run_id}/trace`
OpenTelemetry spans	Distributed-tracing spans across SDK-driven agent / tool calls **and** platform-side operations (queue dispatch, worker lease, continuation lock)	`kneo_agent.observability.OpenTelemetryMiddleware` + `kneo_serv.observability.platform_tracer`, opt-in via `KNEO_SERV_OTEL_ENABLED`

There is **no metrics endpoint** in the 0.4.x line. Scrape latency and error rate from your reverse-proxy access logs or from OTel span attributes.

Structured request logs

Shape

Each request emits a single JSON record on the `kneo_serv.service` logger:

```
{
  "client_ip": "10.0.0.7",
  "duration_ms": 18.214,
  "event": "http_request",
  "method": "POST",
  "path": "/v1/runs",
  "request_id": "f3b3…",
```

```
  "run_id": "run_…",
  "status_code": 201,
  "user_agent": "kneo-serv-client/0.2.2"
}
```

Fields the middleware always emits: `event`, `request_id`, `method`, `path`, `status_code`, `duration_ms`. Optional fields when available: `client_ip`, `user_agent`. Route-derived fields when the path includes them: `run_id`, `continuation_id`. When the request raises: `error` (exception class name) and `message` (exception message). Redaction is applied to every payload before it reaches the log line. (`kneo_serv/observability/structured_logging.py`)

Configuration

Variable	Default	Purpose
`KNEO_SERV_REQUEST_LOGS`	`true`	Enable the JSON request log middleware.
`KNEO_SERV_LOG_LEVEL`	`INFO`	Service logger level.

`request_id` is generated server-side as a UUID unless the client sends `X-Request-ID`; either way the service echoes it back on the response header.

Production tuning

- Keep `KNEO_SERV_LOG_LEVEL=INFO` in production. `DEBUG` doubles log volume and can leak diagnostic payloads from middleware that wraps the request logger.
- Configure your container runtime's log driver (Docker `json-file` with rotation, Kubernetes `kubectl logs` rotation, journald) — the service writes to stdout and relies on the runtime to rotate.

Log aggregation wiring

- **ELK / OpenSearch.** Ship stdout via Filebeat or Vector. The records are already JSON; map `request_id` and `run_id` as indexed fields. Pin `service.name=kneo-serv` from the shipper for cross-deployment search.
- **Loki.** A Promtail pipeline with a `json` stage will lift `request_id`, `run_id`, `status_code`, and `duration_ms` to labels. Keep label cardinality bounded — don't promote `request_id` to a Loki label, query it as content.
- **Cloud-managed (CloudWatch Logs, GCP Logging).** Forward stdout; the managed pipeline parses JSON automatically.

The reverse proxy in front of `kneo-serv` (`tls_and_proxy.md`) has the true client IP. The service logs the immediate TCP peer; correlate to the proxy's access logs by `request_id` (forward `X-Request-ID` upstream).

Service-side trace events

Service-side trace events are persisted as part of run state and returned at `GET /v1/runs/{run_id}/trace`. They cover workflow step transitions, tool calls, checkpoints, and audit boundaries. These events are emitted by `TracingMiddleware` independent of any OTel exporter, so they are always available even without OpenTelemetry.

See `service_api.md § Audit events` and `service_api.md § Replay and checkpoint diff` for the contract.

OpenTelemetry spans

When the deployment includes the SDK telemetry support (the `[telemetry]` or `[deploy]` extras), set `KNEO_SERV_OTEL_ENABLED=true` to attach `kneo_agent.observability.OpenTelemetryMiddleware`. Argument and result capture (`KNEO_SERV_OTEL_RECORD_ARGUMENTS`, `KNEO_SERV_OTEL_RECORD_RESULTS`) are off by default because tool inputs and outputs frequently contain user payloads — enable them only after the deployment's data classification has approved payload capture.

See `environment.md § Observability` for the full env-var reference.

Platform-side spans

The SDK's `OpenTelemetryMiddleware` covers the *agent boundary* — runs, tool calls, model calls. The platform also instruments operations that happen *outside* the agent's execution:

Span name	Where	Attributes
`kneo.queue.dispatch`	`PlatformManager.dispatch_run` — when a run is enqueued for an async worker	`kneo.run.id`
`kneo.worker.lease`	Async worker loop — one span per lease attempt against the queue	`kneo.worker.id`, `kneo.worker.lease_seconds`, `kneo.worker.claimed` (bool), `kneo.run.id` (if claimed)
`kneo.continuation.lock`	`PlatformManager.resume_human_task` — when the per-	`kneo.continuation.id`, `kneo.lock.name`,

Span name	Where	Attributes
	continuation lock is acquired before resume	`kneo.lock.ttl_seconds`, `kneo.lock.acquired` (bool)

These spans share the same `KNEO_SERV_OTEL_ENABLED` flag — they're a clean no-op when telemetry is off (no overhead beyond a single env-var check). Span names use the `kneo.<area>.<operation>` convention so they sort cleanly alongside SDK-owned spans in tracing UIs.

Lease spans with `kneo.worker.claimed=false` indicate an empty queue — useful for measuring how often workers idle. Continuation lock spans with `kneo.lock.acquired=false` correlate with the `LockAcquisitionError` shown in troubleshooting.md § 8.1.

Exporter configuration

The service uses the OpenTelemetry global tracer provider; exporters are configured with standard `OTEL_*` environment variables that the OTel SDK reads. Example for OTLP/HTTP to any compatible backend (Honeycomb, Grafana Tempo, Tempo Cloud, Datadog, an OTel Collector):

```
export KNEO_SERV_OTEL_ENABLED=true

export OTEL_EXPORTER_OTLP_PROTOCOL=http/protobuf
export OTEL_EXPORTER_OTLP_ENDPOINT=https://api.honeycomb.io
export OTEL_EXPORTER_OTLP_HEADERS="x-honeycomb-team=$HONEYCOMB_API_KEY"
export OTEL_SERVICE_NAME=kneo-serv
export OTEL_RESOURCE_ATTRIBUTES="deployment.environment=prod"
```

For a self-hosted OTel Collector running as a sidecar:

```
export OTEL_EXPORTER_OTLP_ENDPOINT=http://localhost:4317
export OTEL_EXPORTER_OTLP_PROTOCOL=grpc
```

If OTel does not appear to be exporting, see `troubleshooting.md § 7.2`.

What to watch in production

A minimal alerting baseline covers the failure modes that page on-call:

Signal	What it means	Where to read it
`/readyz` returns `503` for more than 1 probe interval	A dependency check is failing	Reverse proxy / load balancer health checks

Signal	What it means	Where to read it
Sustained `5xx` rate above baseline	Service-side errors	Proxy access logs; `status_code` from JSON logs
`duration_ms` p95 climbing over baseline	Latency regression — provider, queue, or DB pressure	JSON log records
Queue depth (`status=queued`) growing unbounded	Workers stuck or backpressured	`/readyz` `queue` check; `curl "$BASE/v1/runs?status=queued&limit=20"`
Spike in `event=http_request` records with `error`	Application-level exceptions	JSON log records

Wire your alerting against these signals from the proxy and the aggregated logs; the service does not push its own alerts.

What this page does not cover

- **Per-IP rate limiting and traffic shaping.** The reverse proxy's job (`tls_and_proxy.md`).
- **A Prometheus `/metrics` endpoint.** Not provided in the 0.4.x line. Use OTel spans to derive request rate, error rate, and latency, or scrape the reverse proxy.
- **Tracing internals.** For the design of the in-process tracer and checkpoint events, see `docs/dev/design.md` and `docs/dev/implementation_map.md`.

Backup and recovery

Source: *`docs/user/backup_and_recovery.md`*

Production procedure for backing up `kneo-serv` state, verifying restores, and rolling back a deployment. This page consolidates the operator surface; the underlying Python API and SQL commands stay in their respective references.

For the upgrade context that ends in "and keep a backup", see `upgrade.md`. For the Python backup API used by the SQLite maintenance helpers, see `service_api.md § Backup and restore`.

What needs to be preserved

State	Where it lives	Backup mechanism
Run state, queue, checkpoints, audit events, idempotency, locks, policies	PostgreSQL (`KNEO_SERV_DATABASE_URL` set) or SQLite at `.kneo/kneo_runs.sqlite` (default)	`pg_dump` / SQLite online-backup
Workflow continuations	PostgreSQL when set, otherwise files under `.kneo/continuations/`	DB dump or filesystem backup
Spec bundles	Source repo + your CI artifacts (signed bundles)	Repo + artifact store
Artifacts (workflow outputs)	Filesystem paths declared by your specs	Filesystem backup
Logs	stdout via container log driver → log aggregator	Aggregator retention

The DB is the load-bearing piece. Everything else can be reconstructed from the DB and your spec repo, except for filesystem-stored continuations and artifacts when PostgreSQL is not configured.

PostgreSQL — production path

The Compose stack and any production deployment should set `KNEO_SERV_DATABASE_URL`. In that mode all state above (except artifacts) lives in PostgreSQL.

Take a backup

```
docker compose --env-file deploy/production.env exec db \
  pg_dump -U "$POSTGRES_USER" "$POSTGRES_DB" \
  | gzip > "kneo_serv-$(date +%Y%m%d-%H%M).sql.gz"
```

For a host-level Postgres install, run `pg_dump` directly as the `postgres` user; the data shape is the same.

Restore from a backup *(destructive — wipes current state)*

```
gunzip -c kneo_serv-YYYYmmDD-HHMM.sql.gz \
  | docker compose --env-file deploy/production.env exec -T db \
      psql -U "$POSTGRES_USER" "$POSTGRES_DB"
```

Restore replaces every row in the database. Stop the API container first so no in-flight write races the restore:

```
docker compose --env-file deploy/production.env stop api
gunzip -c kneo_serv-YYYYmmDD-HHMM.sql.gz \
  | docker compose --env-file deploy/production.env exec -T db \
      psql -U "$POSTGRES_USER" "$POSTGRES_DB"
docker compose --env-file deploy/production.env start api
```

Off-site rotation

Local backups protect against operator error, not host loss. After each dump, copy the gzip off the host:

- S3, Azure Blob, or GCS bucket with versioning + lifecycle to archive older dumps.
- Encrypt at rest (server-side encryption is sufficient if your control plane is locked down; client-side encryption for stricter regimes).
- Apply a separate IAM identity for upload-only versus read.

Data-only restore into a clean volume

For test-restore drills and disaster recovery into a fresh PostgreSQL volume, the service handles schema migrations on startup. Capture a data-only dump and exclude the `schema_migrations` rows so the new volume's migration state isn't overwritten:

```
docker compose --env-file deploy/production.env exec -T db \
  pg_dump -U "$POSTGRES_USER" -d "$POSTGRES_DB" --data-only --inserts \
  -f /tmp/kneo_serv_data.sql
docker cp <db-container-id>:/tmp/kneo_serv_data.sql /tmp/kneo_serv_data.sql
```

```
grep -v "INSERT INTO public.schema_migrations" /tmp/kneo_serv_data.sql \
  > /tmp/kneo_serv_data_restore.sql
```

Restore into a clean volume after the API has come up at least once (so migrations have run):

```
 docker compose --env-file deploy/production.env down -v
docker compose --env-file deploy/production.env up --build -d
docker cp /tmp/kneo_serv_data_restore.sql \
  <db-container-id>:/tmp/kneo_serv_data_restore.sql
docker compose --env-file deploy/production.env exec -T db \
  psql -v ON_ERROR_STOP=1 -U "$POSTGRES_USER" -d "$POSTGRES_DB" \
  -f /tmp/kneo_serv_data_restore.sql
docker compose --env-file deploy/production.env restart api
```

`ON_ERROR_STOP=1` aborts the restore on the first failing INSERT so you don't end up with partial state.

SQLite — single-host installs

When `KNEO_SERV_DATABASE_URL` is unset, run state lives in `.kneo/kneo_runs.sqlite` and continuations in `.kneo/continuations/`. The service ships an online backup helper:

```
 from kneo_serv.maintenance import backup_sqlite_database, restore_sqlite_database

# Online — safe while the service is running
backup_sqlite_database(
    ".kneo/kneo_runs.sqlite",
    ".kneo/backups/kneo_runs-2026-05-12.sqlite",
)

# Restore into a new location, then swap into place during a window
restore_sqlite_database(
    ".kneo/backups/kneo_runs-2026-05-12.sqlite",
    ".kneo/kneo_runs.restored.sqlite",
)
```

`backup_sqlite_database` uses SQLite's `backup()` API and is safe to run against a live database. `restore_sqlite_database` is a plain file copy — stop the service before swapping the restored file into the live path, or you'll race the writer.

Also back up `.kneo/continuations/` and any artifact paths your specs write to; these are not inside the SQLite file.

Backup frequency

There is no single recommended cadence. Tie it to your retention policy and your tolerance for re-running work:

Workload shape	Cadence
Low run volume, short retention	Daily dump, 30-day retention
Active production, multi-day retention enabled (`KNEO_SERV_RETENTION_*`)	Hourly dump, 7-day retention; daily off-site copy
Audit-heavy compliance workloads	Per-hour dump kept for the compliance window; verified test-restore monthly

The relevant env vars are in `environment.md § Retention`. A retention policy that prunes runs after 7 days needs backups newer than 7 days, or the restore set is empty.

Verifying a restore

Backups are unproven until they have been restored. Verify on the schedule below, not after a real incident.

1. Provision a scratch host or namespace and restore the backup into it.
2. Start `kneo-serv` against the restored database.
3. Verify dependencies: `bash curl -sf http://127.0.0.1:8000/readyz | jq '.metadata.ready' # → true`
4. Verify a known run survived: `bash curl -sf "http://127.0.0.1:8000/v1/runs?limit=5" \ -H "Authorization: Bearer $OP_TOKEN" | jq '.runs[].run_id'`
5. Run the deployment smoke against the restored stack (`deployment_smoke.md`). It exercises run create → fetch → cancel and confirms checkpoints persist.
6. Verify audit events from before the backup are present: `bash curl -sf "http://127.0.0.1:8000/v1/audit-events?limit=5" \ -H "Authorization: Bearer $OP_TOKEN" | jq '.events[].event_type'`

Recommended cadence: monthly restore drill into a scratch environment, plus a restore drill immediately before any major upgrade.

Rolling back after a failed upgrade

Migrations are schema-forward and not safe to downgrade in place. If a new release misbehaves and the issue can't be patched forward:

1. **Stop the service.** Quiesce writers; the proxy can keep returning `503` from `/readyz` until step 5. `bash docker compose --env-file deploy/production.env stop api`

2. **Restore persistence** from the pre-upgrade backup using the PostgreSQL or SQLite procedure above.
3. **Re-install the previous version.** Pin the image tag or reinstall the pip package at the prior version. Update Compose / Kubernetes manifests accordingly.
4. **Restart the service.** `bash docker compose --env-file deploy/production.env start api`
5. **Verify** with `curl /readyz` and the deployment smoke (`deployment_smoke.md`).

Keep the pre-upgrade backup until you have verified the new version through at least one business cycle.

Disaster recovery checklist

Scenario	Recovery
Lost the host, database survived	Provision new host → install `kneo-serv` → point `KNEO_SERV_DATABASE_URL` at the surviving DB → start.
Lost the database	Provision DB → restore latest dump → start service → verify `/readyz` and a known run.
Lost host and database	Provision DB → restore latest off-site dump → provision host → start service → verify.
Corrupted checkpoints for one run	Use `GET /v1/runs/{run_id}/checkpoints/diff` to identify the bad checkpoint; cancel and re-run from the last good step. The DB itself is fine.
Restore brought back stale data, signs of mismatch	See `troubleshooting.md § 2.5` for the recovery shape.

What this page does not cover

- **Performance and capacity sizing.** Deferred until the benchmark suite ships — see `TODO-docs.md § Performance and capacity guide` .
- **The Python backup API surface.** Stays in `service_api.md § Backup and restore` .
- **Release-team verification gates** for the GA cut. Those live in `release_checklist.md` .

Upgrade guide

Source: `docs/user/upgrade.md`

Conventions for upgrading Kneo Agent Platform (`kneo-serv`) between releases, plus version-specific notes when a release has breaking changes.

For the release process itself (gates, tagging, artifacts), see release_checklist.md. For the supported `kneo_agent` SDK range, see sdk_alignment.md.

Versioning

Kneo Agent Platform follows semantic versioning:

- **Patch** (`0.1.0` → `0.1.1`): bug fixes; persistence schemas, route contracts, CLI commands, and env-var names do not change.
- **Minor** (`0.1.x` → `0.2.0`): additive changes. Persistence schemas may add new tables or columns with migrations; routes and CLI may add new surfaces. Existing surfaces remain available with the same shape unless the release notes call out an exception.
- **Major** (`0.x` → `1.0`): may remove or change surfaces. Read the release notes before upgrading; expect to update calling code.

The HTTP API is also versioned at the URL prefix (`/v1`); legacy unversioned routes remain available alongside `/v1`. See design.md § 13.

Standard upgrade procedure

1. **Read the release notes** for every minor/major version between your current and target version. Patch upgrades only need the latest patch's notes. Notes for the current release are at release_notes_0.1.0.md.
2. **Pin the target version** in your dependency manifest: `kneo-serv[deploy]==X.Y.Z`
3. **Stop traffic** to the service (or drain via a load balancer). Background runs that are queued will be reclaimed by the worker after restart; in-flight runs that complete during the drain will record normally.
4. **Back up persistence.** Follow `backup_and_recovery.md` (`pg_dump` for PostgreSQL, `backup_sqlite_database()` for SQLite). Keep the backup until you have verified the new version through at least one business cycle.
5. **Install the new version** in your deployment image or environment.
6. **Restart the service.** Migrations apply automatically at startup. Watch the structured log for `migration` events and any `migration_failed` errors.
7. **Verify** with `GET /readyz` and the deployment smoke script: `bash python scripts/deployment_smoke.py --base-url http://<host>:<port>`

8. **Resume traffic.**

If `GET /readyz` does not return 200 within a few seconds of restart, see troubleshooting.md § 1.2.

Persistence migrations

Every store that has a schema (`SQLiteRunStateStore`, `PostgresRunStateStore`) tracks its schema version and applies forward-only migrations on first connection. Migrations are idempotent and never drop columns or rows on their own. The file-based stores have no schema; they tolerate older record shapes through the row decoder.

If a migration fails, the service refuses to serve requests rather than running on a partially-migrated schema. Fix the underlying cause (usually a permissions or disk-space problem), then restart.

Downgrades are not supported. Restore from backup if you need to revert.

For contributors authoring new migrations (conventions, the dialect portability rules, the test patterns), see `docs/dev/migrations.md`.

Spec migrations

The YAML spec format is versioned at `version: v1`. The compiler accepts older shapes through automatic normalization, but for clarity the CLI can write upgraded specs to disk:

```
kneo spec migrate legacy_agent.yaml --output migrated_agent.yaml
kneo spec migrate migrated_agent.yaml --check --json
```

Specs that pass `kneo spec validate` on the source version will continue to compile after upgrading; specs that hit deprecation warnings should be migrated proactively before a future release removes the fallback.

Signed bundles created with `kneo spec bundle sign` are tied to the signing key, not the kneo-serv version, so bundles signed before an upgrade continue to verify after as long as the signing key is unchanged.

SDK compatibility

`kneo-serv` declares a `kneo-agent` range in `pyproject.toml`. When upgrading kneo-serv, let pip resolve the matching SDK; do not pin SDK versions outside that range. The compatibility tests (`tests/test_sdk_compatibility.py`) assert the SDK surface used by the service, so a version mismatch surfaces as a test failure.

If you maintain custom runtimes or middlewares that import directly from `kneo_agent`, run those compatibility tests after upgrading and update imports in lockstep.

Configuration changes

Environment-variable names and defaults are part of the public surface. Changes are recorded in environment.md and called out in release notes:

- New variables default to behavior consistent with the previous release.
- Renamed variables retain a deprecation alias for at least one minor release; a startup warning is emitted when the alias is used.
- Removed variables are removed only at major versions.

After upgrading, diff your env file against the latest deploy/production.env.example (or `staging.env.example`) to spot any new optional variables.

CLI changes

The `kneo` CLI is regenerated each release; see cli_reference.md for the current shape. New subcommands are additive within minor releases. Subcommand behavior may change at major releases — check the release notes.

CLI profiles stored at `~/.kneo_serv/profiles.json` carry forward across releases. The profile schema is itself versioned and migrated in place.

Version-specific notes

This section grows as releases ship. Each entry should describe what changed, what action operators must take, and how to verify the upgrade.

0.1.0 — initial release

No upgrade applies; this is the first published version. See release_notes_0.1.0.md for scope, capabilities, and verified release-candidate steps.

0.2.0 — first public distribution

This is the first cut to publish a real `kneo-serv` package. 0.1.0 and 0.1.1 shipped as GitHub Release artifacts only; 0.2.0 is the first version available via `pip install kneo-serv` and `docker pull ghcr.io/kneo-agent/kneo-serv`.

Version trajectory on PyPI: `0.0.0 → 0.2.0`. The `kneo-serv 0.0.0` placeholder published on 2026-05-14 reserved the distribution name; it shipped an empty importable module with no `kneo` CLI binary (no `[project.scripts]` entry). Any user who tried `pip install kneo-serv && kneo --version` during the placeholder window saw `kneo: command not found` — 0.2.0 is the first cut to install the binary. The placeholder is yanked once 0.2.0 ships; existing explicit `==0.0.0` pins still resolve, but default `pip install kneo-serv` jumps straight to 0.2.0.

Install paths: - `pip install kneo-serv` — first time this works end-to-end. - `docker pull ghcr.io/kneo-agent/kneo-serv:0.2.0` (and `:0.2`, and `:latest`) — first time the image is available without a local build.

Deployment migration for operators on 0.1.x using `compose.yaml` with the bundled `build: context: .`: - Default flow becomes `docker compose pull && docker compose up -d` against the GHCR image. - The `build:` block stays in `compose.yaml` for contributors and the CI smoke test (`docker compose up --build`). - No required changes to `deploy/production.env` or `deploy/staging.env` from 0.1.1.

Persistence schemas: unchanged from 0.1.1. No migrations required.

Feature additions visible to operators (full per-feature detail in `release_notes_0.2.0.md`): - `kneo spec lint` — CI-friendly validator subcommand that exits non-zero on any warnings or errors. - Retention windows now live in `.kneo/config.yaml` under a `retention:` block, with env vars as the operator override. - Human-task expiration via `PlatformManager.prune_expired_human_tasks()` — paused runs whose human-step deadline has passed transition to a new `expired` status and emit `human.expired` audit events. - Two new reference example specs: `concurrent_review_workflow.yaml` and `group_chat_workflow.yaml`. - Docker-based local PostgreSQL integration testing via `python scripts/postgres_test.py`.

No breaking changes to spec syntax, HTTP API contracts, CLI command names, env-var names, or persistence schemas. Specs that validated under 0.1.1 continue to validate under 0.2.0.

0.2.1 — `/healthz` version and Docker `/app` permission fix

Patch release fixing two regressions discovered while smoke-testing the published 0.2.0 image. Both are bug fixes; no new features, no contract changes.

Upgrade: - `pip install -U kneo-serv` (resolves to `0.2.1`). - `docker pull ghcr.io/kneo-agent/kneo-serv:0.2.1` — `:0.2` and `:latest` now resolve to the 0.2.1 digest.

What was broken in 0.2.0: - `GET /healthz` returned `"version":"0.1.0"` from the 0.2.0 image because `HealthResponse.version` was a hardcoded string literal. 0.2.1 resolves the field dynamically via `importlib.metadata.version("kneo-serv")`. - Plain `docker run -p 8000:8000 ghcr.io/kneo-agent/kneo-serv:0.2.0` crashed on startup with `PermissionError: [Errno 13] Permission denied: '.kneo'` because `/app` was root-owned but the container drops to the non-root `kneo` user before creating the SQLite-fallback path. 0.2.1 adds `chown -R kneo:kneo /app` to

the install layer. The Docker Compose deployment path was unaffected (it pins `KNEO_SERV_DATABASE_URL` to PostgreSQL).

Persistence schemas: unchanged from 0.2.0. No migrations required.

No breaking changes to spec syntax, HTTP API contracts, CLI command names, env-var names, or persistence schemas.

0.2.2 — FastAPI `info.version` fix + post-0.2.0 docs sweep

Patch release fixing one regression in the same family as 0.2.1 plus a documentation sweep. No feature changes, no contract changes, no schema changes.

Upgrade: - `pip install -U kneo-serv` (resolves to `0.2.2`). - `docker pull ghcr.io/kneo-agent/kneo-serv:0.2.2` — `:0.2` and `:latest` now resolve to the 0.2.2 digest.

What was broken in 0.2.1: - `GET /openapi.json` returned `info.version: "0.1.0"` from the 0.2.1 image because the FastAPI app constructor in `kneo_serv/service/app.py` still pinned a hardcoded literal. The 0.2.1 cut fixed `HealthResponse.version` but missed this parallel occurrence. 0.2.2 resolves both via the same `importlib.metadata.version("kneo-serv")` helper, called at app-construction time.

Documentation: - Forward-looking plan docs and "as of 0.1.0" framing in user/dev docs swept to match the 0.2.x shipped reality. No content lost — historical files (CHANGELOG entries, shipped release notes, `TODO-0.2.0.md`, ADRs) are unchanged.

Persistence schemas: unchanged from 0.2.1. No migrations required.

0.3.0

Next additive minor on the 0.2.x line. No breaking changes to spec syntax, HTTP API contracts, CLI command names, env-var names, or persistence schemas. Full narrative in `release_notes_0.3.0.md`.

Upgrade: - `pip install -U kneo-serv` (resolves to `0.3.0`). - `docker pull ghcr.io/kneo-agent/kneo-serv:0.3.0` — `:0.3` and `:latest` now resolve to the 0.3.0 digest. The image is now signed (cosign keyless via Sigstore) and ships with a CycloneDX SBOM attestation; verification commands are in `supply_chain_review.md § Verification commands`.

SDK floor bump: - The `kneo-agent` SDK floor moves from `>=1.1.1` to `>=1.2.0`. Pip auto-resolves on `pip install -U kneo-serv`, but operators pinning the SDK separately (e.g. via a constraints file or a monorepo lockfile) must ensure their install is on `1.2.0` or newer. The compat test suite passed against `kneo-agent 1.2.0` throughout the 0.2.x line; the floor was kept low to avoid forcing 0.1.x users to upgrade. 0.3.0 is the natural inflection point to lift it.

New `timed_out` lifecycle status: - Runs that hit their run-level deadline transition to a new terminal `timed_out` status (alongside `completed`, `failed`, `cancelled`, `expired`). Operator tooling that switches on `state.status` should accept it as terminal — e.g. dashboards, alerting rules, retention sweeps (which the platform's own `RetentionPolicy.run_statuses` already includes). - The `error.type` field on a timed-out run is `run_timed_out`, distinct from `human_task_expired` (which the existing `expired` status uses).

New runtime surfaces: - `start_run_from_spec(..., timeout_seconds=N)` and `run_from_spec(..., timeout_seconds=N)` accept an optional wall-clock deadline. Operator-callable `PlatformManager.prune_timed_out_runs()` walks runs and force-cancels those past their deadline. Same operator-cron pattern as `prune_retention()` and `prune_expired_human_tasks()` — no built-in scheduler. - The human-task `on_timeout: continue` and `on_timeout: escalate` literals are now wired in the runtime (they were accepted by the spec but silently treated as `fail` in 0.2.x). Operators with specs that declared these literals will see the documented behaviour for the first time. Audit consumers should expect new event types: `human.continued`, `human.continue_failed`, `human.escalated`, `run.timed_out`. - New route `GET /v1/runs/{run_id}/policy-report` returns the spec policy report for a stored run, no spec bundle required client-side. Auth: `specs:read` scope (same as the existing `POST /v1/specs/policy-report`).

New observability surfaces: - Three new platform-side OpenTelemetry spans (`kneo.queue.dispatch`, `kneo.worker.lease`, `kneo.continuation.lock`) join the SDK's agent-boundary spans when `KNEO_SERV_OTEL_ENABLED=true`. Pre-existing OTel pipelines pick them up automatically once telemetry is enabled — no extra configuration required. See `observability.md § Platform-side spans`.

Persistence schemas: unchanged. The new `RunState.deadline_at` and `Checkpoint.iteration` fields default to `None` and `1` respectively in the dataclass, so existing rows round-trip cleanly through the JSON-payload SQLite / PostgreSQL stores.

0.4.0

Next additive minor on the 0.3.x line. **No breaking changes** to spec syntax, HTTP API contracts, CLI command names, env-var names, or persistence schemas. Specs that validated under 0.3.x continue to validate under 0.4.0. The cut is a **docs + tooling release** — runtime semantics are identical to 0.3.0. Full narrative in `release_notes_0.4.0.md`.

Upgrade: - `pip install -U kneo-serv` (resolves to `0.4.0`). - `docker pull ghcr.io/kneo-agent/kneo-serv:0.4.0` — `:0.4` and `:latest` now resolve to the 0.4.0 digest. Image continues to be signed (cosign keyless via Sigstore) and ships with a CycloneDX SBOM attestation; the 0.4.0 cut adds a Trivy CVE scan report attached to the GitHub Release. Verification commands are in `supply_chain_review.md § Verification commands`.

SDK floor: unchanged. The `kneo-agent` floor stays at `>=1.2.0` — same as 0.3.0. No operator action required for operators pinning the SDK separately.

New auto-generated API reference: the docs site at `kneo-agent.github.io/kneo-serv/` gains a new top-level **API Reference** nav section with 17 pages (16 subpackages + `sdk`), rendered at build time by `mkdocstrings` from the Python docstrings. Operator surface unchanged — the API ref is a **developer lookup surface**, not a runtime change. See `docs/api/README.md` for the index.

Image vulnerability scanning (Trivy): the release pipeline now scans the pushed GHCR image with Trivy under the CVSS≥7 policy (HIGH/CRITICAL findings block the publish step). On every release-tag build, the JSON scan report is attached to the GitHub Release as the `trivy-report-<version>` artifact, 90-day retention. Deployers can re-run the scan locally with `trivy image ghcr.io/kneo-agent/kneo-serv:<tag>`; full policy + escape hatch documented in `security_hardening.md § Image vulnerability scanning`.

Developer-facing changes (no operator surface impact): - Ratcheting ruff D-rule gate (`D100` / `D101` / `D102`) now enforced project-wide for `kneo_serv/**/*.py`. New public classes / methods without docstrings fail CI. Forks adding code should follow the Google docstring convention; the chain-reference files are `security/secrets.py` and `platform/manager.py`. - Full mypy strict coverage across `kneo_serv/`. The `[[tool.mypy.overrides]]` block in `pyproject.toml` now covers every public module. Forks that subclass or extend public types should expect `disallow_untyped_defs` + `warn_return_any` + `strict_equality`. - `mkdocstrings[python]>=0.27` added to the `docs` optional-dep block. Operators using `pip install kneo-serv` (without `[docs]`) are unaffected — the dep is build-time only for the rendered site.

New 0.3.0-feature worked examples: - `examples.md` picked up a *Timeout branches* subsection on the `human_approval_workflow.yaml` entry covering the `on_timeout: fail/continue/escalate` literals (all wired since 0.3.0). - New `examples/run_with_timeout.py` walks through `start_run_from_spec(..., timeout_seconds=N)` + `prune_timed_out_runs()`. Companion to the human-task timeout example above.

Persistence schemas: unchanged. No new fields, no migrations.

Rolling back

Schema-forward migrations make in-place downgrade unsafe; the only supported rollback path is **restore from the pre-upgrade backup**, then re-install the previous version.

For the full step-by-step procedure — stop, restore, re-install, restart, verify with the deployment smoke — see `backup_and_recovery.md § Rolling back after a failed upgrade`.

Keep the pre-upgrade backup until you have verified the new version through at least one business cycle.

Reporting upgrade issues

Capture the same context listed in troubleshooting.md § What to capture before opening a bug, plus:

- Source version (`pip show kneo-serv` before the upgrade).
- Target version (after the upgrade).
- Migration log lines from the first start on the new version.
- The exact env file or compose `.env` (with secrets redacted).

Incident response

Source: *`docs/user/incident_response.md`*

On-call entry point for `kneo-serv`. This page is the triage tree — "the service looks wrong, where do I look first?" The symptom-by-symptom deep dive lives in `troubleshooting.md` ; this page sends you there.

For backup and rollback procedures, see `backup_and_recovery.md` . For the API definition of the health endpoints, see `service_api.md § Health checks` .

Triage tree

```
    ┌── /healthz returns 200 ──────  service process is alive
    │                               ─ check /readyz next
    │
1.  ├── /healthz times out ────────  process is down or unreachable
    │                               ─ check the container / supervisor
    │                               ─ check the reverse proxy upstream
    │
    └── /healthz 5xx ──────────────   application crashed mid-request
                                     ─ check stderr / container logs
                                     ─ see troubleshooting.md § 1

    ┌── /readyz returns 200 ──────  dependencies healthy
    │                               ─ problem is in a specific run/spec
    │                               ─ check the run path below
    │
2.  ├── /readyz returns 503 ──────   read the `metadata.checks` payload
    │                               ─ use the matrix below to find the
    │                                 failing dependency, then jump to
    │                                 the matching troubleshooting § n
    │
    └── /readyz times out ──────────  same path as /healthz timeout above
```

`/healthz` and `/readyz` are unauthenticated by design — you can probe them from anywhere you can reach the service port.

```
 curl -sf http://<host>:<port>/healthz | jq
curl -sf http://<host>:<port>/readyz  | jq '.metadata.checks'
```

`/readyz` failure matrix

`/readyz` runs eight checks; when any fails, the response is `503` with `{"error": "not_ready", "metadata": {"checks": {...}}}`. The keys you will see and what each means:
(kneo_serv/service/routes_health.py)

Check key	What it probes	Common failure	Where to go
`api`	API wiring sentinel	Never fails on its own	If it does, the manager isn't configured — § 1.3
`run_state_store`	`manager.run_state_store.list_runs(limit=1)` succeeds	DB unreachable, schema missing, credentials wrong	§ 2.1 / § 2.2
`continuation_store`	`manager.continuation_store.list()` succeeds	File path missing, permissions wrong, DB issue	§ 2
`queue`	`list_queued_runs(status=…)` returns for `queued` / `running` / `failed`	Queue table missing or DB stall	§ 2, § 5.1
`runtime_registry`	Number of registered runtimes (declared via factories)	Empty registry — service started without runtimes	`extending.md` for runtime registration
`tool_registry`	Number of registered tools	Empty registry — service started without tools	`extending.md`
`providers`	Secrets named in `KNEO_SERV_HEALTH_PROVIDERS` resolve	Provider env var missing or empty	§ 3.2
`mcp`	Secrets named in `KNEO_SERV_HEALTH_MCP_SECRETS` resolve	MCP secret missing	§ 3.2

The payload includes per-check `error` (exception class) and `message` for failed checks, so you usually don't have to guess which subsystem is at fault — copy the message into the relevant troubleshooting section.

Common production incidents

If `/healthz` and `/readyz` are both green but the service is "wrong":

Symptom	First check	Deep dive
All requests return `401`	`Authorization` / `X-Kneo-Api-Key` header is present and valid	§ 4.1, § 4.3
Specific consumer returns `403`	The key's role/scope covers the route	§ 4.2, `security_hardening.md` § 2
Async runs stuck in `queued`	Worker process is up; queue table reachable	§ 5.1
Runs hang mid-workflow	The step's tool or provider call is timing out	§ 5.3
`409 idempotency_key_conflict`	Caller is reusing a key with a different payload	§ 5.4
Tool reports `MissingSecretError`	Provider secret env var is set on the service host	§ 3.1
Logs missing `request_id`	You are reading the right logger (`kneo_serv.service`), not raw uvicorn	§ 7.1, `observability.md`
OpenTelemetry exporter silent	`KNEO_SERV_OTEL_ENABLED=true` and `[telemetry]` extra installed	§ 7.2, `observability.md`
Human task `409 resource_locked`	Another resume is in flight for the same continuation	§ 8.1
Restored backup but state looks stale or mismatched	Stop, re-verify the dump source, follow the recovery shape	§ 2.5, `backup_and_recovery.md`

What to capture before escalating

When the runbook doesn't have an entry that fits, capture this context before paging the on-call developer. It is the same set `troubleshooting.md` asks for in a bug report:

- Service version and commit (`pip show kneo-serv` or the image tag).
- Environment context (`uname -a`, Python version, Postgres version).
- The `request_id` and `run_id` of an affected request.

- `GET /readyz` body, even when it returns `200`.
- For run-shaped problems: `GET /v1/runs/{run_id}`, `GET /v1/runs/{run_id}/trace`, and `GET /v1/runs/{run_id}/checkpoints` (redacted output is fine to share).
- For spec-shaped problems: the output of `kneo spec validate <path> --json`.
- Recent audit events that mention the affected resource: `GET /v1/audit-events?run_id=<id>`.

When to roll back

If the incident started immediately after a deploy and isn't covered by the matrix above:

1. Confirm the deploy is the trigger — diff the running version against the previous version, check the time correlation with the first 503/5xx.
2. If yes, follow `backup_and_recovery.md § Rolling back after a failed upgrade`.

Rolling back when the trigger isn't the deploy throws away forward progress. Diagnose first.

What this page does not cover

- **Severity definitions and paging policy.** Owned by your on-call rota, not by `kneo-serv`.
- **Post-incident review template.** Out of scope.
- **Failure modes during a `kneo-serv` release itself** — those are in `release_checklist.md` and `troubleshooting.md § 9`.

Troubleshooting

Source: *docs/user/troubleshooting.md*

An operator-facing runbook indexed by symptom. Each entry lists the symptom, how to confirm the root cause, and the fix; cross-references point at the authoritative configuration doc when one exists.

If you are responding to a live incident and don't yet have a symptom, start at `incident_response.md` — it walks `/healthz` → `/readyz` → the right section here. This page is the symptom-indexed deep dive.

When you're not sure where to start, check `GET /readyz` (§ 1.2) — it exposes the per-dependency status the service uses internally, and most "service is unhealthy" tickets resolve to one of its check entries.

1. Service won't start or won't accept traffic

1.1 `RuntimeError: KNEO service auth is enabled but no API keys are configured`

The service refuses to start when auth is enabled without keys. `service/auth.py`

- Confirm: check the startup log for the message above.
- Fix: set `KNEO_SERV_API_KEYS` (entries are `name:key:role_or_scope[,role_or_scope]`, semicolon-separated) and/or `KNEO_SERV_ADMIN_API_KEY`. To run without auth, set `KNEO_SERV_AUTH_ENABLED=false` (only for local dev).
- Reference: environment.md § Service Auth, production_readiness_review.md § Role Boundary Review.

1.2 `/readyz` returns 503

`GET /readyz` returns `{"error": "not_ready", "metadata": {"checks": {...}}}` when any dependency check fails. `service/routes_health.py`

- Confirm: `curl -sf http://<host>:<port>/readyz | jq`. Each per-dependency entry has `ok: false` plus `error` and `message` for failed checks.
- Fix: the per-check failure matrix (which check maps to which recovery action) lives in `incident_response.md § /readyz failure matrix`. In summary: store failures are covered by §2; provider/MCP secret failures by §3.

1.3 `RuntimeError: PlatformManager has not been configured`

The default `app` factory configures the platform manager automatically. This error appears when you pass `configure_default_manager=False` to `create_app()` and never call `set_platform_manager()` before serving. `service/dependencies.py`

- Fix: either drop the override, or call `kneo_serv.service.dependencies.set_platform_manager(...)` before the first request.

1.4 `RuntimeError: Invalid KNEO_SERV_API_KEYS entry`

The format is `name:key:role_or_scope[,role_or_scope]` per entry, separated by semicolons. Whitespace inside entries is trimmed; missing colons trigger this error. `service/auth.py`

- Fix: re-render `KNEO_SERV_API_KEYS`. Examples:
- `operator:OP_TOKEN:operator;reviewer:REV_TOKEN:reviewer`
- `svc:SVC_TOKEN:runs:write,human:read,human:write`
- Reference: environment.md § Service Auth, production_readiness_review.md § Route Scope Matrix.

2. Persistence and store failures

2.1 PostgreSQL DSN configured but service falls back to SQLite

The service uses PostgreSQL only when `KNEO_SERV_DATABASE_URL` is set **and** the `[postgres]` or `[deploy]` extra is installed. See `service/factory.py` .

- Confirm: in a dev shell, `python -c "import psycopg; print(psycopg.__version__)"` .
- Fix: install `kneo-serv[deploy]` (Docker image already does this), or `kneo-serv[postgres]` if you don't need telemetry.

2.2 `psycopg.OperationalError` on startup or first request

The DSN can't connect. Common causes: wrong host, missing TLS, wrong credentials, database not yet created.

- Confirm: `psql "$KNEO_SERV_DATABASE_URL" -c '\dt'` from the same network context as the service.
- Fix: correct the DSN, ensure the database exists, and confirm the user has privileges. `KNEO_SERV_DATABASE_URL` must be a libpq-style URL.

2.3 SQLite `database is locked` errors

Concurrent writes to a single SQLite file can collide. The default service worker is single-threaded per process; this typically appears when running multiple service processes against the same SQLite file.

- Fix: switch to PostgreSQL (set `KNEO_SERV_DATABASE_URL`). Multi-process SQLite is not a supported deployment topology.

2.4 Schema migration appears to have run but old data is missing

Migrations are idempotent and version-tracked per store; they don't drop data. If rows look missing after an upgrade, check whether you actually upgraded the same database the service is reading.

- Confirm: compare `KNEO_SERV_DATABASE_URL` (or SQLite path) between the upgrade context and the running service. A stale state file at `.kneo/kneo_runs.sqlite` is a common cause.

2.5 Backup/restore mismatch

`kneo_serv.maintenance.backup_sqlite_database()` produces a file copy that restore expects to find on the same SQLite version line. Restoring across incompatible SQLite versions can fail.

- Fix: align `sqlite3` versions, or migrate to PostgreSQL where backup goes through `pg_dump` / `pg_restore`. See staging_release_runbook.md and release_checklist.md for the seeded recovery drill.

3. Secrets, credentials, and provider integration

3.1 `MissingSecretError` on agent run

Provider keys, MCP credentials, and runtime settings are resolved through env-var references in project config; raw values are never stored. `security/secrets.py`

- Confirm: `kneo config secrets --json` lists which references exist and whether each resolves. The endpoint `GET /security/credentials` exposes the same view (requires `credentials:read`).
- Fix: export the env var named in the error. Set `KNEO_SERV_REQUIRE_PROVIDER_SECRETS=true` to fail fast at startup instead of at first run.

3.2 `GET /readyz` reports missing provider/MCP secrets

Readiness reports the secrets named in `KNEO_SERV_HEALTH_PROVIDERS` and `KNEO_SERV_HEALTH_MCP_SECRETS`. These are operator-curated allowlists, so expect 503 if you list a secret that isn't actually exported.

- Fix: trim the list to secrets you actually use, or export the missing one.

3.3 `kneo spec bundle verify` fails

Bundle verification requires `KNEO_SERV_SPEC_SIGNING_KEY` to match the key used to sign. Bundles signed with a different key (or unsigned) fail verification.

- Fix: rotate the signing key consistently across signing and verifying hosts. The key is HMAC-only; do not commit it.

4. Authentication and authorization

4.1 `401 Unauthorized — A valid Kneo service API key is required`

The route requires auth and the request didn't carry a valid token. `service/auth.py`

- Confirm: send `Authorization: Bearer <key>` or `X-Kneo-Api-Key: <key>`.
- Fix: use one of the configured keys. The CLI service client reads `KNEO_SERV_API_KEY`. For multi-environment workflows use CLI profiles (`kneo config profile use ...`).

4.2 `403 Forbidden — Missing required scope: <scope>`

The token authenticated but the principal does not hold the scope the route requires. `service/auth.py`

- Confirm: the scope in the error body tells you exactly what is missing.
- Fix: assign the principal a role that includes the scope, or add the scope explicitly in `KNEO_SERV_API_KEYS`. See the route ↔ scope matrix in production_readiness_review.md.
- Common gotchas:
- `POST /specs/run` requires `runs:write`, not `specs:read`.
- Reviewer cannot create runs or change policies.
- Service role cannot mutate environment policies (only operator/admin).

4.3 Health endpoints work, all other routes 401

`/healthz`, `/livez`, and `/readyz` are intentionally unauthenticated for load-balancer probes. Everything else is gated by the auth dependency. This is by design; see production_readiness_review.md § Route Scope Matrix.

5. Run lifecycle problems

5.1 Async runs sit in `queued` and never progress

The platform worker is started by `create_default_platform_manager()` in `service/factory.py`. If a custom embedding skips `manager.start_worker()`, queued runs never drain.

- Confirm: `GET /runs?status=queued` shows queued items; `GET /readyz` shows the queue dependency as ok; the service log has no "worker" lines.
- Fix: ensure `start_worker()` is called in the host process after constructing `PlatformManager` directly, or use the default factory.

5.2 Cancelled run still finishes as `succeeded`

Cancellation is cooperative through `CancellationToken` and propagates only at unit-of-work boundaries. A step that completes between the cancel request and the next checkpoint will record its result, but `RunState` remains `cancelled` — the platform does not overwrite cancelled status with completed results.

- Confirm: `GET /runs/{run_id}` should still report `status: cancelled` even if the last checkpoint shows completion of a step.
- If `status` shows `succeeded` after a cancel, file an issue with the run id, the checkpoint timeline (`/runs/{run_id}/checkpoints`), and the trace (`/runs/{run_id}/trace`).

5.3 Run hangs at a workflow step

Workflow steps support `on_error: retry` , `max_retries` , and `timeout_seconds` . If a step has no timeout and the underlying provider/MCP call blocks, the step blocks too.

- Fix: set step-level timeouts, or set the global defaults `KNEO_SERV_PROVIDER_TIMEOUT_SECONDS` / `KNEO_SERV_MCP_TIMEOUT_SECONDS` .

5.4 `409 Conflict — idempotency_key_conflict`

`Idempotency-Key` was reused with a different request body for the same scope. `service/idempotency.py`

- Fix: pick a new key for the new request body, or reuse the same body for the original key. Idempotency records hash the canonical JSON of the request payload and replay the original response on match.

5.5 `400 Bad Request — invalid_idempotency_key`

`Idempotency-Key` headers must be 1–256 characters after trimming. `service/idempotency.py`

- Fix: shorten the key. UUIDs are sufficient.

6. Spec validation and compilation

6.1 `SpecCompilationError` with diagnostics

`SpecCompiler` raises this on either schema or semantic validation failure. `spec/compiler.py`

- Confirm: `kneo spec validate <path>` prints the same diagnostics with location info.
- Fix: address each diagnostic. Common causes:
- Missing/extra fields in `version: v1` shape.
- References to undefined components/tools.
- Memory/guardrail policy mis-shape.
- Migrate older specs with `kneo spec migrate <path> --output <new>`.

6.2 `ValueError: Tool '<name>' has no implementation`

The spec references a tool name that is not registered with the `ToolRegistry`. `spec/builder.py`

- Fix: register the tool (programmatically or via MCP import), or remove the reference. The default service registers example tools; toggle with `include_example_tools` when constructing `PlatformManager` directly.

6.3 Inline spec rejected with size error

Inline specs and overrides are bounded. `service/limits.py`

- Fix: tune the relevant limit (`KNEO_SERV_MAX_INLINE_SPEC_BYTES`, `KNEO_SERV_MAX_OVERRIDES_BYTES`, `KNEO_SERV_MAX_METADATA_BYTES`, `KNEO_SERV_MAX_BODY_BYTES`) or move the spec to a path on disk. Limits exist to keep the service from deserializing arbitrarily large payloads.

7. Observability

7.1 Structured logs missing `request_id`

The structured logging middleware always populates `request_id`; missing fields usually mean the log was emitted before `RequestLoggingMiddleware` attached, or you're reading raw uvicorn access logs instead of the service logger.

- Fix: filter by logger name `kneo_serv` or by JSON log format. Clients can supply `X-Request-ID` to override the generated id; the service echoes it on the response.

7.2 OpenTelemetry not exporting

The SDK `OpenTelemetryMiddleware` only attaches when both `KNEO_SERV_OTEL_ENABLED=true` and the `[telemetry]` extra is installed.

- Fix: install `kneo-serv[deploy]` (Docker image) or `kneo-serv[telemetry]`, set `KNEO_SERV_OTEL_ENABLED=true`, and ensure standard OTel exporter env vars (`OTEL_EXPORTER_OTLP_ENDPOINT`, etc.) are set.
- Tool arguments and results are **not** captured by default. Enable with `KNEO_SERV_OTEL_RECORD_ARGUMENTS=true` and/or `KNEO_SERV_OTEL_RECORD_RESULTS=true` only after you've confirmed the data classification allows payload capture.

7.3 Trace events missing for a run

Service-side trace events live in run metadata and at `/runs/{run_id}/trace`. They are emitted by `TracingMiddleware` and the in-process `Tracer`, independent of OTel. Missing events most often mean the run was never executed (e.g. queued and abandoned) or the spec disabled tracing.

- Fix: confirm the run reached `running` / `succeeded`. If the workflow middleware list omits `TracingMiddleware`, restore it (the default chain includes it).

8. Human-in-the-loop

8.1 `LockAcquisitionError` on resume

A `POST /human-tasks/{continuation_id}/resume` failed because another caller currently holds the resume lock for the same continuation. `platform/manager.py`

- Confirm: the error body identifies the lock name. The first caller is still in flight.
- Fix: wait for the in-flight resume to complete; do not retry blindly. Use idempotency keys on resume to make retries safe.

8.2 Continuation expired or missing

If the continuation store was rotated (e.g. `.kneo/continuations` recreated, or PostgreSQL row deleted), `/human-tasks/{continuation_id}` returns 404.

- Fix: the run cannot be resumed. Start a new run.

9. Release and supply chain

For release-flow issues (mypy, pip-audit, build, tag, publish), follow release_checklist.md and supply_chain_review.md. The release workflow at `.github/workflows/release.yml` emits the gate

that failed in its job summary.

What to capture before opening a bug

When a problem isn't covered above:

- Service version and commit (`pip show kneo-serv`, plus the git commit if installed from source).
- Environment context (`uname -a`, Python version, Postgres version if used).
- The `request_id` and `run_id` from logs.
- `GET /readyz` body.
- For run problems: `GET /runs/{run_id}`, `GET /runs/{run_id}/trace`, and `GET /runs/{run_id}/checkpoints` (redacted output is fine to share).
- For spec problems: the output of `kneo spec validate <path> --json`.

Audit events are accessible at `GET /audit-events` and frequently contain the operator action that preceded a fault.

Deployment smoke test

Source: `docs/user/deployment_smoke.md`

This smoke test validates a running service deployment through the public HTTP API. It uses the self-contained `examples/smoke_human_workflow.yaml` spec and covers health, readiness, auth behavior, spec validation, run creation, human resume, audit listing, credential inventory, and environment policy updates.

Compose stack

Prepare a production env file:

```
cp deploy/production.env.example deploy/production.env
```

`deploy/production.env` is intentionally ignored by source control. Replace all placeholder tokens and passwords before binding a deployment to a real network. For local CI/smoke runs, the documented placeholder values are used only to exercise the auth path.

Start the API and PostgreSQL:

```
docker compose --env-file deploy/production.env up --build -d
```

Run the smoke test against the unversioned routes:

```
python scripts/deployment_smoke.py \
  --base-url http://127.0.0.1:8000 \
  --api-key replace-admin-token \
  --operator-api-key replace-operator-token \
  --reviewer-api-key replace-reviewer-token \
  --viewer-api-key replace-viewer-token \
  --expect-auth
```

Run the same smoke test against the versioned routes:

```
python scripts/deployment_smoke.py \
  --base-url http://127.0.0.1:8000 \
  --api-prefix /v1 \
  --api-key replace-admin-token \
  --operator-api-key replace-operator-token \
  --reviewer-api-key replace-reviewer-token \
```

```
  --viewer-api-key replace-viewer-token \
  --expect-auth
```

Shut the stack down when finished:

```
docker compose --env-file deploy/production.env down
```

PostgreSQL coverage

The compose stack uses PostgreSQL by default through `KNEO_SERV_DATABASE_URL=postgresql://...@db:5432/...`, so the smoke path above also validates PostgreSQL-backed run state, checkpoints, continuations, queue records, locks, audit events, and project metadata.

For a separately managed PostgreSQL database, start the API with `KNEO_SERV_DATABASE_URL` set and run the same `scripts/deployment_smoke.py` commands against that service URL.

Staging and remote smoke

Prepare a staging env file from the example:

```
cp deploy/staging.env.example deploy/staging.env
```

Replace every placeholder token, database password, provider key, and telemetry endpoint before use. For compose-based staging rehearsals, set `KNEO_SERV_ENV_FILE` so the API container reads the staging file:

```
python scripts/validate_staging_env.py deploy/staging.env
```

The validator fails if required staging settings are missing, scoped API roles are incomplete, payload telemetry capture is enabled, provider secret checks are disabled, or placeholder values remain.

For repeatable staging gates, render the local file from secret-backed environment variables instead of editing it by hand:

```
 export KNEO_STAGING_API_KEYS="operator:<operator-token>:operator;reviewer:<reviewer-
token>:reviewer;viewer:<viewer-token>:viewer"
export KNEO_STAGING_ADMIN_API_KEY=<admin-token>
export KNEO_STAGING_SPEC_SIGNING_KEY=<signing-key>
export KNEO_STAGING_DATABASE_URL=<postgresql-dsn>
export KNEO_STAGING_POSTGRES_PASSWORD=<compose-db-password>
export KNEO_STAGING_OTEL_EXPORTER_OTLP_ENDPOINT=<otel-endpoint>
```

```
export KNEO_STAGING_OPENAI_API_KEY=<openai-key>
export KNEO_STAGING_MCP_API_KEY=<mcp-key>
python scripts/render_staging_env.py --output deploy/staging.env
```

```
KNEO_SERV_ENV_FILE=./deploy/staging.env \
  docker compose --env-file deploy/staging.env up --build -d
```

For a remote staging deployment, run the smoke script against the public staging URL with scoped keys:

```
export KNEO_STAGING_BASE_URL=https://staging.example.com
export KNEO_STAGING_OPERATOR_TOKEN=<operator-token>
export KNEO_STAGING_REVIEWER_TOKEN=<reviewer-token>
export KNEO_STAGING_VIEWER_TOKEN=<viewer-token>
python scripts/deployment_smoke.py --api-prefix /v1 --expect-auth
```

The operator key validates specs, creates runs, reads audit and credential inventory, and updates policy state. The reviewer key resumes the human task. The viewer key is optional; when supplied, the smoke verifies that policy writes are rejected with `403`.

The full staging release gate validates `deploy/staging.env`, derives the scoped smoke tokens from `KNEO_SERV_API_KEYS`, and runs the `/v1` remote smoke:

```
python scripts/staging_release_gate.py \
  --env-file deploy/staging.env \
  --base-url https://staging.example.com
```

The GitHub Actions `Staging Gate` workflow runs the same renderer and release gate from the `staging` environment secrets. Dispatch it with the deployed staging URL after the service is reachable.

See also

- `backup_and_recovery.md § Data-only restore into a clean volume` — the seeded recovery drill that pairs with the scoped Compose smoke above. Run the smoke first to populate run, checkpoint, audit, and policy state, then exercise the data-only restore to verify it survives a fresh volume.