k8s-scale-app-rs: Scale or Restart a Kubernetes Deployment from a CronJob

2026-07-06

A Rust CLI that scales or restarts a single Kubernetes Deployment from a CronJob — with cosign-signed images, SPDX SBOM attestation, and SLSA build provenance.

TL;DR

k8s-scale-app-rs is a Rust CLI that does exactly one of two things and then exits:

- **`scale`** — sets a Deployment's replica count to a fixed value via the `deployments/scale` subresource.
- **`restart`** — patches the pod template's `restartedAt` annotation, which is the same mechanism `kubectl rollout restart` uses.

Designed to run as a `CronJob`. Version 1.0.0 is published; container images and the Helm chart are on `ghcr.io`, signed with cosign (keyless via GitHub OIDC), and shipped with an SPDX SBOM plus SLSA build provenance.

Why I built it

The classic way to "run `kubectl scale` on a schedule in a cluster" is to put a `kubectl` binary into a Job image. That works, but it drags in a few things I did not want:

- The full `kubectl` binary (about 50 MB stripped) for one API call.
- A kubeconfig / ServiceAccount plumbing that is broader than the one operation actually needs.
- Enough YAML around the pod spec to hide the actual behavior behind a template.

A purpose-built tool is a natural fit for Rust here: a single self-contained binary — no OpenSSL, no runtime dependency beyond glibc and the CA trust store — and a ServiceAccount that only needs `deployments/scale: patch`. That is what k8s-scale-app-rs is.

KEDA and Argo Workflows solve a related but different problem — reacting to events or orchestrating multi-step pipelines — not "flip a replica count or a restart annotation on a fixed schedule." KEDA in particular has its own well-known scale-from-zero gap: a Prometheus-metric trigger can scale 1⟶N but not 0⟶1, because a Deployment at zero replicas emits no metric to react to (see the LLM inference series for the concrete failure mode). None of that machinery applies here — a CronJob calling a purpose-built binary on a fixed schedule has nothing to react to.

The same binary also covers restart. `kubectl rollout restart deployment/foo` under the hood is one patch on `spec.template.metadata.annotations["kubectl.kubernetes.io/restartedAt"]`. Two subcommands, same binary, same image.

What it does

The CLI is two subcommands sharing common arguments:

```
k8s-scale-app-rs scale   --deployment <NAME> --replicas <N> [--dry-run]
[--extra-ca-bundle <PATH>]
k8s-scale-app-rs restart --deployment <NAME>                [--dry-run]
[--extra-ca-bundle <PATH>]
```

Configuration is via environment variables, overridable by CLI flags:

Flag	ENV	Subcommands
`--deployment`	`K8S_SCALE_DEPLOYMENT`	both
`--namespace`	`K8S_SCALE_NAMESPACE`	both
`--replicas`	`K8S_SCALE_REPLICAS`	scale
`--dry-run`	`K8S_SCALE_DRY_RUN`	both
`--extra-ca-bundle`	`K8S_SCALE_EXTRA_CA_BUNDLE`	both

The `--namespace` flag exists, but the CronJob does not hardcode it. Instead, the pod reads its own namespace from the Downward API and passes it through as an environment variable. That keeps the ServiceAccount's Role safely namespace-scoped: the tool can only ever act on Deployments in the namespace where the CronJob itself runs. Not a general-purpose "scale anything anywhere" tool.

Design highlights

Minimal RBAC via the `scale` subresource

Setting replicas does not need `patch` on the full Deployment. Kubernetes exposes `deployments/scale` as a dedicated subresource:

```yaml
rules:
  - apiGroups: ["apps"]
    resources: ["deployments/scale"]
    verbs: ["get", "patch", "update"]
```

That is all the scale mode needs. Restart mode does need `patch` on `deployments` itself (to update the pod template annotation), so the shipped Role covers both:

```yaml
  - apiGroups: ["apps"]
    resources: ["deployments"]
    verbs: ["get", "patch"]
```

`patch` on the full Deployment is broader than `patch` on `deployments/scale` — RBAC review at deploy time should factor that in when the CronJob is configured for restart mode.

Extra-CA bundle merged into the trust store

The kube client uses the in-cluster CA from the ServiceAccount token secret automatically. For clusters whose API server certificate is signed by a corporate CA that is not in the SA-mounted `ca.crt`, `--extra-ca-bundle <path>` reads a PEM file (or a full chain of them) and appends

every `CERTIFICATE` block to `kube::Config.root_cert`. Non-certificate blocks in the file are filtered out.

Rendered in Helm as an optional ConfigMap mount; in Kustomize as an opt-in Component. Neither path forces the mount when there is no corporate CA to add.

The rustls 0.23 CryptoProvider gotcha

If you build against `kube-rs` with the `rustls-tls` feature on the current release, the binary compiles cleanly — and then panics at the first TLS handshake with:

```
Could not automatically determine the process-level CryptoProvider from
Rustls crate features
```

`rustls` 0.23 no longer picks a crypto backend based on downstream feature flags. The application itself must call `rustls::crypto::ring::default_provider().install_default()` before any TLS work happens. In k8s-scale-app-rs this runs as the first step in `main()`.

Two notes for anyone hitting the same wall: the panic never fires at compile time, and no test caught it locally because integration tests without a cluster do not open a TLS connection. It only surfaces when the CronJob actually tries to reach the API server.

Two deploy paths, independent

Helm and Kustomize both ship in the repo, and either can be used on its own:

- **Helm chart** with `values-{dev,preprod,prod}.yaml`, a `mode: scale | restart` toggle that swaps the subcommand argument and drops the unused replicas env, and `serviceAccount.create` / `rbac.create` / `extraCA.enabled` toggles for the pieces that a given cluster may or may not need.
- **Kustomize** with a base that has only the CronJob, plus three opt-in Components (`serviceaccount/`, `rbac/`, `extra-ca/`) that overlays include as needed. Overlays under `overlays/{dev,preprod,prod}/` set namespace and image tag and a patch that touches only the fields that vary per stage.

No Helm-inside-Kustomize glue, no Kustomize-on-Helm-render. Whichever tool matches an existing pipeline is the one to use.

Two container images

The default image and a `-debug` variant, both built in the same GitHub Actions matrix — the production/`-debug` split itself is a general pattern, covered in more depth in Part 2 of the K8s/OpenShift Best Practices series:

File	Base	Contents	Approx. size
`Containerfile`	`registry.access.redhat.com/ubi10/ubi-micro`	Binary + CA trust store + `/licenses/`, no package manager	≈35 MB
`Containerfile.debug`	`registry.access.redhat.com/ubi10/ubi-minimal`	Everything above + `bash`, `curl`, `bind-utils` (`dig`)	≈115 MB

`ubi10-micro` is not fully distroless in the Google sense — bash and coreutils are still there — but it has no package manager and no network tools, which fits the pod's actual operational surface for a scheduled scale or restart call. The `-debug` variant exists for the day a pod loops in `ImagePullBackOff` or a network policy silently drops the API-server traffic and someone needs to `kubectl exec` in with tools to check.

Why UBI10

The base itself, not just the two variants built from it, was a deliberate choice. I went with Red Hat's UBI over an Alpine or a generic Docker Hub base image for one reason: it gives the image a level of maintenance stability I don't have to police myself — CVE scanning and rebuilds are Red Hat's job, not something I track on my own. That comes with a tradeoff worth stating plainly: UBI is not "digitally sovereign" in the sense the upcoming Digital Sovereignty in Practice series covers — it is a US vendor's supply chain, the same as pulling from Docker Hub or almost any other public registry would be. I did not find a base that solved that concern without giving up the maintenance guarantees I wanted, so UBI10 here is a compromise I made deliberately, not a claim that the underlying problem is solved.

Both images ship an aggregated `/licenses/LICENSES.txt` bundled with `cargo-about` during the container build, generated from a whitelist of accepted SPDX identifiers. A new dependency that pulls in an unfamiliar license fails the container build on purpose.

Supply chain

The build-publish workflow runs three jobs on every push, plus a release job on tag pushes:

1. **`test`** — `cargo fmt --check`, `cargo clippy -D warnings`, `cargo test --release`. Cluster tests auto-skip in CI (no `KUBECONFIG`, no in-cluster SA token).
2. **`build-image`** — matrix over `Containerfile` and `Containerfile.debug`. Each variant is built with buildx, pushed to `ghcr.io/git001/k8s-scale-app-rs`, smoke-tested (`--version`, `scale --help`, `restart --help`), then:
   - signed with **cosign keyless** via the GitHub OIDC token (no key management — Fulcio issues a short-lived certificate, the signature is logged to Rekor);
   - has an **SPDX-JSON SBOM** generated with syft and attached via `cosign attest --type spdxjson`;
   - has **SLSA build provenance** pushed with `actions/attest-build-provenance`.
3. **`publish-chart`** — `helm lint` and `helm package` on every push; OCI push to `oci://ghcr.io/git001/charts/k8s-scale-app-rs`, plus cosign sign and SLSA provenance only on tag events (avoids overwrite conflicts on GHCR-immutable tags).
4. **`release`** (tag events only) — creates the GitHub Release with auto-generated notes from commits since the previous tag, attaches the packaged chart `.tgz` as an asset, and prepends the release body with pull and verify snippets.

All of that runs with `permissions: contents:read, packages:write, id-token:write, attestations:write` — the minimum needed for OIDC-based signing plus attestation upload.

Verifying a published image

A consumer that wants to check what came out of that pipeline needs cosign v2:

```
IMG=ghcr.io/git001/k8s-scale-app-rs:v1.0.0
IDENTITY='^https://github.com/git001/k8s-scale-app-rs/'
ISSUER=https://token.actions.githubusercontent.com

cosign verify \
  --certificate-identity-regexp "$IDENTITY" \
  --certificate-oidc-issuer "$ISSUER" \
  "$IMG"

cosign verify-attestation --type spdxjson \
  --certificate-identity-regexp "$IDENTITY" \
  --certificate-oidc-issuer "$ISSUER" \
  "$IMG"

cosign verify-attestation --type slsaprovenance \
  --certificate-identity-regexp "$IDENTITY" \
  --certificate-oidc-issuer "$ISSUER" \
  "$IMG"
```

For enforcement inside the cluster, a cosign-aware admission controller — Kyverno, Sigstore policy-controller, or Connaisseur — can turn the signature into a hard gate that rejects any pod trying to pull an image that was not produced by exactly this workflow. That is a separate deployment concern; the tool repo does not ship the policy itself.

Mirroring the image into another registry

Verifying a signature only works if the signature actually travels with the image. Not every `docker pull && docker push`-style mirroring tool preserves it — some tools do, some quietly drop it, and one (`oc-mirror v2`) solves a related but entirely different signature problem. I wrote up a full, sourced comparison — Which Tool Mirrors a Cosign-Signed Image into a Private Registry? — with `regctl` coming out as the most complete option for this specific image.

Code and repository

- Repository: k8s-scale-app-rs on GitHub
- Container images: `ghcr.io/git001/k8s-scale-app-rs:v1.0.0` and `ghcr.io/git001/k8s-scale-app-rs:v1.0.0-debug`
- Helm chart: `oci://ghcr.io/git001/charts/k8s-scale-app-rs:1.0.0`

Feedback and PRs welcome.