KUBERIK · Kubernetes-native Continuous Delivery

Safe, Hands-Off Deployments for Kubernetes

Declarative multi-stage progressive delivery from commit to production. Batteries included.

Luka Rumora

CNC Switzerland Meetup · 21 May 2026 · Zürich

github.com/kuberik/kuberik

Hi, I'm Luka.

Almost ten years as an SRE. Building and breaking CD pipelines, running Kubernetes platforms.

Across Jenkins, GitLab CI, GitHub Actions, Flux. The same problems show up every time.

Today I work on **Kuberik**, a Kubernetes-native delivery orchestrator.

github.com/kuberik/kuberik

Agenda

What we'll cover today

01 **Why pipelines are the wrong shape**

One dimension. Production needs many.

02 **Kuberik's mental model**

Composable primitives that own the loop.

03 **The resources, with examples**

Rollout, gates, health checks, canary tests, environments, OCI artifacts.

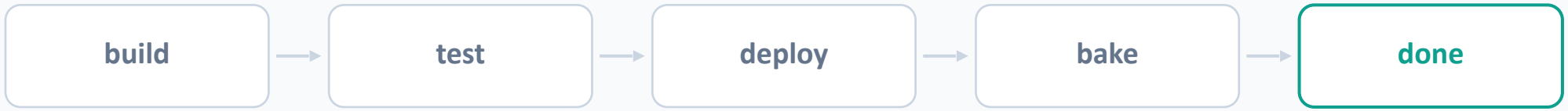
04 **Operating in production**

Pinning, force-deploy, local rollback.

05 **Demo and questions**

See it run, then your turn.

Pipeline pain you've felt.



A single line. Goes forward once, when everything works.

01

You build every box.

Canary, health checks, smoke tests, rollback, promotions. Each team writes their own, in their own pipeline.

No shared primitives. Each pipeline is a one-off.

02

The line only goes forward.

Pin a version. Roll back. Skip the gates. Hotfix one environment. None of these are on the line.

Production isn't the happy path. Operations go sideways.

03

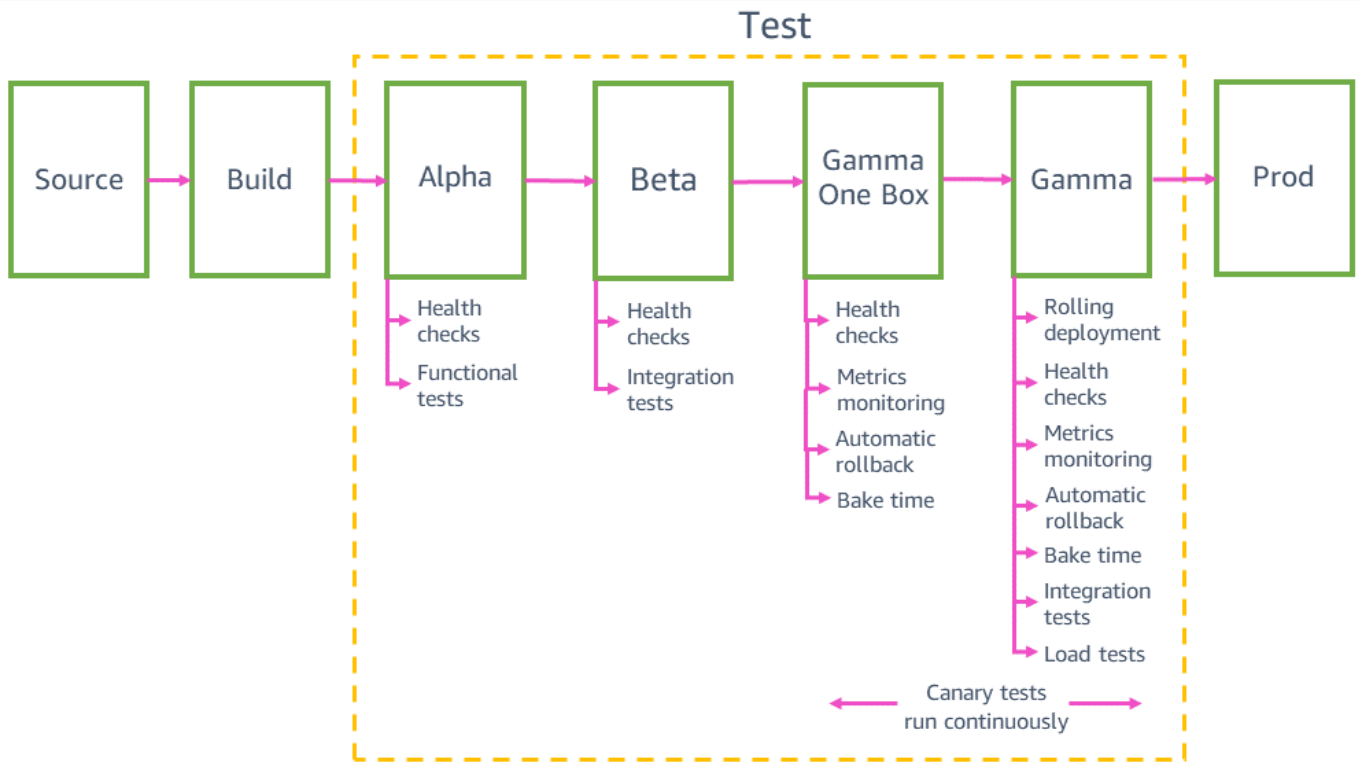
One line. No environments.

"Healthy in staging for a day, then allowed in prod" isn't in Argo Rollouts, Flagger, or any GitOps tool.

No second dimension. The orchestration layer doesn't exist.

An opinionated pipeline. Hands-off.

*From the **AWS Builders' Library**: "Automating safe, hands-off deployments."*



*Most teams build a fraction of this. **Kuberik gives you all of it. Out of the box.***

Introducing

Kuberik

Everything a pipeline makes you script, Kuberik makes declarative.

imperative scripts → **declarative resources**

Canary, health checks, promotion, rollback: things you declare, not code.

only the happy path → **the exceptional cases too**

Rollback, incident pins, force-deploy, deployment windows are first-class.

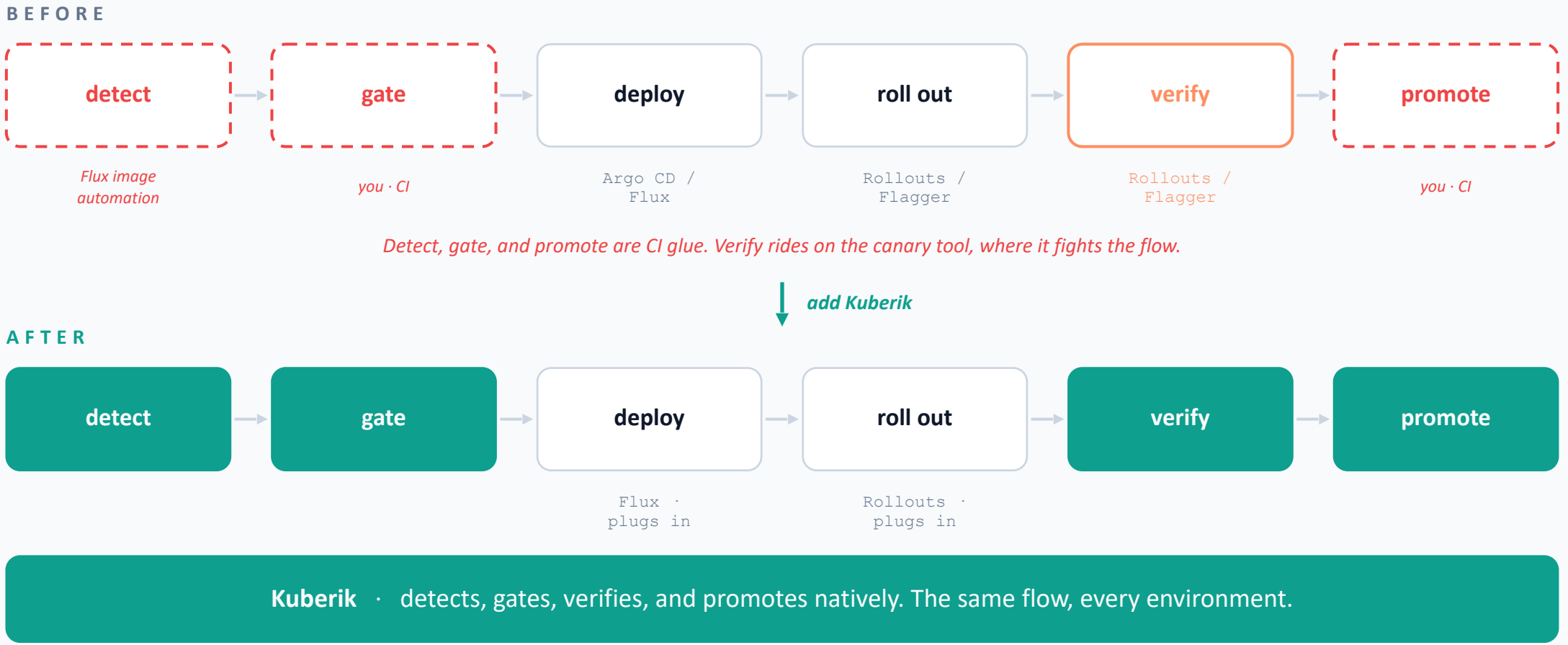
a scripting DSL → **native components that plug in**

You never wire a stage. Kuberik knows where each piece goes and when to fire it.

In-cluster. Reactive. Composable.

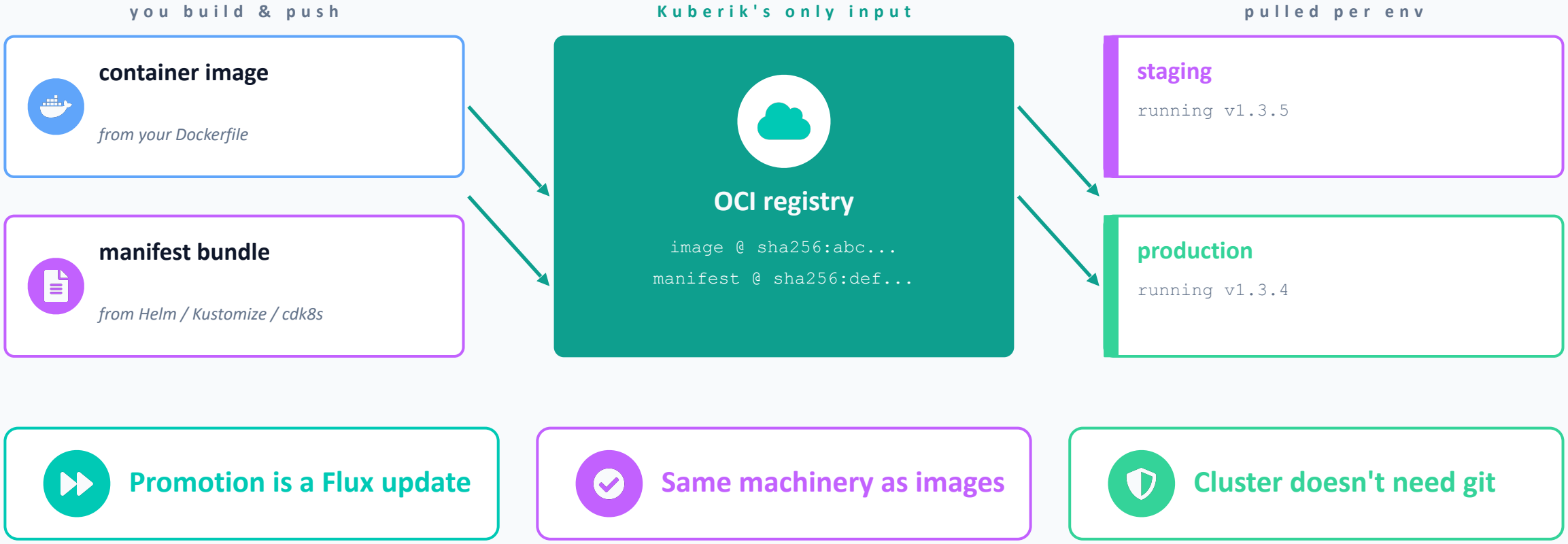
Where Kuberik fits.

The same end-to-end flow. Before: you glue the orchestration together in CI. After: Kuberik owns it.



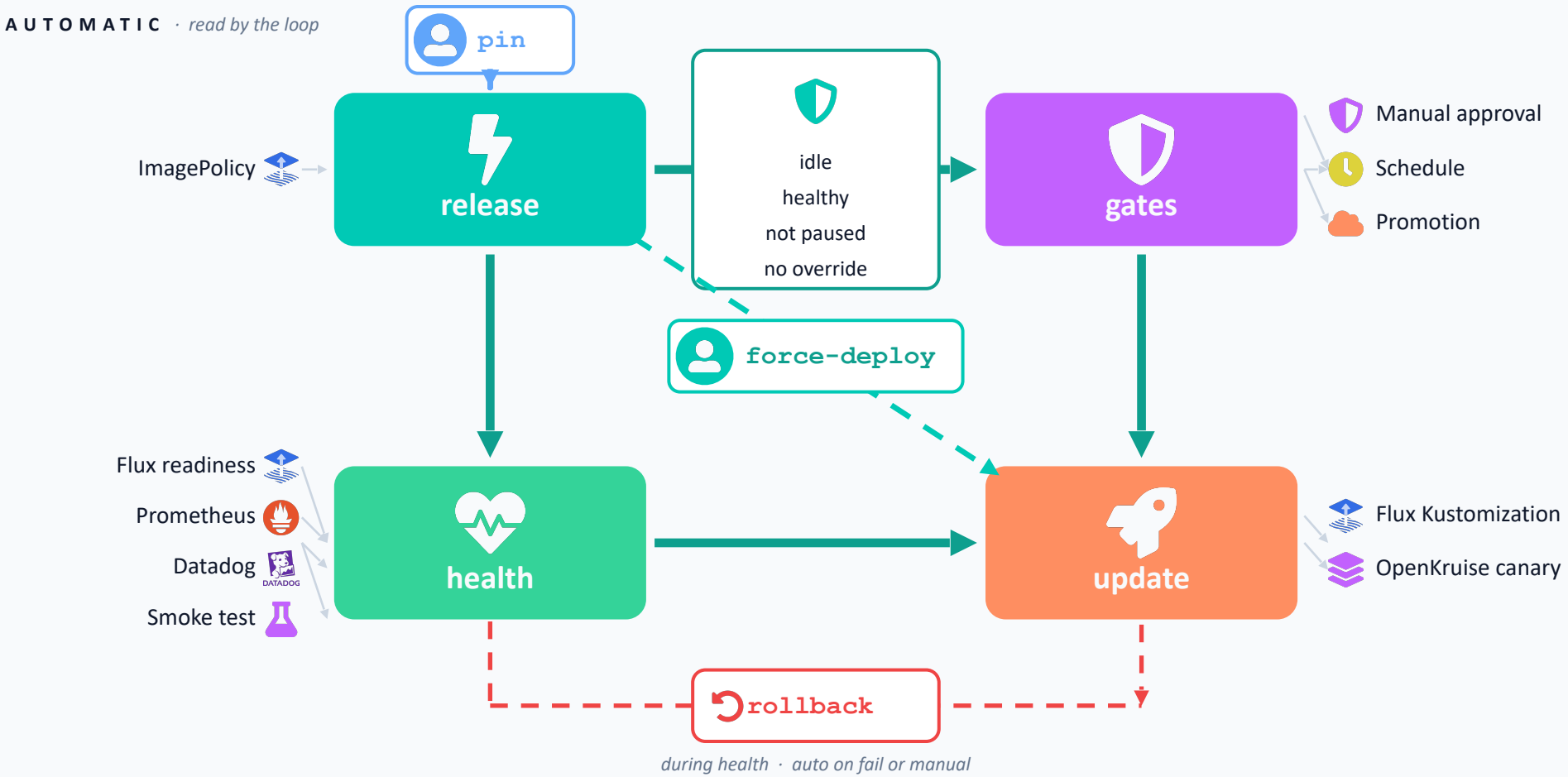
You release. Kuberik delivers.

You build, tag, and push to your OCI registry. That is the entire handoff. Same channel for images and manifests.



Kuberik doesn't build images, decide when to release, or replace your GitOps tool. It picks up at the registry.

Components watch. Users intervene. The loop reacts.



Everything you need to set up automated deploys.

Three Flux resources, one Kuberik resource, and a $VERSION placeholder in your deployment.

01 · image-automation.yaml (Flux)

```yaml
apiVersion: image.toolkit.fluxcd.io/v1beta2
kind: ImageRepository
spec:
  image: ghcr.io/me/my-app
---
kind: ImagePolicy
spec:
  policy: { semver: { range: ">=0.1.0" } }
```

02 · rollout.yaml (Kuberik)

```yaml
apiVersion: kuberik.com/v1alpha1
kind: Rollout
metadata: { name: my-app }
spec:
  releasesImagePolicy: { name: my-app }
  bakeTime: 5m
```

03 · kustomization.yaml (Flux)

```yaml
apiVersion: kustomize.toolkit.fluxcd.io/v1
kind: Kustomization
metadata:
  annotations:
    rollout.kuberik.com/substitute.VERSION.from: my-app
spec:
  path: ./deployments/prod
# Kuberik injects spec.postBuild.substitute.VERSION
```

04 · deployment.yaml (yours, with one placeholder)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata: { name: my-app }
spec:
  template:
    spec:
      containers:
      - name: my-app
        image: ghcr.io/me/my-app:${VERSION}
```

Healthchecks.

kustomization watches Flux resources. prometheus-alert watches firing alerts. The class system is open.

kustomization-health.yaml

```yaml
apiVersion: kuberik.com/v1alpha1
kind: HealthCheck
metadata:
  annotations:
    healthcheck.kuberik.com/kustomization: my-app
spec:
  class: kustomization
```

prometheus-health.yaml

```yaml
apiVersion: kuberik.com/v1alpha1
kind: HealthCheck
metadata:
  annotations:
    healthcheck.kuberik.com/prometheus-url: \
      http://prometheus.monitoring:9090
    healthcheck.kuberik.com/prometheus-alert-labels: \
      app=my-app,severity=critical
spec:
  class: prometheus-alert
```

Watches a Flux Kustomization.

Pass: every resource Ready (kstatus).
Fail: any resource stuck. Bake fails.

Watches Prometheus alerts.

Pass: no matching alert firing.
Fail: any matching alert. Rollback fires.

Without Kuberik: *a post-deploy bash check per service, querying Prometheus by hand.*

A canary controller that doesn't own which version runs.

OpenKruise enhances the native Deployment in place. Rollback stays with Kuberik.

STEP DEFINITION — third-party

Kruise Rollout

Defines canary steps.

extends →

EXTENDED BY KUBERIK — Kuberik

Bake time + RolloutTest

Adds bake time and per-step tests.

feeds status →

OBSERVED BY — core

Kustomization HealthCheck

Watches both. Fails update if either does.

openkruise-rollout.yaml

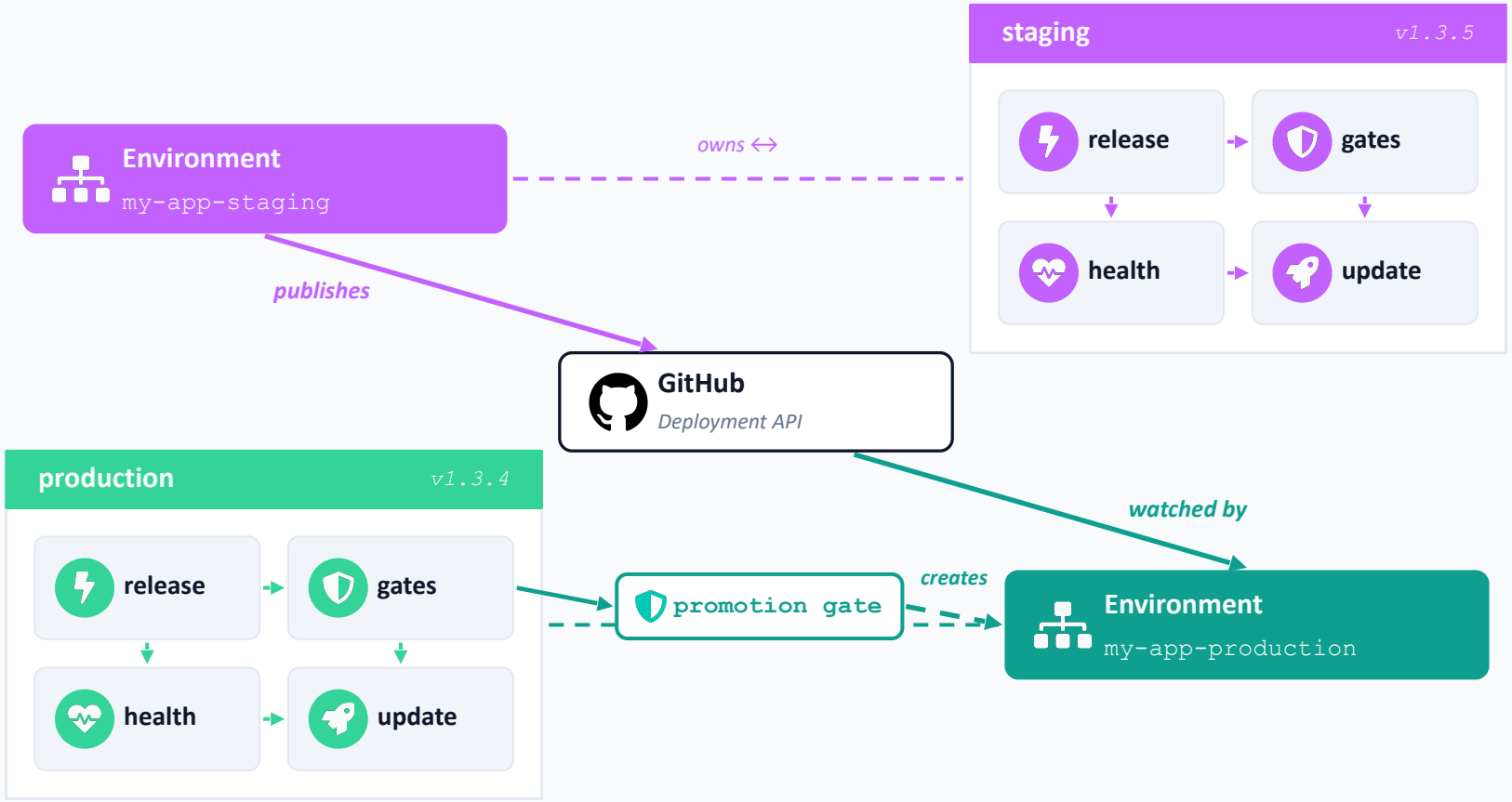
```yaml
apiVersion: rollouts.kruise.io/v1beta1
kind: Rollout
metadata:
  name: my-app
  annotations:
    rollout.kuberik.io/step-1-bake-time: "5m"
spec:
  workloadRef: { kind: Deployment, name: my-app }
  strategy:
    canary:
      steps:
        - { replicas: 1 }
        - { traffic: 25% }
        - { traffic: 50% }
        - { traffic: 100% }
```

smoke-test.yaml

```yaml
apiVersion: rollout.kuberik.com/v1alpha1
kind: RolloutTest
metadata: { name: smoke-test }
spec:
  rolloutName: my-app
  stepIndex: 1
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: smoke
            image: curlimages/curl
            command: ["curl", "--fail",
              "http://my-app-canary/health"]
```

A gate that watches another environment.

Each Rollout owns an Environment. Environments mirror state to the GitHub Deployment API and create gates on their own Rollout.



production-env.yaml

```yaml
apiVersion:
environments.kuberik.com/v1alpha1
kind: Environment
metadata:
  name: my-app-production
spec:
  rolloutRef:
    name: my-app
  name: production
  backend:
    type: github
    project: org/repo
  relationship:
    environment: staging
    type: After
```

Overrides: pin a version, or skip the gates

Real operations need an off-ramp. Kuberik gives you two: pinning, and force-deploy.

PINNING

Stick to a known-good

pinned-rollout.yaml

```yaml
kind: Rollout
metadata:
  name: my-app
spec:
  releasesImagePolicy:
    name: my-app
  wantedVersion: "v1.2.3"
```

The Rollout sticks to wantedVersion. Commit it to Git, freeze a release.

FORCE-DEPLOY

Skip every gate, ship now

shell

```
$ kubectl annotate rollout my-app \
    rollout.kuberik.com/force-deploy=v1.2.3

# version must exist in available releases
# annotation is cleared after deployment
```

For hot-fixes, rollbacks, or just "I want this version live now."

Without Kuberik: *branch hacks, --skip-tests flags, hand-rolled rollback.sh nobody touched in eight months.*

Each environment decides its own intake.

Upstream can't ram a version into a downstream environment. Each one pulls when its own guard clears. Picture a production incident:

PIPELINE-DRIVEN CD

Upstream pushes downstream

dev	shipped v1.5
triggers	
staging	shipped v1.5
triggers	
production	incident · rammed

The trigger fires regardless of prod's state.

KUBERIK

Each environment pulls

dev	v1.5 baking
staging	v1.5 ready
production	incident · guard holds

Prod's guard holds. Staging finishing changes nothing.

What you get out of the box.

Things that are bespoke pipeline code in every other tool. Resources you declare in Kuberik.

01

Resources, not pipeline scripts

Canary, health checks, rollback, promotions are CRDs. Not bash. Not YAML steps. The controller does the work.

02

Each environment is autonomous

Roll back production. Staging keeps shipping. No global pipeline freeze when one environment goes bad.

03

Operational patterns are built in

Pin, force-deploy, sequential version processing, deploy windows. First-class. Not bolted on by your team.

04

Stays out of your git repo

No image-tag write-back. The cluster shouldn't even have read access to the git repo, let alone write. Manifests ship as OCI artifacts.

LIVE

Demo

Watch a release flow through the loop:

- push a new image
- schedule gate evaluates
- bake plus Prometheus checks
- promote to next environment

Thank you.

Questions, gripes, war stories. Bring them.

`github.com/kuberik/kuberik`

★ *If you liked what you saw, please star it.*