AIVAR INNOVATIONS

kro

Kube Resource Orchestrator

Pronounced "crow" · kubernetes-sigs · v0.9.2 · CNCF SIG Cloud Provider

Define custom Kubernetes APIs without writing a single controller — just YAML, CEL expressions, and a dependency graph kro infers for you.

Manan Jain · Pod Weekly

EKS Live Demo

June 2026

Building for Kubernetes, Not with It

The Operator pattern is powerful — but creating custom platform APIs today requires a painful amount of infrastructure work.

Every CRD Needs a Controller

Deep Go expertise, custom reconciliation loops, RBAC, leader election, and extensive testing — for every custom resource.

Fleet of Bespoke Controllers

Large orgs end up managing dozens of controllers — each needing monitoring, upgrades, versioning, and a dedicated team.

Platform, Not Product

Teams spend all capacity on controller infra, leaving no room for business logic. "We build for K8s, not with it."

"We often feel like we are building for Kubernetes rather than with Kubernetes." — AWS customer research, 2024

kro Turns Your YAML Into

A CRD + Micro-Controller

At runtime. No Go required. Define a ResourceGraphDefinition (RGD) — kro validates it, generates the CRD, and deploys a dedicated controller for your new API.

Originally:	AWS Labs (Nov 2024)
Now:	kubernetes-sigs · CNCF
Latest:	v0.9.2 · Experimental

```
# Install kro:
$ helm install kro \
  oci://registry.k8s.io/kro/charts/kro \
  --namespace kro-system \
  --create-namespace

# One CRD appears:
$ kubectl get crds | grep kro
resourcegraphdefinitions.kro.run
```

```
# kro creates for each RGD:
> New CRD (webapps.kro.run)
> Dedicated micro-controller
> Full reconciliation loop
> Status propagation
> Cascade delete on instance
  removal
```

The ResourceGraphDefinition

PART 1 — SCHEMA (Your API surface)

```
spec:
  schema:
    kind: WebApp  # new K8s resource
    apiVersion: v1alpha1
    spec:
      name: string
      image: string | default="nginx"
      replicas: integer | default=2
      hpa.enabled: boolean | default=false
    status:
      # CEL: bubble up from children
      availableReplicas:
${deployment.status.availableReplicas}
```

SimpleSchema — types, defaults, constraints, and validation in one readable line. No OpenAPI boilerplate.

PART 2 — RESOURCES (What to create)

```
resources:
  # DAG inferred from ${...} references
  - id: deployment
    template:
      kind: Deployment
      spec.replicas: ${schema.spec.replicas}

  - id: service
    template:
      kind: Service
      # depends on deployment (auto!):
      spec.selector:
${deployment.spec.selector.matchLabels}
```

CEL expressions ${…} create implicit dependencies — no explicit dependsOn: field needed.

The Glue: Common Expression Language

No Side Effects

Can't modify state, write files, or make network calls. Safe to execute from user input.

Always Terminates

No loops or recursion. Evaluates in nanoseconds — ideal for tight reconciliation loops.

Type-Checked at Apply

kro fetches OpenAPI schemas, validates every field path and type. Errors caught before any instance runs.

Dependency Signal

Cross-resource references encode dependencies. The DAG is implicit in your CEL expressions.

```
# Standalone: ${schema.spec.replicas}   |  String template: "${schema.spec.name}-svc"   |  Conditional:
${schema.spec.env == "prod" ? "nginx:stable" : "nginx:latest"}   |  includeWhen: - ${schema.spec.hpa.enabled}
```

Auto-Inferred Dependency Graph

kro parses every ${...} expression and builds a Directed Acyclic Graph (DAG). You never declare order — it's inferred from relationships.

Linear Chain

Parallel / Diamond

Conditional Subgraph

At Runtime:

↑ Creation: topological order

↓ Deletion: reverse order — no orphans

⏸ Blocking: waits for CEL field to be resolvable (not just resource-exists)

⊗ Cycles: caught at apply time, clear error

```
$ kubectl get rgd webapp-rgd -o
jsonpath='{.status.topologicalOrder}'
["deployment","service","hpa","ingress"]
```

WebApp ResourceGraphDefinition

PLATFORM TEAM DEFINES ONCE:

```
kind: ResourceGraphDefinition
spec.schema.kind: WebApp   # new API
spec.schema.spec:
  name: string
  image: string | default="nginx:alpine"
  replicas: integer | default=2
  hpa.enabled: boolean | default=false
  ingress.enabled: boolean | default=false
```

DEVELOPER WRITES (8 lines!):

```
apiVersion: kro.run/v1alpha1
kind: WebApp
metadata:
  name: my-webapp
spec:
  name: my-webapp
  image: nginx:alpine
  hpa.enabled: true
```

DEMO COMMANDS:

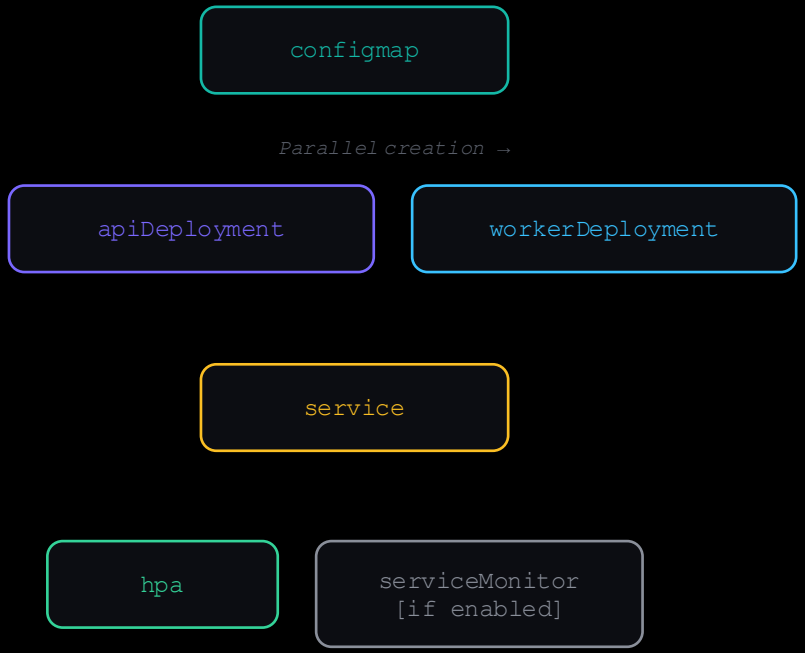
```
# 1. Apply RGD (done before call)
$ kubectl apply -f 01-simple-rgd.yaml

# 2. Inspect topology
$ kubectl get rgd webapp-rgd -owide
> STATE: Active
> ORDER: deployment,service,hpa

# 3. Create instance
$ kubectl apply -f 02-simple-instance.yaml

# 4. Watch resources appear
$ kubectl get deploy,svc,hpa -w

# 5. Cascade delete
$ kubectl delete webapp my-webapp
```

KRO CREATES AUTOMATICALLY:

- ✓ Deployment · Service · HPA (if enabled)
- ✓ Ingress with ALB annotations (if enabled)
- ✓ Status bubbled up to WebApp.status
- ✓ Full cascade delete on instance removal

MicroserviceApp — Multi-Resource Stack

DAG INFERRED BY KRO:



KEY CONCEPTS DEMONSTRATED:

Parallel Branches

apiDeployment and workerDeployment both reference only ${configmap.metadata.name} — kro creates them simultaneously.

Diamond Convergence

Service references ${apiDeployment.spec.selector.matchLabels} — waits for both branches before creating.

Config Injection via CEL

Worker logs show ENVIRONMENT=staging from ConfigMap — CEL-wired from schema through configmap to container env.

Conditional Subgraph

Set monitoring.enabled: true → ServiceMonitor and its subtree materialize automatically.

kro + ACK — Cloud + K8s in One Graph

The Idea

ACK exposes AWS resources (S3, RDS, DynamoDB) as K8s CRDs. kro treats them like any other resource in the graph — CEL wires cloud status fields directly into native K8s resources.

AUTO-INFERRED DAG (bucket → deployment → service):

Key Insight

No glue code — the CEL expression creates the dependency edge. Cloud provisioning and app deployment are a single reconciliation loop.

RGD SNIPPET — 05c-ack-rgd.yaml:

```
resources:
  - id: bucket
    readyWhen:
      - ${bucket.status.ackResourceMetadata.arn != ""}
    template:
      apiVersion: s3.services.k8s.aws/v1alpha1
      kind: Bucket
      spec.name: "${schema.spec.name}-${schema.spec.bucketSuffix}"

  - id: deployment
    template:
      ...env:
        - name: S3_BUCKET_ARN
          value: "${bucket.status.ackResourceMetadata.arn}"
                   ^— cloud status → app env
```

DEMO COMMANDS:

```
# Install ACK S3 controller
$ bash demo/05b-ack-setup.sh

# Apply the RGD (new kind: WebAppWithS3)
$ kubectl apply -f demo/05c-ack-rgd.yaml

# Deploy — kro orchestrates cloud + K8s
$ kubectl apply -f demo/05d-ack-instance.yaml

# Verify ARN injected into the pod
$ kubectl exec deploy/s3-demo-app-deploy \
    -- env | grep S3_BUCKET_ARN
S3_BUCKET_ARN=arn:aws:s3:::s3-demo-app-kro-demo-2024
```

How Does kro Compare?

Capability	Helm	Crossplane	Custom Operator	kro
Abstraction unit	Chart templates	Compositions	CRD + controller	ResourceGraphDefinition
Runtime reconciliation	✗ (static)	✓	✓	✓
Dependency ordering	Manual hooks	Readiness checks	Custom code	Auto-inferred DAG
Status propagation	✗	Partial	✓	✓ CEL-defined
Conditional resources	if/else templates	Patches	✓	includeWhen CEL
Go expertise required	✗	✗	✓ required	✗ YAML + CEL only
Production ready	✓	✓	✓	Experimental

kro fills the gap between "Helm for packaging" and "full Operator with custom controllers." Combines well with ACK (AWS Controllers for K8s) to orchestrate cloud + native resources in one RGD.

Experimental Today, Production Tomorrow

Nov 2024 — **AWS Labs launch**

Open-sourced as awslabs/kro. Initial ResourceGroup CRD + blog post.

2025 — **Donated to kubernetes-sigs**

Now under CNCF SIG Cloud Provider. API renamed: ResourceGraphDefinition.

v0.9.2 — **Latest release**

CEL libraries, forEach collections, external refs, graph revisions, controller metrics.

Future — **On the roadmap**

RGD versioning · schema migration · enhanced multi-tenancy · production graduation.

WHEN TO USE KRO TODAY:

✓ **Internal Developer Platform prototyping**

Build standardised app blueprints quickly without controller code.

✓ **Platform API experimentation**

Quickly iterate on K8s abstractions without controller boilerplate.

✗ **Production workloads (not yet)**

APIs are not solidified. Not integrated into EKS managed add-ons. Evaluate carefully.

Questions?

Q1 — Can kro manage AWS resources (S3, RDS)?

Yes — combine with ACK (AWS Controllers for K8s). An RGD can include both native K8s and ACK CRDs; kro wires CEL across them.

Q2 — How does kro handle RGD schema updates?

Graph Revisions (in progress). Existing instances continue on old version; schema migration is a planned first-class feature.

Q3 — Does it work with ArgoCD / GitOps?

Yes — RGDs are just K8s resources. Apply via ArgoCD. App teams commit instances in their own repos — works like any CRD.

Q4 — What happens when a child resource fails?

kro micro-controller retries with exponential backoff. Instance status reflects the failure condition from the child resource.