KubeCon CloudNativeCon
North America 2026

Health Check Hell

A call to standardize CR health across the ecosystem



We experience it from different perspectives



Reggie from the maintainer point of view



Dag from the user point of view



Regina (Reggie) Voloshin

Argo CD maintainer at Octopus Deploy

CNCF Ambassador

Octopus Deploy

Continuous Delivery Platform

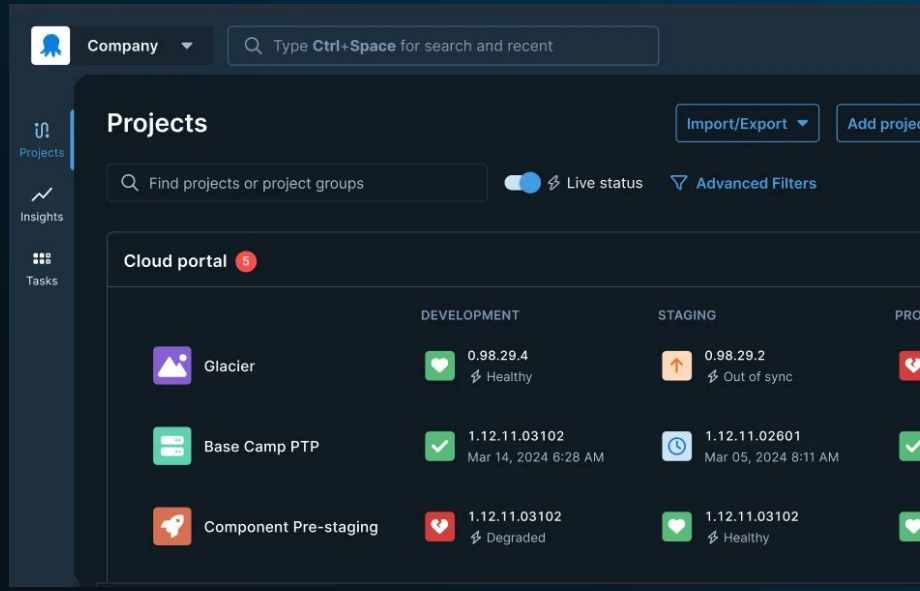
AI is accelerating Development.

Can your CD keep up?

CD for all your apps
Powerful pipelines and dashboards to track who deployed what, when and where

Agentic-ready governance
Powerful rules and governance, ready for your org or agentic agents

Kubernetes-native
K8s Agent and Native integration for Argo CD



4,000+
Organizations Worldwide

91%
Report improved deployment confidence

ARGO CD MAINTAINER

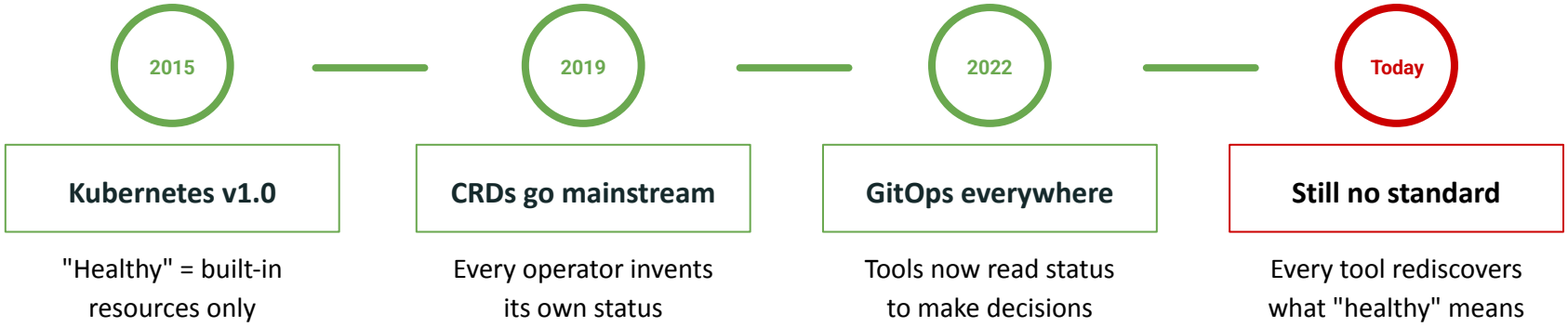
Users Love Us 4.6

octopus.com/free

The question that never went away

Is this resource healthy? 💚💔

Asked in 2015. Asked in 2020. Still asked today.



How do you know something is healthy?



Is a patient healthy?

Universal vitals exist to base the answer on: temperature, pulse, blood pressure, respiration.

Same metrics. Every patient. You know the metrics even if you see the patient for the first time.



Is a K8s CR healthy?

No universal vitals exist to answer this: the vitals and the answer depend on knowing *this particular resource type* intimately.

Different metrics for different CRs. You only know the metrics of a CR you have encountered in the past.

Medicine standardized what "healthy" means. Kubernetes never did — for CRs.

Resource health means different things to different people

It also means different things about different K8s resources

Health is highly subject to interpretation

Spec and Status

spec

- A hard contract
- Strongly typed, validated by the API server
- Built to be read by machines and humans

status

- Optional, loosely structured
- Treated as a lesser citizen of the K8s API
- No enforced shape — conditions, phases, both, neither
- Feels like it was built to be read by humans in a YAML dump

CRs: pick your own adventure


```yaml
apiVersion: cert-manager.io/v1alpha2
kind: Certificate
...
status:
  conditions:
  - lastTransitionTime: "2019-02-15T18:26:37Z"
    message: |
      Resource validation failed: spec.acme.config:
      Required value: no ACME solver configuration
      specified for domain "cd.apps.argoproj.io"
    reason: ConfigError
    status: "False"
    type: Ready
```

Certificate

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
...
status:
  parents:
    - conditions:
        - lastTransitionTime: "2025-10-14T11:19:41Z"
          message: No listeners match this parent ref
          observedGeneration: 1
          reason: NoMatchingParent
          status: "False"
          type: Accepted
        - ...
      controllerName: gateway.envoyproxy.io/gateway...
      parentRef:
        group: gateway.networking.k8s.io
        kind: Gateway
        name: eg
        namespace: envoy-gateway-system
        sectionName: foo-nonexistent
```

HTTPRoute

```yaml
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
...
status:
  availableReplicas: 2
  conditions:
  - lastTransitionTime: "2022-09-01T19:54:00Z"
    status: "True"
    type: Available
  - lastTransitionTime: "2022-09-02T09:57:03Z"
    status: "True"
    type: Reconciled
  paused: false
  replicas: 2
  shardStatuses:
  - availableReplicas: 2
    replicas: 2
    shardID: "0"
    unavailableReplicas: 0
    updatedReplicas: 2
  unavailableReplicas: 0
  updatedReplicas: 2
```

Prometheus

CR health check popular consumers















Main health consumer: Argo CD



KubeCon CloudNativeCon North America 2026

Hundreds

of contributed Lua health checks

At least 2-3 files per resource kind. Growing every release.

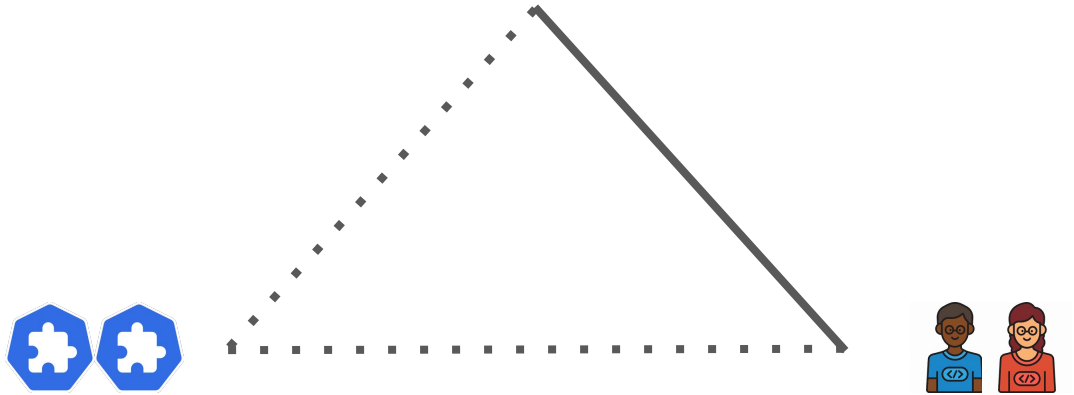
Argo CD became the main machine consumer analyzing resource status.

So the ecosystem's unsolved problem — what does "healthy" mean for a CR — became Argo CD's problem to carry, file by file.

The wrong triangle



Argo CD Maintainers
Maintain the engine, not the CR knowledge



Project Maintainers
Hold the CR domain expertise

Argo CD users (Project users)
Contribute most of the health checks

Why is this triangle wrong?



Users - the main contributors of health checks:

They only see a partial, mostly undocumented slice of the CR's real behavior.

Contributing a health check requires knowing internals most users never touch.

… and just imagine some of the contributed AI-generated health checks. 🤦

Project / CR maintainers:

They know exactly what "healthy" means for their resource.

But currently they don't own the health check.

How are health checks contributed to Argo CD today?

- Argo CD's health check statuses: **Healthy, Suspended, Progressing, Missing, Degraded, Unknown**
- Lua scripts map: CR status field => Argo CD statuses
- **Real CRs** from the contributor's cluster are required



```lua
health_status = {
  status = "Progressing",
  message = "Provisioning ..."
}

for i, condition in ipairs(obj.status.conditions) do
  if condition.type == "Synced" then
    if condition.status == "False" then
      health_status.status = "Degraded"
      health_status.message = condition.message
      return health_status
    end
  end

  if condition.type == "RolloutCompleted" then
    if condition.status == "False" then
      health_status.status = "Degraded"
      health_status.message = condition.message
      return health_status
    end
  end

  if condition.type == "Ready" then
    if condition.status == "True" then
      health_status.status = "Healthy"
      health_status.message = "Resource is up-to-date."
      return health_status
    end
  end
end
```

Challenges with contributed health checks - 1

Complex status structure

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
...
status:
  parents:
    - conditions:
        - lastTransitionTime: "2025-10-14T11:19:41Z"
          ...
          reason: NoMatchingParent
          status: "False"
          type: Accepted
        - lastTransitionTime: "2025-10-14T11:19:41Z"
          ...
          reason: ResolvedRefs
          status: "True"
          type: ResolvedRefs
      controllerName: gateway.envoyproxy.io/gatewayclass-controller
      parentRef:
        group: gateway.networking.k8s.io
        kind: Gateway
        name: eg
        namespace: envoy-gateway-system
        sectionName: foo-nonexistent
```

100 lines health check

```lua
local hs = {}

function checkConditions(conditions, conditionType)
  for _, condition in ipairs(conditions) do
    if condition.type == conditionType and condition.status == "False" then
      return false, condition.message or ("Failed condition: " .. conditionType)
    end
  end
  return true
end

-- isParentGenerationObserved checks if a parent's conditions match the current resource generation
-- For HTTPRoute, observedGeneration is stored in each condition within a parent
function isParentGenerationObserved(obj, parent)
  if obj.metadata.generation == nil then
    -- If no generation is set, accept all conditions
    return true
  end

  if parent.conditions == nil or #parent.conditions == 0 then
    return false
  end

  -- Check if all conditions have observedGeneration matching current generation
  for _, condition in ipairs(parent.conditions) do
    if condition.observedGeneration ~= nil then
      if condition.observedGeneration ~= obj.metadata.generation then
        return false
      end
    end
  end

  return true
end

if obj.status ~= nil then
```

Challenges with contributed health checks - 1

Complex status structure ealth check

A complex status structure
requires a long health script
- which still covers just a subset of possible
health statuses!

```
apiVersion: gateway.
kind: HTTPRou
...
status:
  parents:
    - condi
        - la
          ...
          rea
          stat
          type:
        - lastT
          ...
          reason:
          status:
          type: ResolvedRefs
      controllerName: gateway.envoyproxy.io/gatewayclass-controller
      parentRef:
        group: gateway.networking.k8s.io
        kind: Gateway
        name: eg
        namespace: envoy-gateway-system
        sectionName: foo-nonexistent
```

```
tatus == "False" then
ition: " .. conditionType)

conditions match the current resource generation
in each condition within a parent
obj, parent)
on == nil then
ration is set, accept all conditions
turn true
end

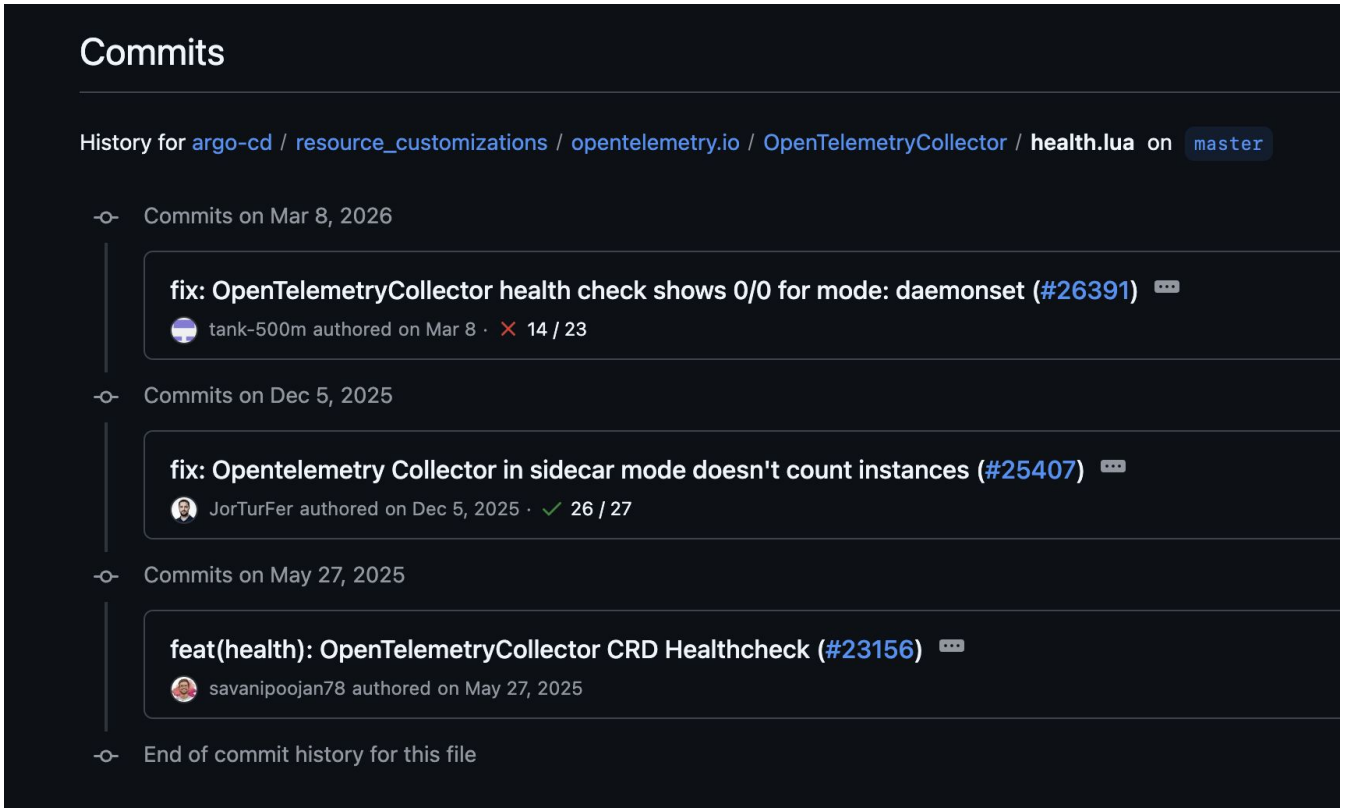
if parent.conditions == nil or #parent.conditions == 0 then
  return false
end

-- Check if all conditions have observedGeneration matching current generation
for _, condition in ipairs(parent.conditions) do
  if condition.observedGeneration ~= nil then
    if condition.observedGeneration ~= obj.metadata.generation then
      return false
    end
  end
end

return true
end

if obj.status ~= nil then
```

Challenges with contributed health checks - 1



Challenges with contributed health checks - 1



Further health contributions for the same CR are often required

Challenges with contributed health checks - 2


```
+ apiVersion: postgresql.cnpg.io/v1
+ kind: Backup
+ metadata:
+   name: cluster-backup-20260627
+   namespace: db
+ spec:
+   cluster:
+     name: cluster-example
+   method: barmanObjectStore
+ status:
+   phase: running
+   startedAt: "2026-06-27T01:00:00Z"
```

```
+ apiVersion: longhorn.io/v1beta2
+ kind: Volume
+ metadata:
+   name: pvc-abc123
+   namespace: longhorn-system
+ spec:
+   numberOfReplicas: 3
+   size: "10737418240"
+ status:
+   state: attached
+   robustness: degraded
+   currentNodeID: node-1
```

Challenges with contributed health checks - 1


```
+ apiVersion: postgresql.cnpg.io/v1
+ kind: Backup
+ metadata:
+   name: cluster-backup-20260627
+   namespace: db
+ spec:
+   cluster:
+     name: cluster-example
+   method: barmanObjectStore
+ status:
+   phase: running
+   startedAt: "2026-06-27T01:00:00Z"
```

```
+ apiVersion: longhorn.io/v1beta2
+ kind: Volume
+ metadata:
+   name: pvc-abc123
+   namespace: longhorn-system
+ spec:
+   numberOfReplicas: 3
+   size: "10737418240"
+ status:
+   state: attached
+   robustness: degraded
+   currentNodeID: node-1
```

Challenges with contributed healt - 1

Were those manifests really exported from the cluster or mocked up by AI?

```
+ apiVersion: ...
+ kind: Back...
+ metadata:
+   name: clu...
+   namespace:...
+ spec:
+   cluster:
+     name: clust...
+   method: barmanObjectStore
+ status:
+   phase: running
+   startedAt: "2026-06-27T01:00:00Z"
```

```
...horn.io/v1beta2
...c123
...amespace: longhorn-system
+ spec:
+   numberOfReplicas: 3
+   size: "10737418240"
+ status:
+   state: attached
+   robustness: degraded
+   currentNodeID: node-1
```

What if the CNCF projects maintainer submitted a health check PR ?




YAY!

Argo CD “standardized” health structure in a way …

… but internally, for its own consumption

Can this standard be used globally across the ecosystem?

KubeCon CloudNativeCon
North America 2026



Dag Bjerre Andersen

Senior Platform Engineer at EGMONT

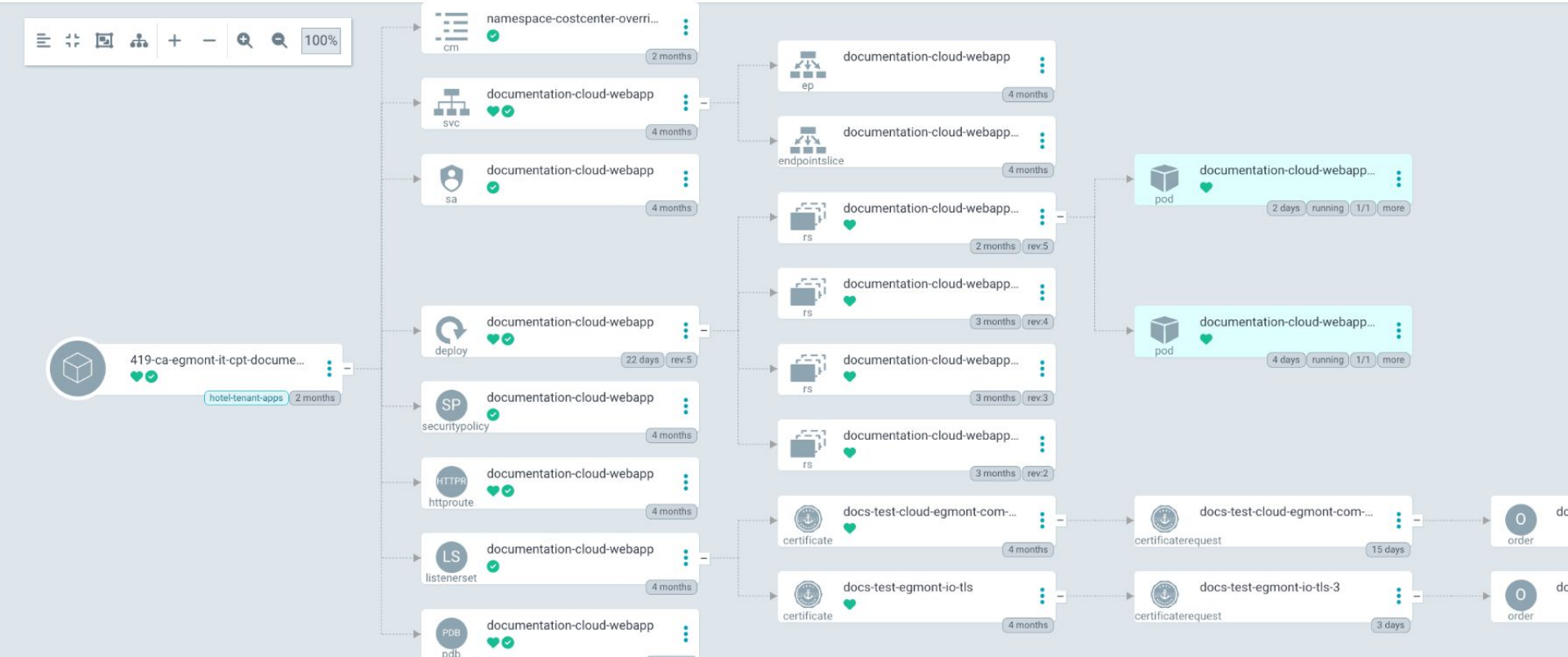
"Regular CNCF project user"

DETAILS | DIFF | SYNC | SYNC STATUS | HISTORY AND ROLLBACK | DELETE | REFRESH

APP HEALTH
Healthy

SYNC STATUS
Synced to HEAD (9feb359)
Auto sync is enabled.
Author: GitHub Actions <actions@github.com> -
Comment: Updated service-cloudtags/cloudtags/backend.yaml in eks-hotel-a-

LAST SYNC
Sync OK to a84570f
Succeeded 22 days ago (Fri Sep 25 2026 14:29:08 GMT+0200)
Author: GitHub Actions <actions@github.com> -
Comment: Updated itshop/itshop-frontend/frontend.yaml in eks-hotel-a-nonp



Why is Argo CD great for developers?



Graph Visibility

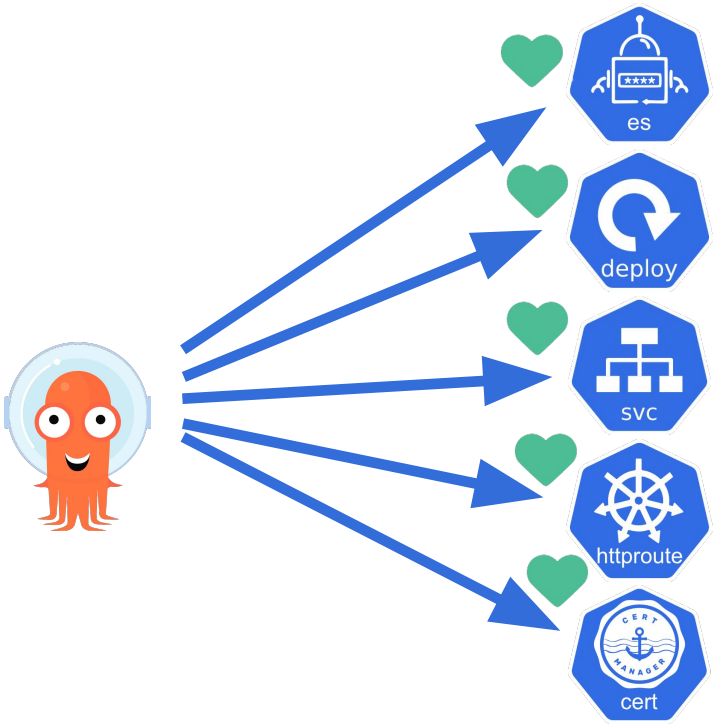


Resource Health



one-stop-shop

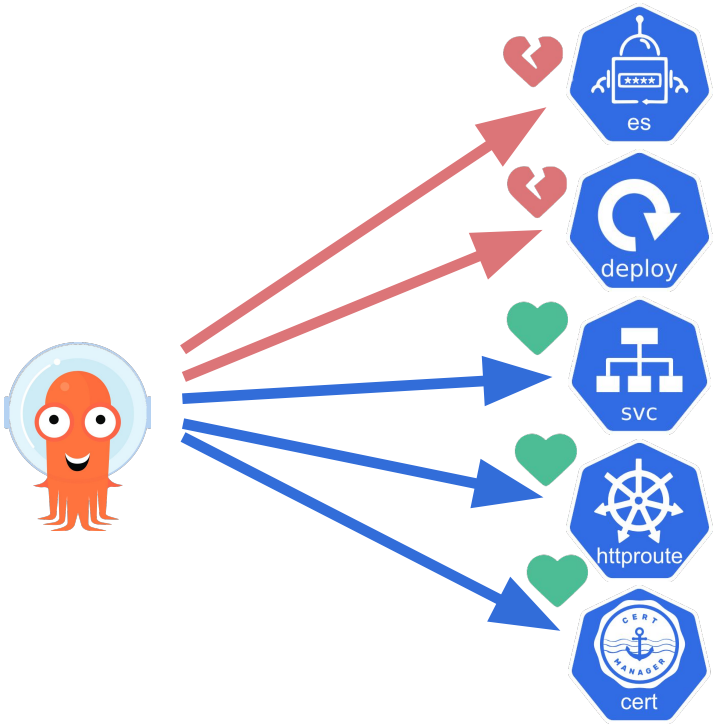
Health Checks



"It just works"

Someone *manually* made a custom health check for each resource

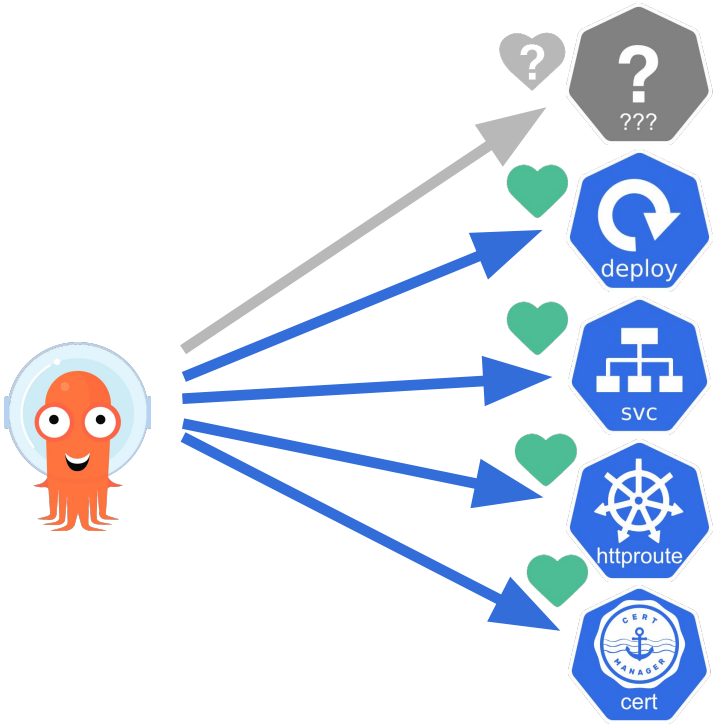
Health Checks



"It just works"

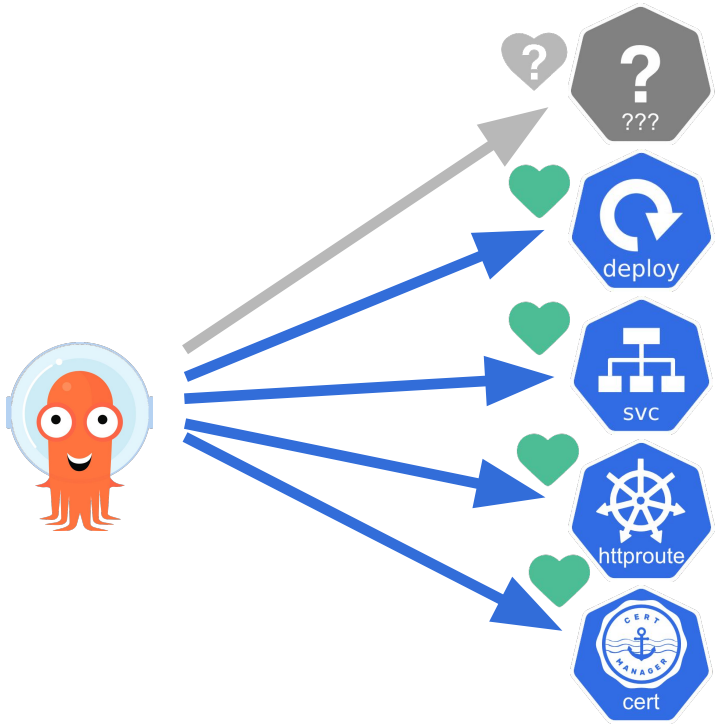
Someone *manually* made a custom health check for each resource

Health Checks



~~"It just works"~~

Health Checks




```lua
health_status = {
  status = "Progressing",
  message = "Provisioning ..."
}

for i, condition in ipairs(obj.status.conditions) do
  if condition.type == "Synced" then
    if condition.status == "False" then
      health_status.status = "Degraded"
      health_status.message = condition.message
      return health_status
    end
  end

  if condition.type == "RolloutCompleted" then
    if condition.status == "False" then
      health_status.status = "Degraded"
      health_status.message = condition.message
      return health_status
    end
  end

  if condition.type == "Ready" then
    if condition.status == "True" then
      health_status.status = "Healthy"
      health_status.message = "Resource is up-to-date."
      return health_status
    end
  end
end
```

When is this needed?



New or unpopular project



Homegrown tooling

Let’s take a look at

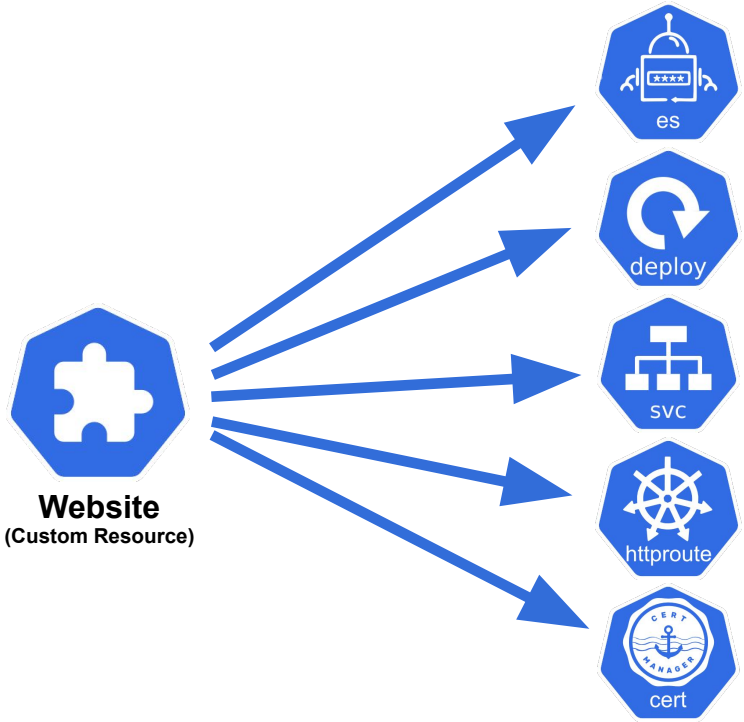


Crossplane

kro

Kube Resource Orchestrator

Health Checks



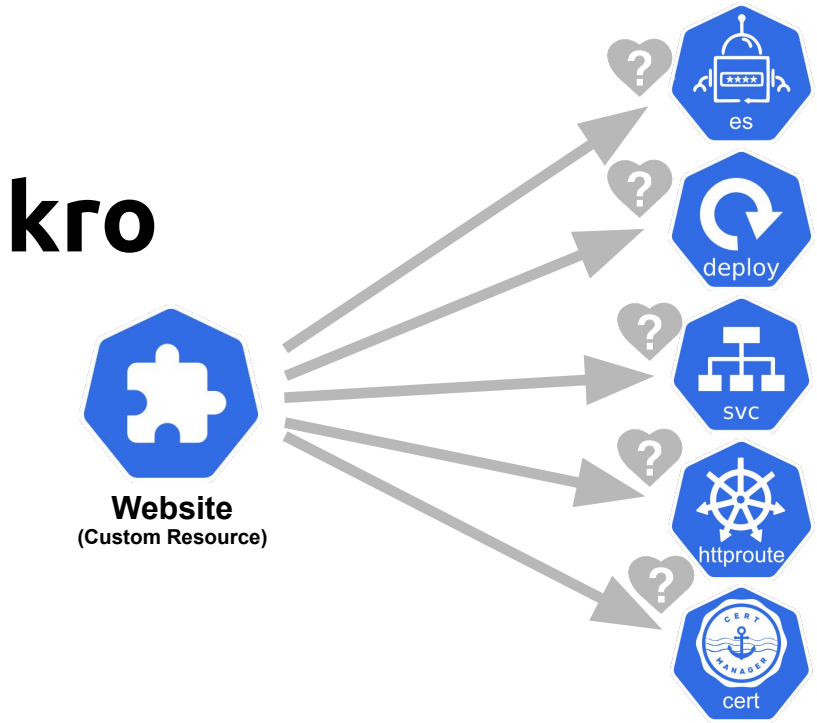


Crossplane

kro

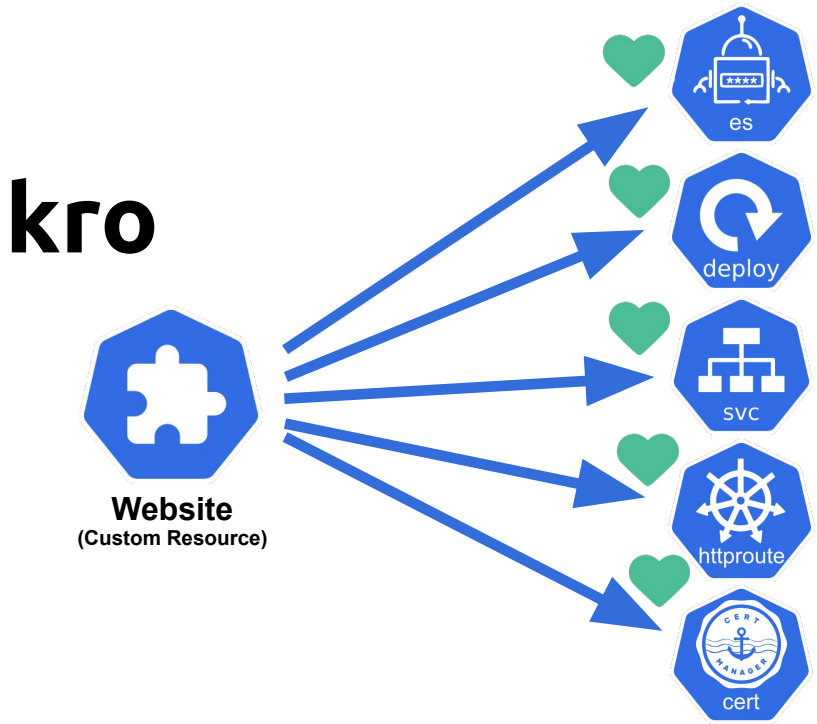
Kube Resource Orchestrator

Health Checks | Kro

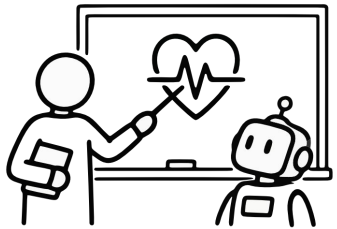


~~“It just works”~~

Health Checks | Kro

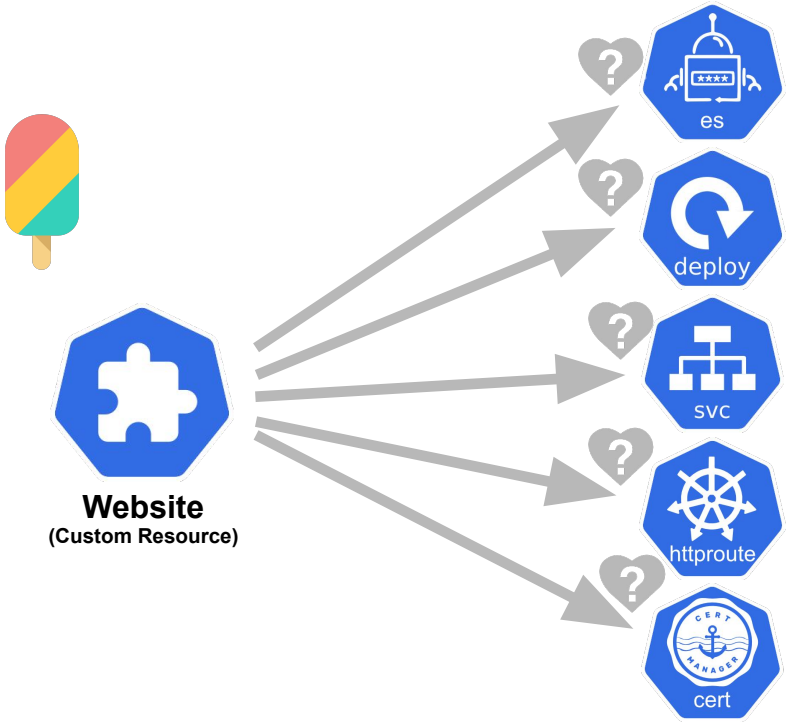


Kro



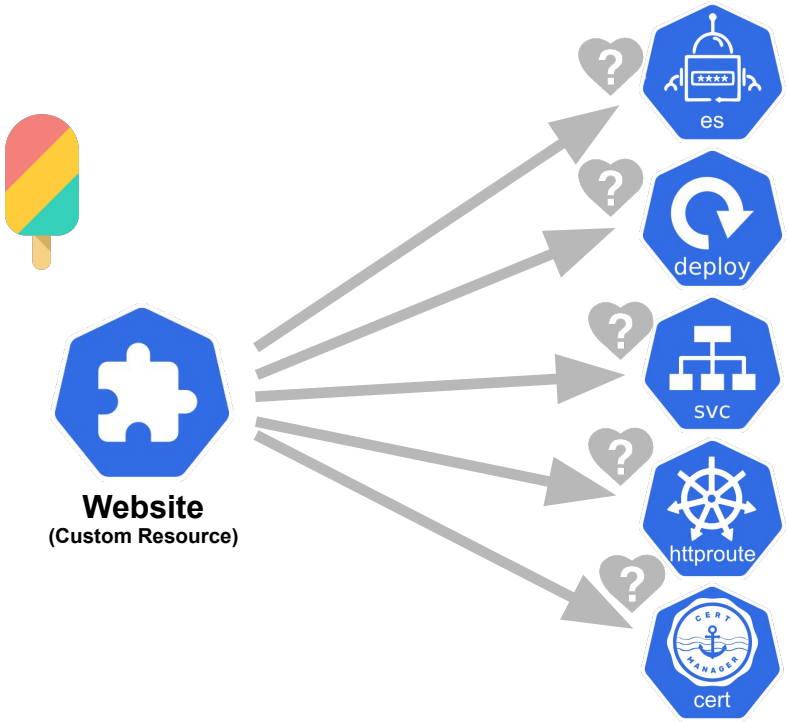
```yaml
apiVersion: kro.run/v1alpha1
kind: ResourceGraphDefinition
metadata:
  name: website-template
spec:
  schema:
    ...
  resources:
    - id: deployment
      readyWhen:
        - >-
          ${deployment.status.conditions.exists(
            c, c.type == 'Available' && c.status == 'True'
          )}
      template:
        apiVersion: apps/v1
        kind: Deployment
        metadata:
          name: ${schema.spec.name}
        spec:
          replicas: ${schema.spec.deployment.replicas}
          selector:
            matchLabels:
              app: ${schema.spec.name}
```

Health Checks | Crossplane



~~"It just works"~~

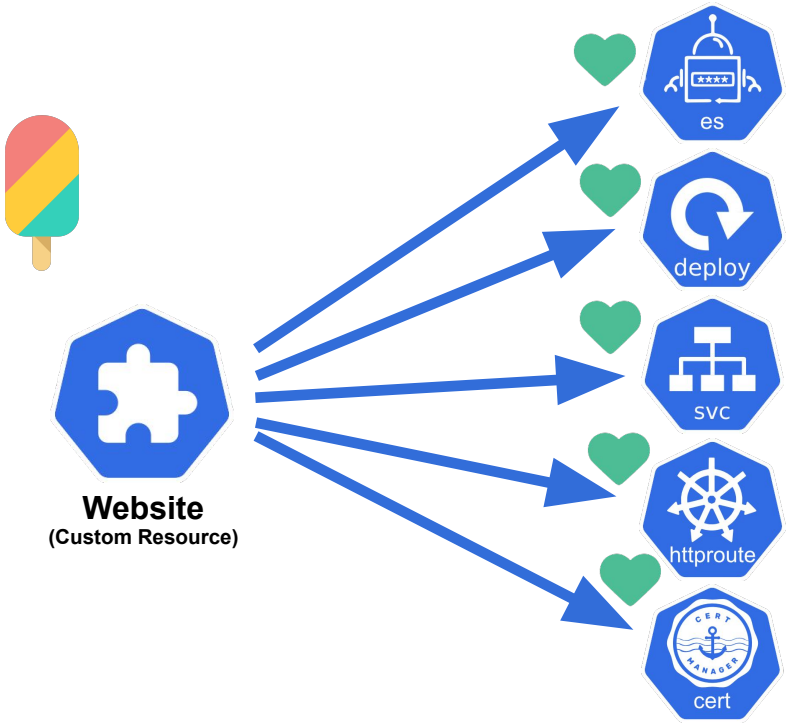
Health Checks | Crossplane



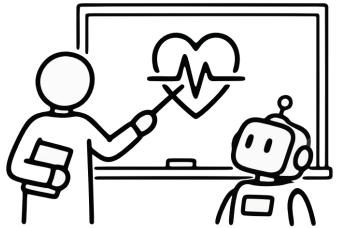
~~*"It just works"*~~

Composition Function:
`crossplane-contrib/function-auto-ready`
has some built-in health checks

Health Checks | Crossplane



Crossplane

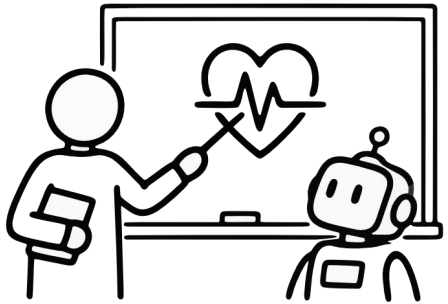


```yaml
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  name: website-templated
spec:
  compositeTypeRef:
    apiVersion: example.my-domain.io/v1
    kind: Website
  mode: Pipeline
  pipeline:
  - step: create-deployment-service-and-ingress
    functionRef:
      name: crossplane-contrib-function-go-templating
    input:
      ... some steps ...

  - step: crossplane-contrib-function-auto-ready
    functionRef:
      name: crossplane-contrib-function-auto-ready
    input:
      apiVersion: autoready.fn.crossplane.io/v1beta1
      kind: Input
      celHealthCheckCustomization:
        gateway.networking.k8s.io_v1_HTTPRoute: |-
          object.status.parents.exists(p,
            p.conditions.exists(c,
              c.type == 'Accepted' &&
              c.status == 'True' &&
              c.observedGeneration == object.metadata.generation) &&
            p.conditions.exists(c,
              c.type == 'ResolvedRefs' &&
              c.status == 'True' &&
              c.observedGeneration == object.metadata.generation))
```

Conclusion

No health standard

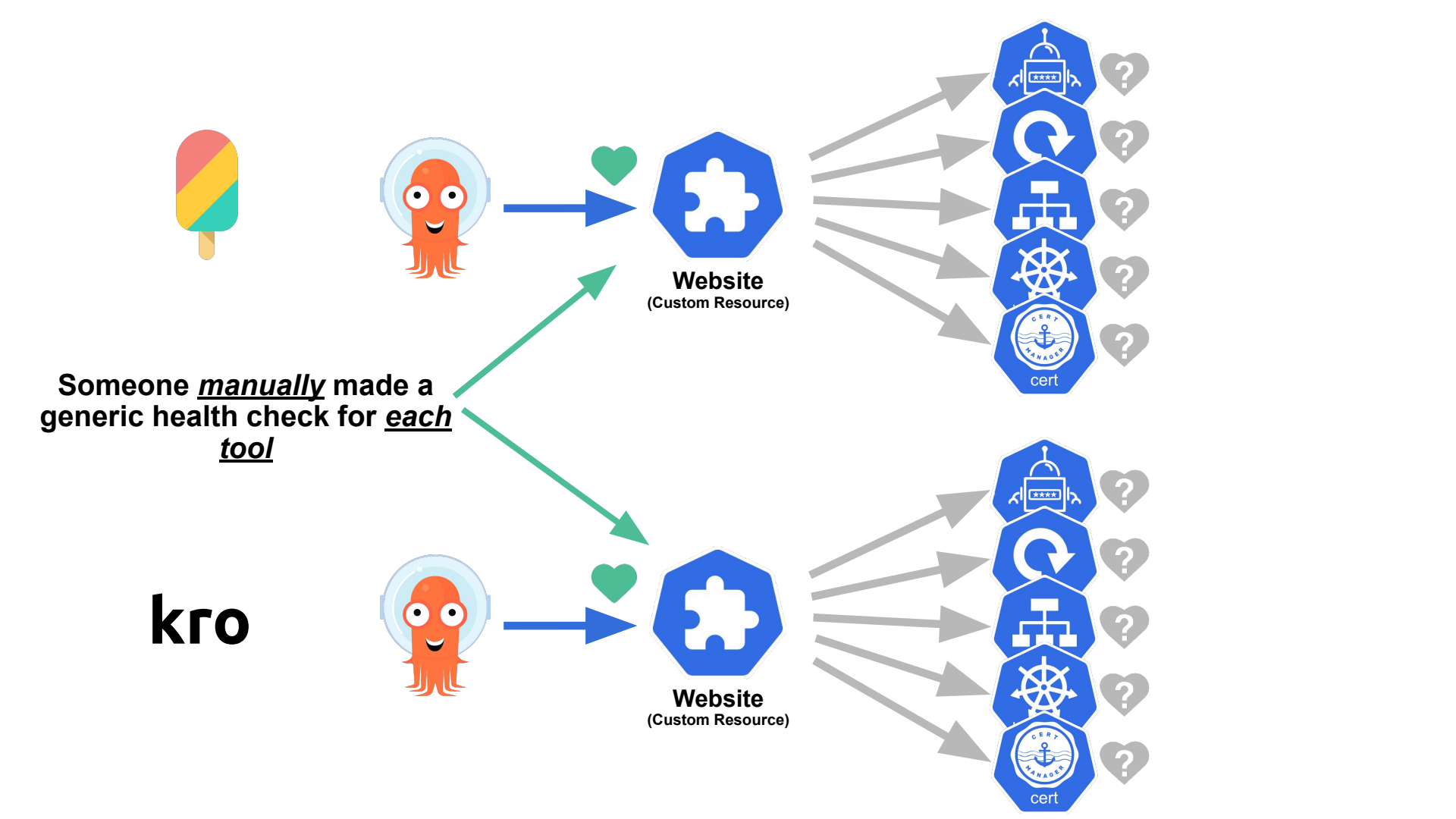


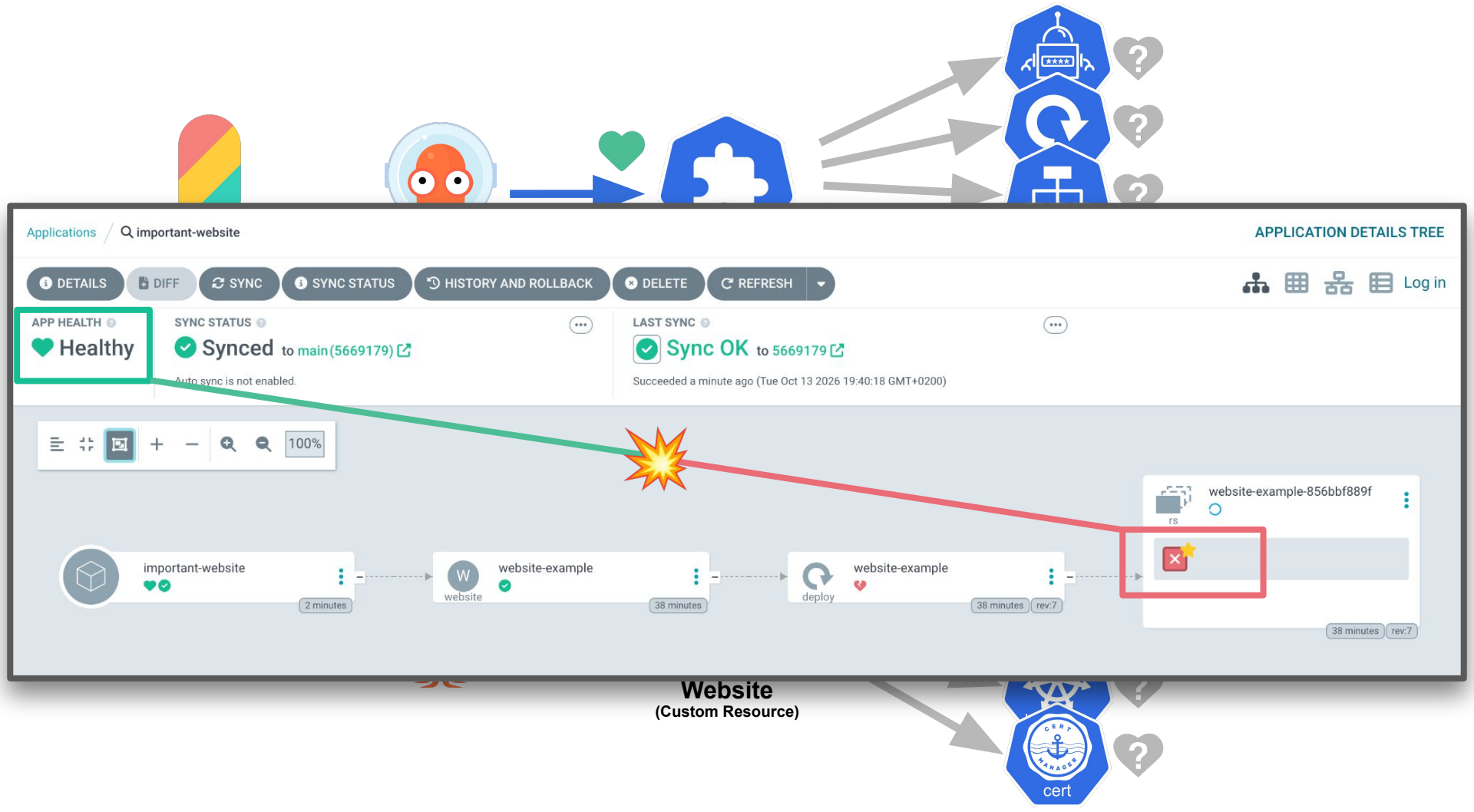
You have to teach the tool what health means



everyone has to reinvent the wheel

Let's combine Kro and Crossplane with Argo CD


Website
(Custom Resource)
cert
Someone *manually* made a generic health check for *each tool*
kro
Website
(Custom Resource)
cert


Applications
important-website
APPLICATION DETAILS TREE
DETAILS
DIFF
SYNC
SYNC STATUS
HISTORY AND ROLLBACK
DELETE
REFRESH
Log in
APP HEALTH
Healthy
SYNC STATUS
Synced to main (5669179)
Auto sync is not enabled.
LAST SYNC
Sync OK to 5669179
Succeeded a minute ago (Tue Oct 13 2026 19:40:18 GMT+0200)
100%
important-website
2 minutes
website-example
website
38 minutes
website-example
deploy
38 minutes
rev:7
website-example-856bbf889f
rs
38 minutes
rev:7
Website
(Custom Resource)
cert

kstatus

Attempt at fixing this problems



kstatus

A Library that has been around since 2020

Out of 77 (Graduate and Incubating) projects only 2 consumes it: Flux CD and Helm 4

*> kstatus provides tools for checking the status of Kubernetes resources. The primary use case is knowing when (or if) a given set of resources **has been fully reconciled** after an apply operation.*

https://github.com/kubernetes-sigs/cli-utils/blob/master/pkg/kstatus/README.md

- **Current**:
  The actual state of the resource matches the desired state, (...)
- **InProgress**:
  The actual state of the resource has not yet reached the desired state as specified in the resource manifest, (...)
- **Failed**:
  The process of reconciling the actual state with the desired state has encountered an error or it has made insufficient progress.
- **Terminating**:
  The resource is in the process of being deleted.
- **NotFound**:
  The resource does not exist in the cluster.
- **Unknown**:
  This is for situations when the library are unable to determine the status of a resource.

Kstatus vs. Argo CD health statuses

Kstatus		*Argo CD Health*
Current	⇒	Healthy
InProgress	⇒	Progressing
Failed	⇒	Degraded
NotFound	⇒	Missing
Unknown	⇒	Unknown
Terminating	⇒	Progressing
-	⇒	Suspended

kstatus: Current
does not
necessarily mean
Argo CD: Healthy

kstatus asks:
"Has the desired state been reconciled?"

Argo CD asks:
"Does the live resource appear operationally ready"

Problems with 1 dimensional statuses

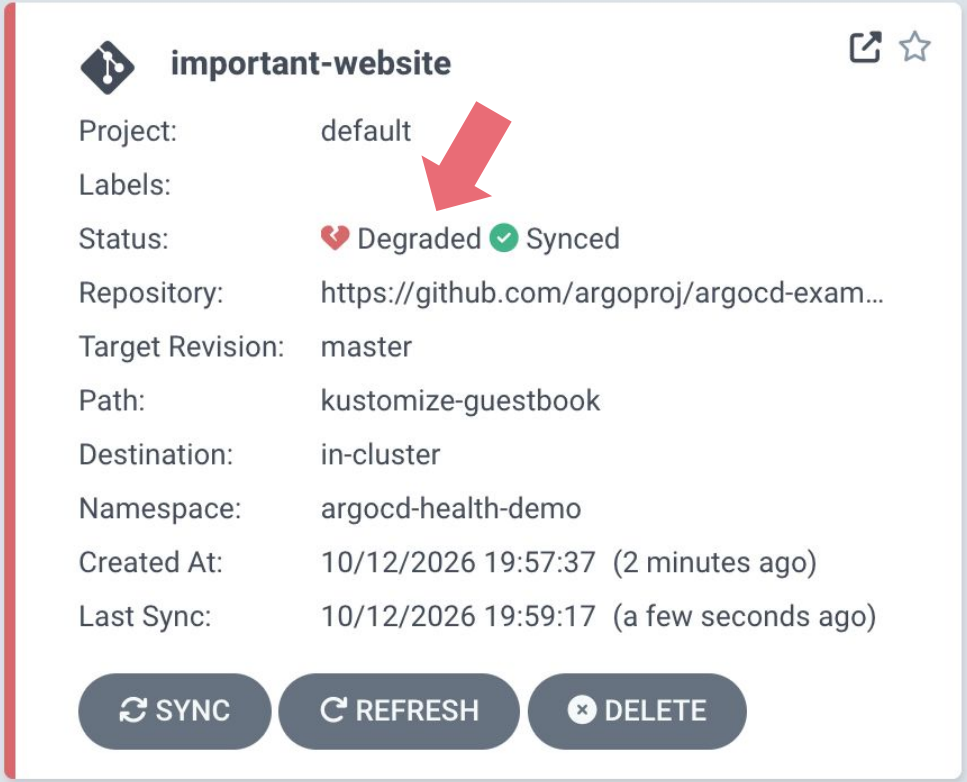


Argo CD example

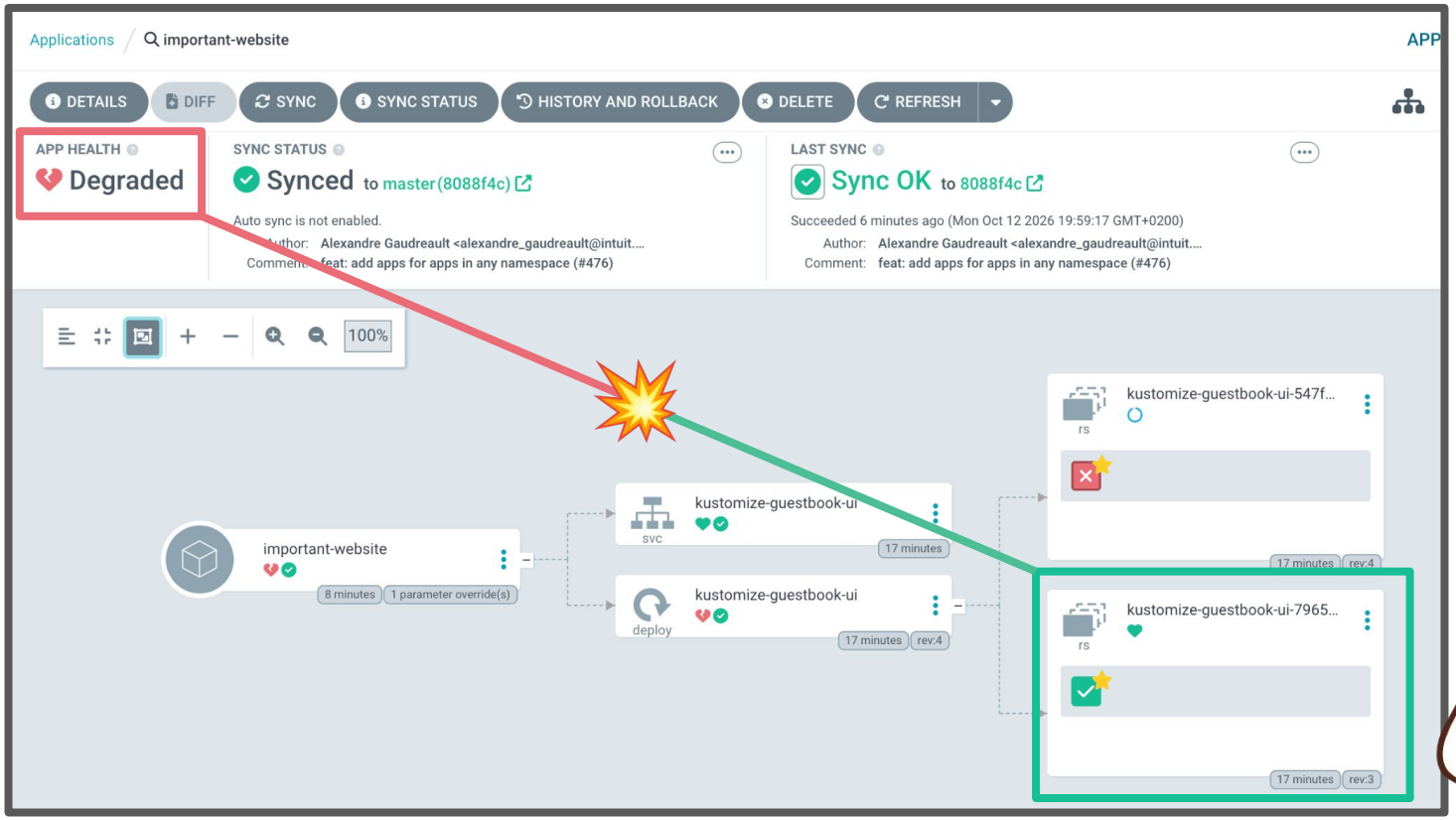
"Is my website down?"

"Should I panic?"

"Should i be running towards my computer?"



Argo CD has the same problem

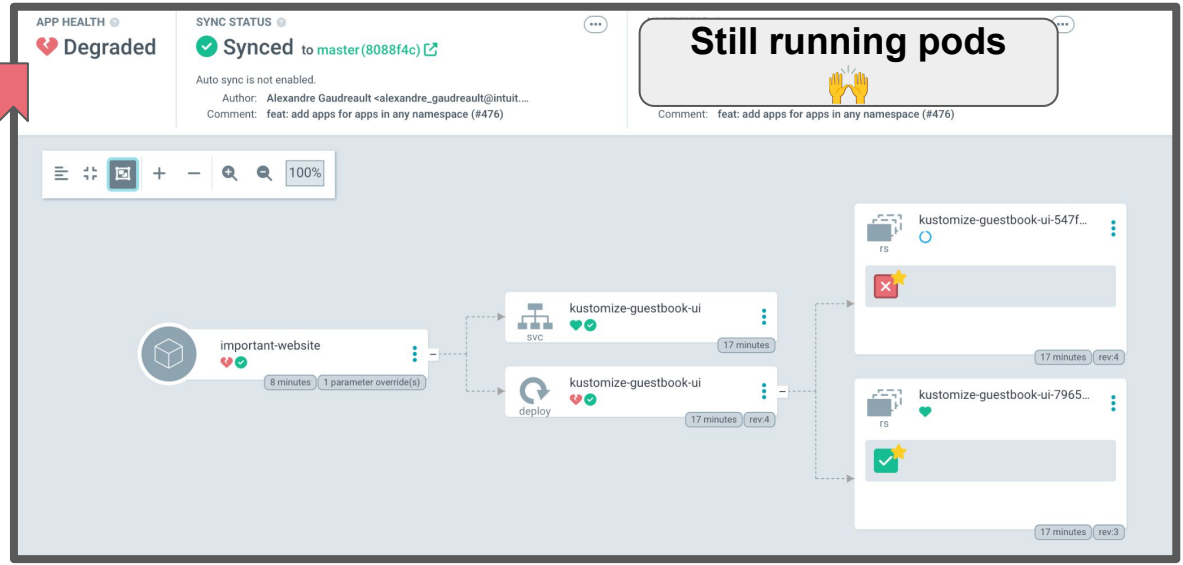


"My website is still available"

"I'll fix that later"



I can’t tell if we have available pods or not 😬



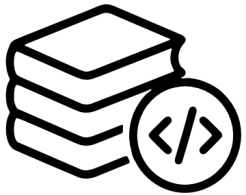


Creating a new standard

And how it could be used…

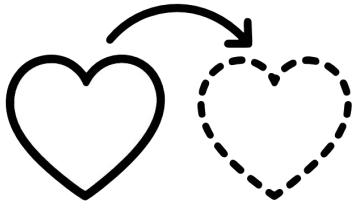


10:00



Kstatus' library idea

+



Copy-paste Argo CD's current catalog

+



Split status-value into 3 dimensions

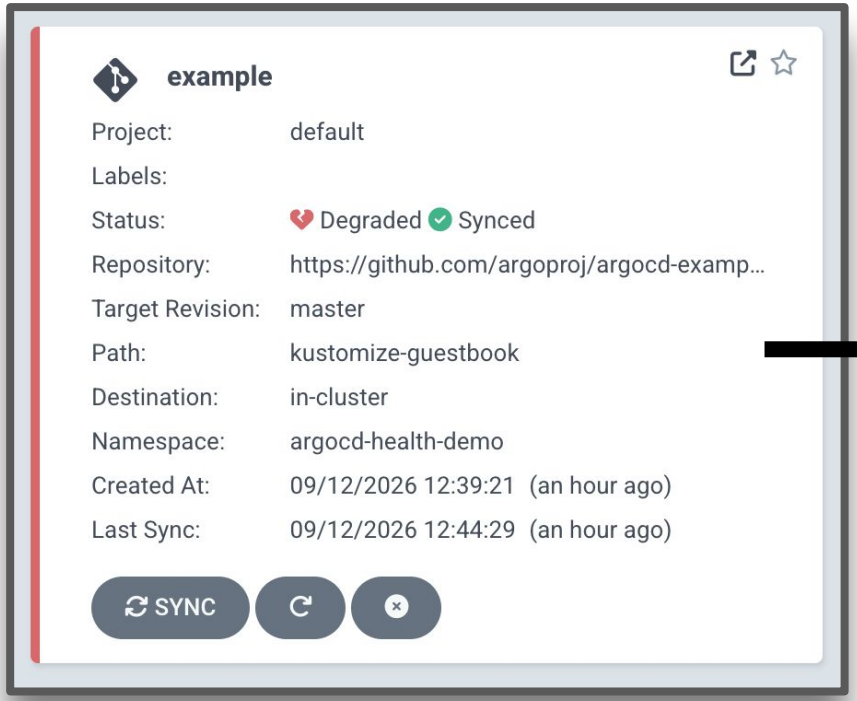
Introducing:

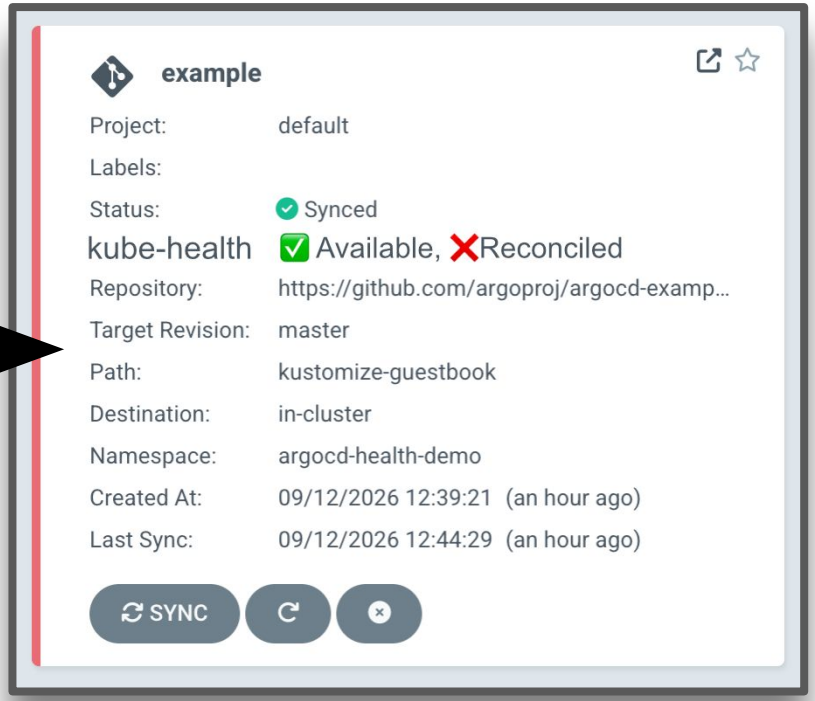
KubeHealth

Availability	Reconciliation	Life Cycle
● Available ● PartiallyAvailable ● Unavailable ● NotApplicable ● Unknown	● Reconciled ● InProgress ● Failed ● NotApplicable ● Unknown	● Initializing ● Active ● Terminating ● NotFound ● Unknown

https://github.com/kubehealth-project/kubehealth

Argo CD with KubeHealth





KubeHealth

We have a working Library here:
https://github.com/kubehealth-project/kubehealth

It currently has 35 health checks. Including native Kubernetes resources and ecosystem tools like:



External Secrets Operator



Kyverno



Cert Manager



Argo Rollouts



Prometheus Operator

KubeHealth is compatible with kstatus

- KubeHealth is a more rich standard, that contains a superset of the information that kstatus delivers

- That means you can always reduce the KubeHealth to kstatus

```
switch KubeHealth.Availability {
    // not handled
}

switch kubehealth.Lifecycle {
case LifecycleInitializing                  → return Kstatus.InProgress
case LifecycleTerminating                   → return Kstatus.Terminating
case LifecycleNotFound                      → return Kstatus.NotFound
case LifecycleUnknown                       → return Kstatus.Unknown

switch kubehealth.Reconciliation {
case ReconciliationReconciled               → return Kstatus.Reconciled
case ReconciliationInProgress               → return Kstatus.InProgress
case ReconciliationFailed                   → return Kstatus.Failed
case ReconciliationSuspended                → return Kstatus.InProgress
case ReconciliationUnknown                  → return Kstatus.Unknown

return Kstatus.Current
```

How does it work?

1. First it looks for these fields ➡️
   a. Set by the operator/maintainer
2. Secondly check the KubeHealth resource catalog (written the community)
   a. https://github.com/kubehealth-project/kubehealth

```yaml
apiVersion: xxx
kind: xxx
metadata: xxx
spec: xxx
status:
  kubeHealth:
    contractVersion: v1alpha1
    observedGeneration: 12

    reconciliation:
      status: Failed
      reason: ProgressDeadlineExceeded
      message: The desired revision did not become ready
      lastTransitionTime: "2026-09-13T12:00:00Z"

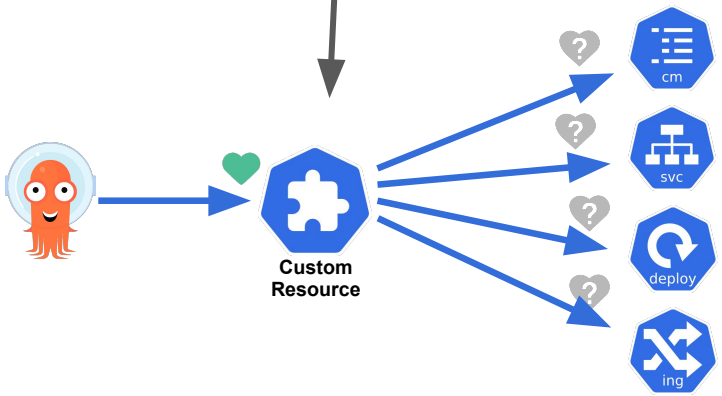
    availability:
      status: Available
      reason: PreviousRevisionServing
      message: The previous revision continues to serve traffic
      lastTransitionTime: "2026-09-13T11:55:00Z"

    lifecycle:
      status: Active
      reason: DeletionNotRequested
      lastTransitionTime: "2026-09-13T10:00:00Z"
```

```yaml
reconciliation:
  status: Failed
  reason: ProgressDeadlineExceeded
  message: The desired revision did not become ready
  lastTransitionTime: "2026-09-13T12:00:00Z"

availability:
  status: Available
  reason: PreviousRevisionServing
  message: The previous revision continues to serve traffic
  lastTransitionTime: "2026-09-13T11:55:00Z"

lifecycle:
  status: Active
  reason: DeletionNotRequested
  lastTransitionTime: "2026-09-13T10:00:00Z"
```





Homegrown Operator



Crossplane Compositions

kro

KubeHealth | Automation health aggregation

Kro implementation:
https://github.com/kubehealth-project/kro

Crossplane function:
https://github.com/kubehealth-project/function-kubehealth

Kro

```yaml
apiVersion: kro.run/v1alpha1
kind: ResourceGraphDefinition
metadata:
  name: website-template
spec:
  kubehealth:
    standardHealthAggregation: true
  schema:
    apiVersion: v1alpha1
    kind: Website
    spec:
      name: string
      deployment:
        image: string | default="nginx"
        replicas: integer | default=3
      service:
```

kro

Crossplane

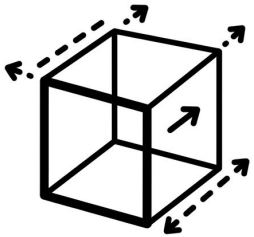
```yaml
apiVersion: pkg.crossplane.io/v1
kind: Function
metadata:
  name: crossplane-function
spec:
  package: ghcr.io/kubehealth-project/crossplane-function:v0.1.0
```

```yaml
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  name: website-templated-yaml
spec:
  compositeTypeRef:
    apiVersion: example.my-domain.io/v1
    kind: Website
  mode: Pipeline
  pipeline:
  - step: create-deployment-service-and-ingress
    functionRef:
      name: crossplane-contrib-function-go-templating
    input:
      ... some steps ...
  - step: propagate-kubehealth
    functionRef:
      name: crossplane-function
```

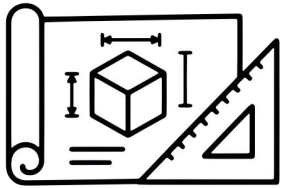




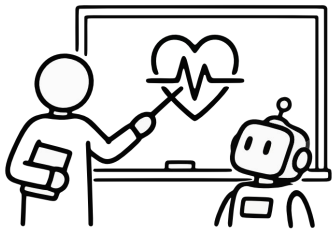
Just an idea



More dimensions?



KubeHealth is just a prototype



Users should not re-teach each tool what healthy means



Maintainers should not re-implement health logic

Health check ownership

Who maintains the health checks?



Proposing ownership shift:

Argo CD Maintainers => CR Maintainers



2 paths for the shift

Either

A group of CR maintainers



Health checks library maintenance

Or

Each CR maintainer



CR status and controller code change

Shared health check library

Maintained by CR maintainers

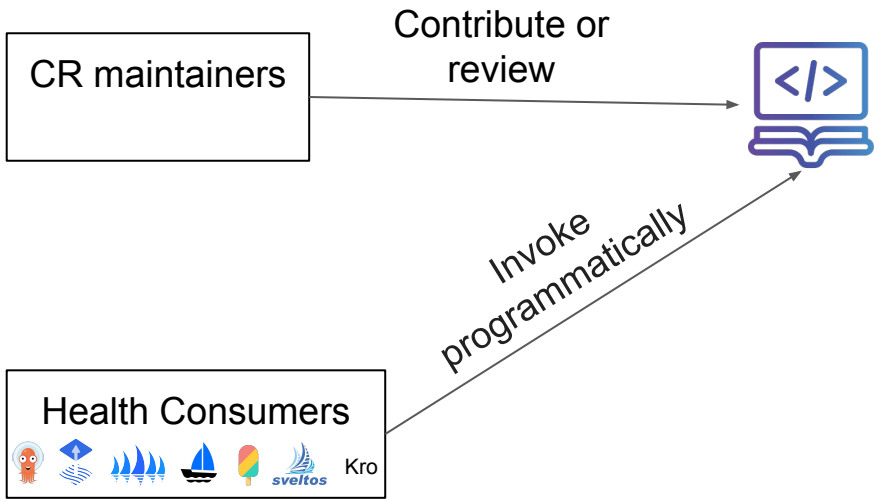


Important

Even if we don’t create a new health standard, we can still just move the current Argo CD health logic into a new library.

However this only makes sense if the community is willing to maintain their own health logic. If the Argo CD maintainers have to maintain this health logic, then this’d just make our work harder.

Health consumed from a library



Pros:
- Requires the least amount of work
- Closest to what we have today

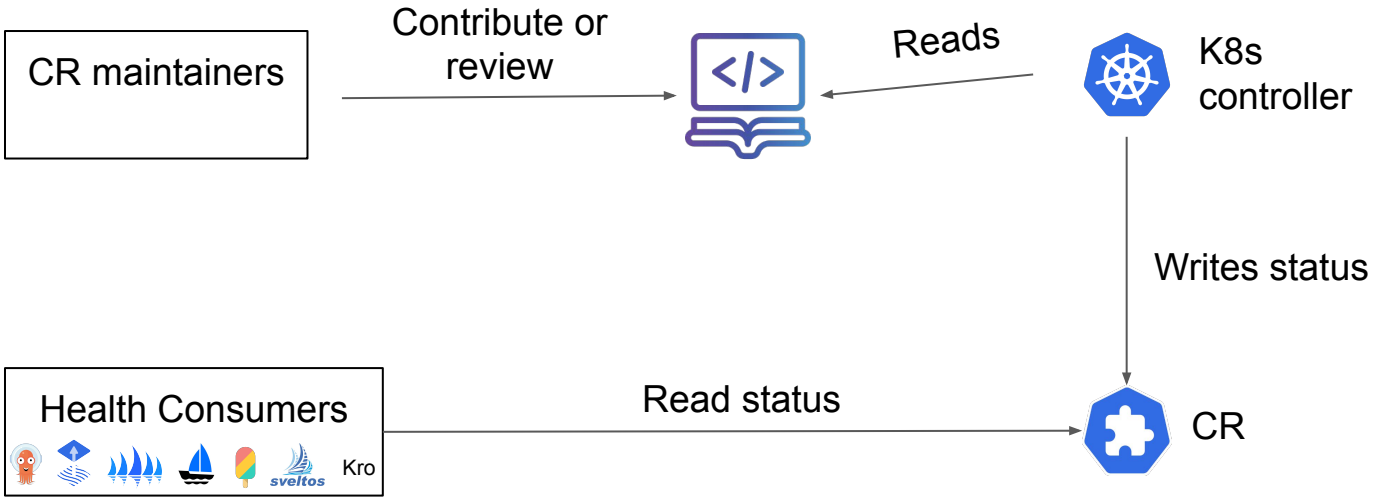
Cons:
- Language dependent
- You need to have the lib in order to work with this

Health controller

Apply KubeHealth status to all resources



Health consumed directly from CR



Pros:
- No need for a lib
- Just use K8s API

Cons:
- Single point of failure
- Latency
- Stale data?

If it used KubeHealth it could work like this

How it works:

- Loops over resources
- Lookup the kubehealth from lib
- Apply the 3 dimensions as status on the inspected resource

The operator sets these 3 statuses

➡️

on all resources continuously

```yaml
apiVersion: xxx
kind: xxx
metadata: xxx
spec: xxx
status:
  kubeHealth:
    contractVersion: v1alpha1
    observedGeneration: 12

    reconciliation:
      status: Failed
      reason: ProgressDeadlineExceeded
      message: The desired revision did not become ready
      lastTransitionTime: "2026-09-13T12:00:00Z"

    availability:
      status: Available
      reason: PreviousRevisionServing
      message: The previous revision continues to serve traffic
      lastTransitionTime: "2026-09-13T11:55:00Z"

    lifecycle:
      status: Active
      reason: DeletionNotRequested
      lastTransitionTime: "2026-09-13T10:00:00Z"
```

Aggregated API

Return the health check in a CR-like way



K8s Aggregated API

This draft solution explores the idea of exposing the health checks as a K8s API service.

https://github.com/crenshaw-dev/argocd-health-apiservice

Author: Michael Crenshaw, Argo CD lead maintainer

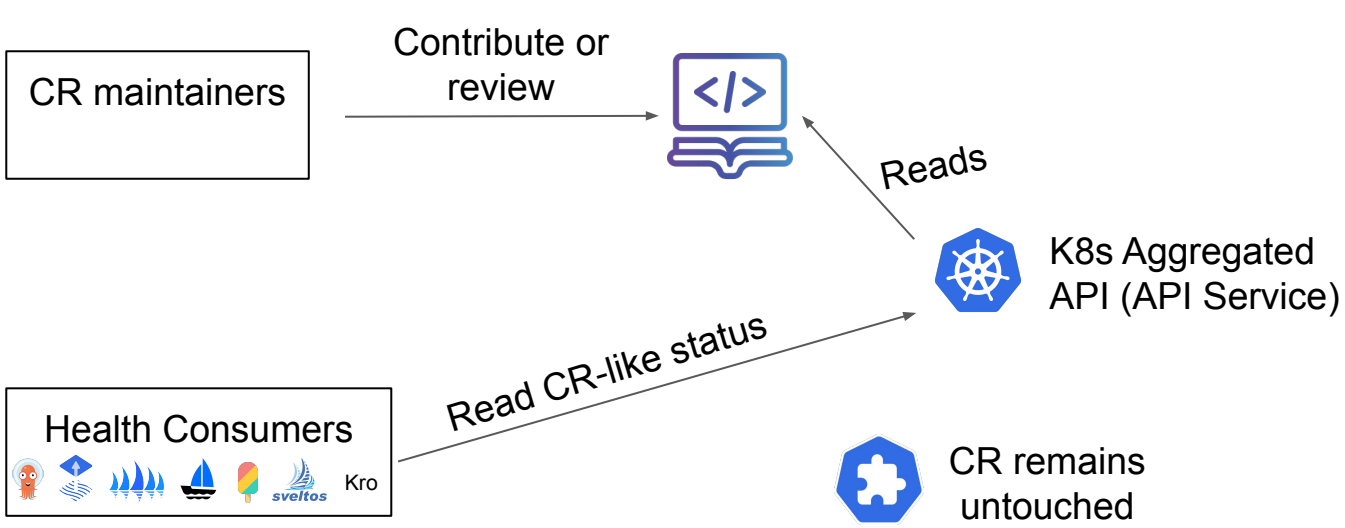
Currently based on Argo CD health spec and bundled health checks

Usage

```
kubectl get resourcehealths -n default
kubectl get rh -n default
kubectl get clusterresourcehealths
```

```yaml
apiVersion: health.crenshaw.dev/v1alpha1
kind: ResourceHealth
metadata:
  name: nginx.deployment.apps
  namespace: default
  ownerReferences:
    - apiVersion: apps/v1
      kind: Deployment
      name: nginx
      uid: ...
spec:
  group: apps          # empty for core
  kind: Deployment
  name: nginx
status:
  status: Healthy      # Unknown | Progressing | Healthy | Suspended | Degraded | Missing
  message: ""
```

Health consumed from a K8s API Service



Pros:
- No need for a lib
- Just use K8s API
- No ETCD load

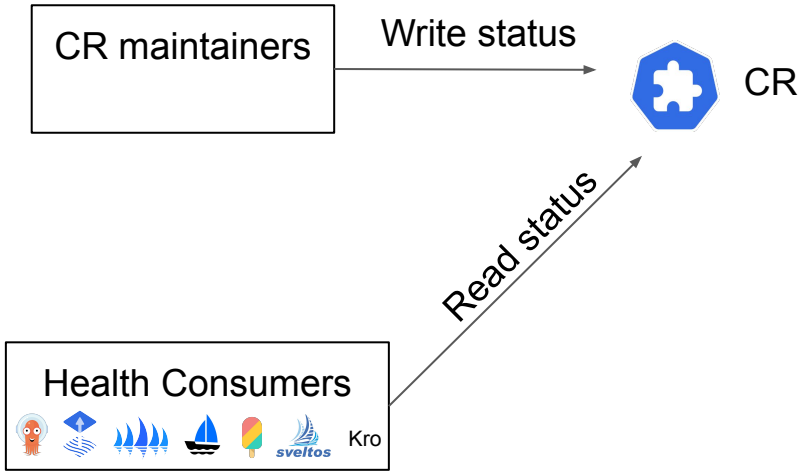
Cons:
- Single point of failure
- Latency
- Stale data?

Health checks in CR status

CR maintainer owned status and controller code changes



Health consumed from a CR without a library



My dream:

We wouldn't have this problem at all, because maintainers would have just applied this common health standard to the resource itself, without a lib

Summary

What did we learn today?



What did we learn?

Challenges with health check logic ownership, knowledge, and placement

Challenges with using a 1 dimensional health indicator

Potential solutions to these challenges including how to create a new standard + move health checks out of Argo CD.

So what now?

Do you, the maintainers, agree that what we present is a real problem?

Would you go and contribute your health checks to a shared library?

.. or maybe even add a new status section and follow a convention if we set it together?