

NeoNephos Foundation

Platform Mesh

Breaking API Lock-In for True Multi-Cloud Service Portability

Mirza Kopic –



Mangirdas (MJ) Judeikis –



**...and we are building a
platform... yet another one...**

...and we are building a platform... yet another one...



I feel your pain. "Platform" has become the "synergy" of the 2020s—a word that often means everything and nothing at the same time. If you lead with "we're building a platform," half your audience will mentally check their email.

To fix this, you need to acknowledge the cliché, kill it, and then redefine what *your* platform actually does.



kcp & friends*



kube-bind



multicluster-runtime



Platform Mesh &
friends**

* api-syncagent, init-agent, kcp-operator and more

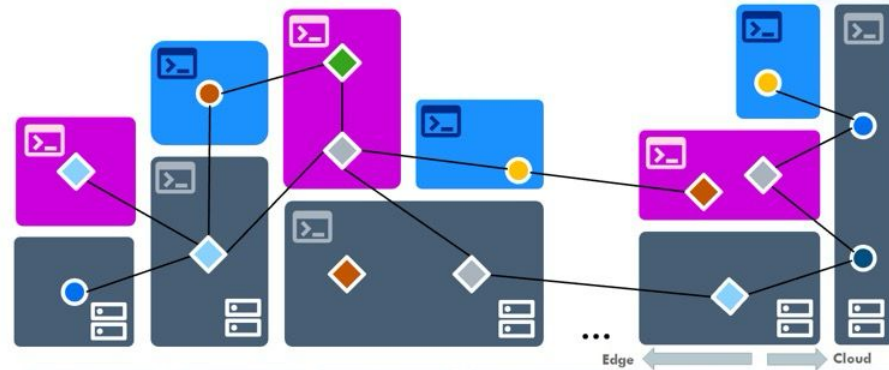
** resource-broken, kubernetes-graphql-gateway and more

Multi-Provider Cloud-Edge Continuum

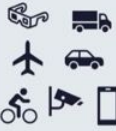
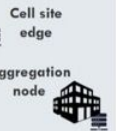
Digital (Public) Services

Ecosystem: Software Providers

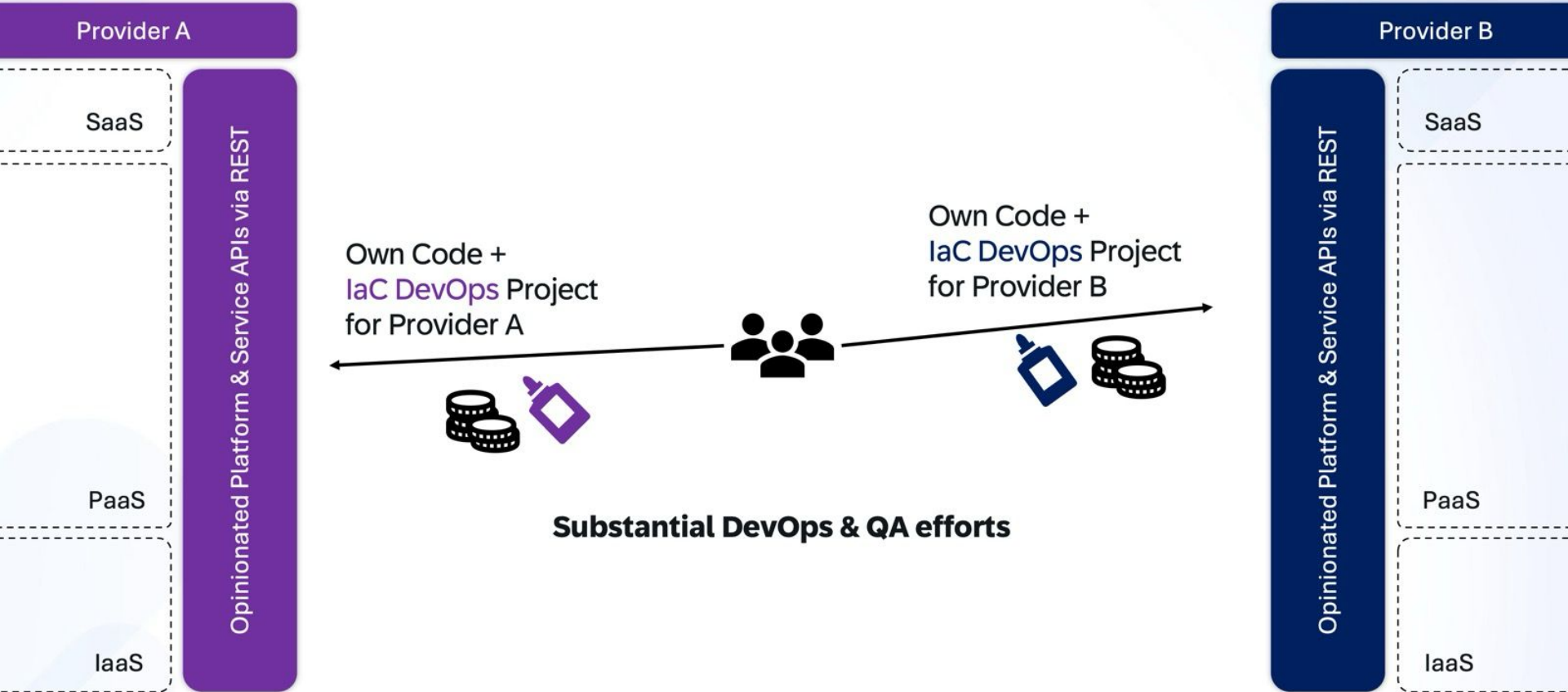
Ecosystem: Infrastructure Providers




 CLOUD·EDGE
 CONTINUUM

	ON-DEVICE	ON-PREMISE	FAR EDGE	NEAR EDGE	REGIONAL EDGE	CLOUD
						
	Millions	100,000s	1,000s 100s	10s	10s	<10s
Typical distance		<1 km	1-100 km	100-1000 km		>1000 km
Average latency		1 ms	2-5 ms	10-20 ms		>20 ms
Power range		kW	20-100 kW	100kW-1MW	5-30 MW	50-100 MW

Business/End User Experience: Fragmentation, Lock-in, Snowflakes



Platform Mesh

How can we offer services from multiple vendors through a single, uniform interface — with consistent definitions for ordering, discovery, and API management — all powered by KRM?



Platform Mesh + Service Portability

How can we offer services from multiple vendors through a single, uniform interface — with consistent definitions for ordering, discovery, and API management — all powered by KRM?



Azure

```
kind: Cluster
name: my-cluster
finalizers:
  finalizer: true
spec:
  provider: azure
count: 3
```

Switch Provider



AWS

```
kind: Cluster
name: my-cluster
finalizers:
  finalizer: true
spec:
  provider: aws
count: 3
```

Agenda

1. Problem Description
2. Service with a multiple provider and multiple implementations
3. Demo
4. Summary

Part 1:

Problem: No consistent way to consume services across the continuum

THE THREE-VENDOR PROBLEM



To build a public (or private) cloud - is (still) feels like lego.

A necessary shift in mindset



Traditional

lots of fun, great for creativity, but everything built out of these will look differently. Generally too low level.



A more Grown up Approach

Everything lives in one environment. We want off-the-shelf components. We want one environment where we can collaborate. Where solution from one party fit to setup of somebody else.



Previous Talk *

To build a public (or private) cloud - is (still) feels like lego.

A necessary shift in mindset



Traditional

lots of fun, great for creativity, but everything built out of these will look differently. Generally too low level.

We are moving...



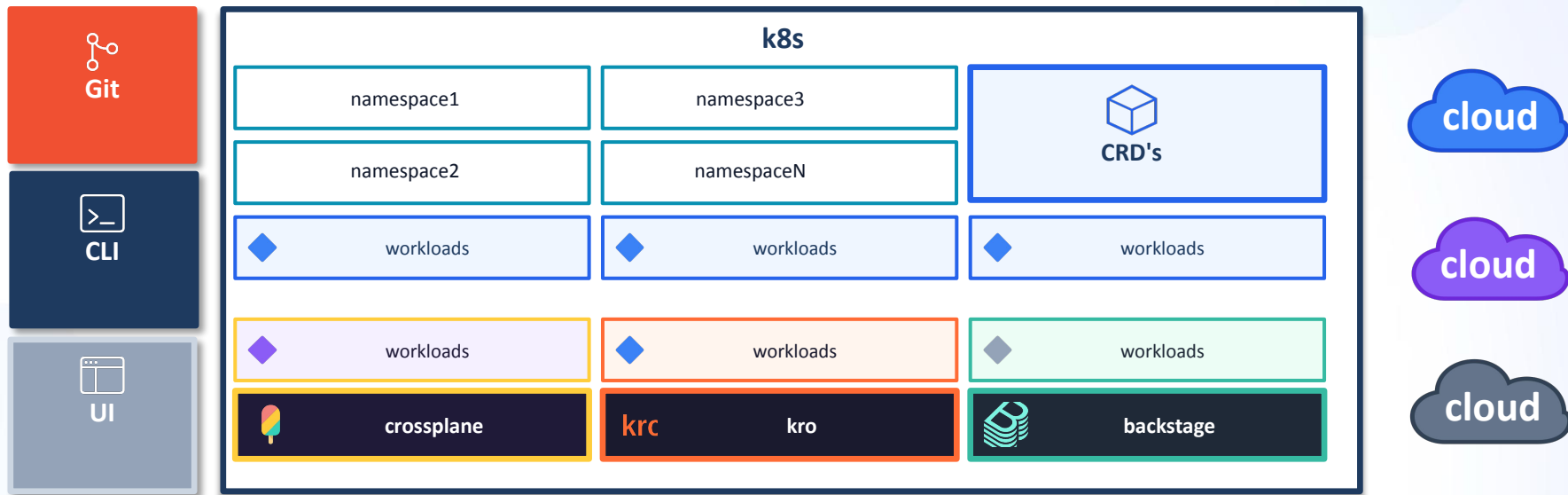
A more Grown up Approach

Everything lives in one environment. We want off-the-shelf components. We want one environment where we can collaborate. Where solution from one party fit to setup of somebody else.



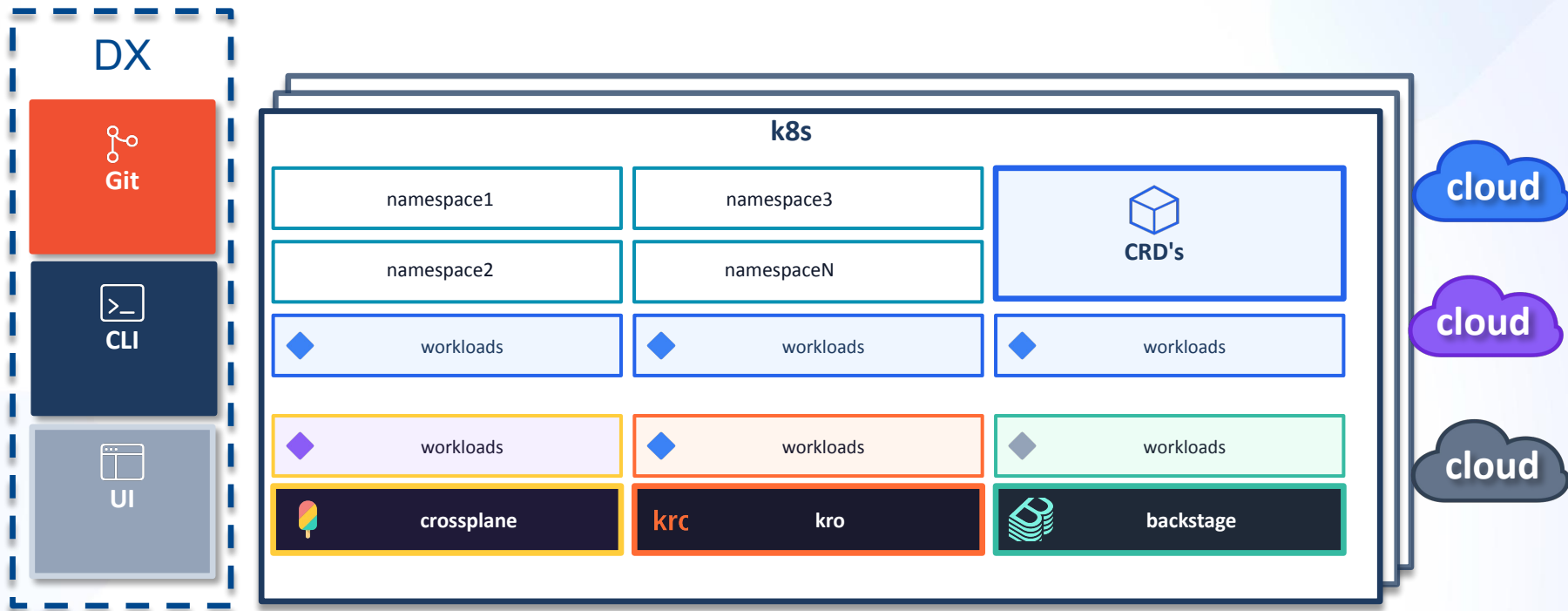
Previous Talk *

Todays Platforms: Assembly Required



Typical Cloud Scenario - YMMV

Today's Platforms: Assembly Required



Platforms in the wild

The platform challenge



KubeCon



CloudNativeCon

Europe 2024

Three Personas – Three Views



The platform owner

the ones with the keys
for everything,
the ones connecting and aligning
everything



The service provider

AI/ML service team,
DB team, PKI team,
CI/CD service, 3rd-parties
service owner



The users

Developer,
Data scientist,
application owner



KubeCon 2024 *

Se

33

Platforms in the wild

The platform challenge



KubeCon



CloudNativeCon

Europe 2024

Three Personas – Three Views



The platform owner

the ones with the keys
for everything,
the ones connecting and aligning
everything



The service provider

AI/ML service team,
DB team, PKI team,
CI/CD service, 3rd-parties
service owner



The users

Developer,
Data scientist,
application owner



KubeCon 2024 *

Se

33

Platforms in the wild

The platform challenge



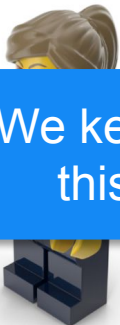
KubeCon



CloudNativeCon

Europe 2024

Three Personas – Three Views



We keep ignoring
this person!

The platform owner

the ones with the keys
for everything,
the ones connecting and aligning
everything



The service provider

AI/ML service team,
DB team, PKI team,
CI/CD service, 3rd-parties
service owner



The users

Developer,
Data scientist,
application owner



KubeCon 2024 *

Se

33

kcp - Kubernetes-like Control Plane @CNCF

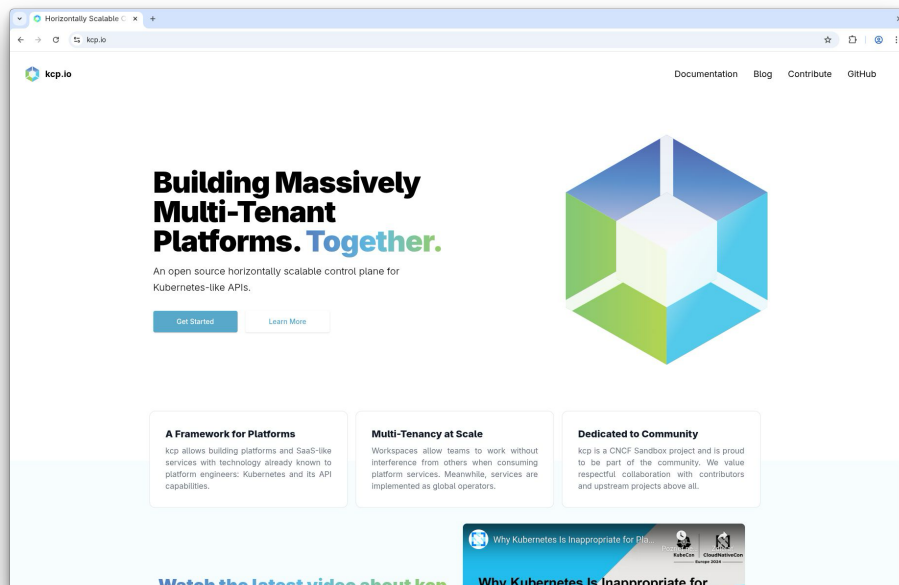


“A horizontally scalable **control-plane** for Kubernetes-style APIs”

“**Framework** for building platforms”

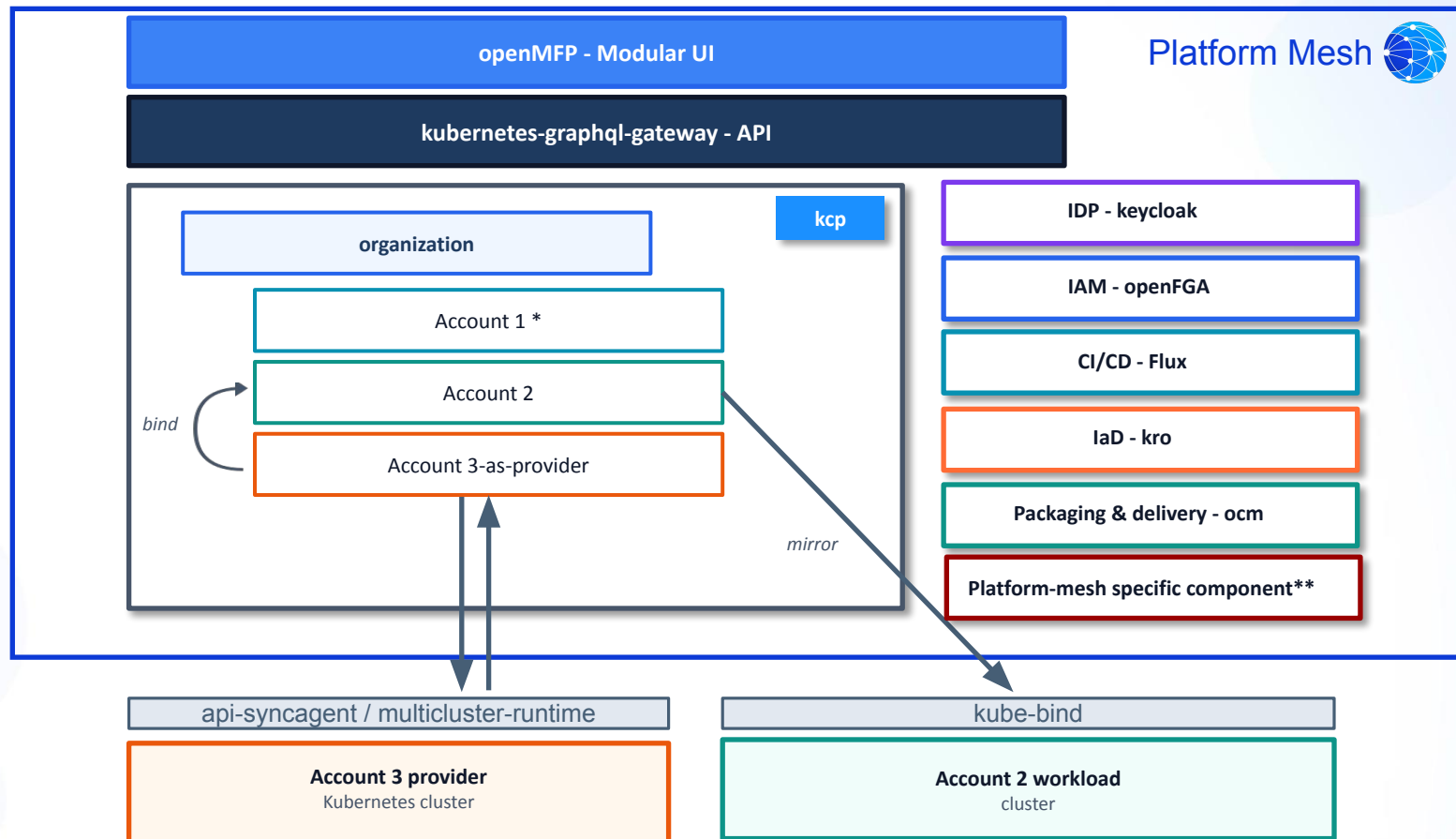
“**APIs as a Service**”

“**Multi-Tenancy**, fully-isolated workspaces”



“Our” opinionated platform

Platform mesh - high level architecture



* Account = Workspace

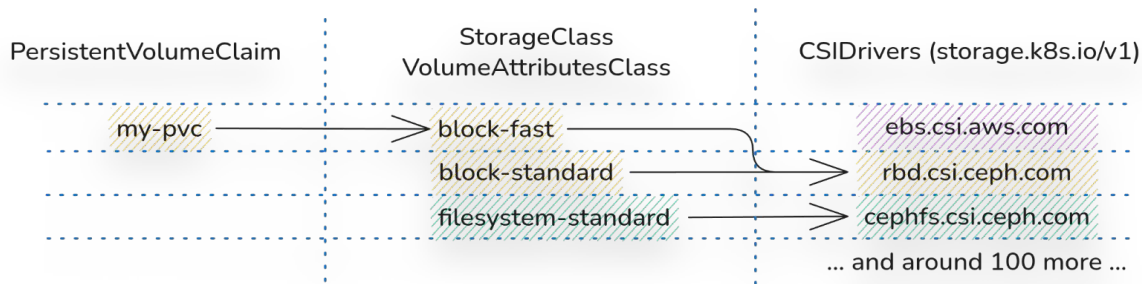
** Resource-broker, security-operator, and more

Part 2: Service with a multiple provider and multiple implementations

X implementations : M provider : N consumers

Well-established concept in vanilla Kubernetes!

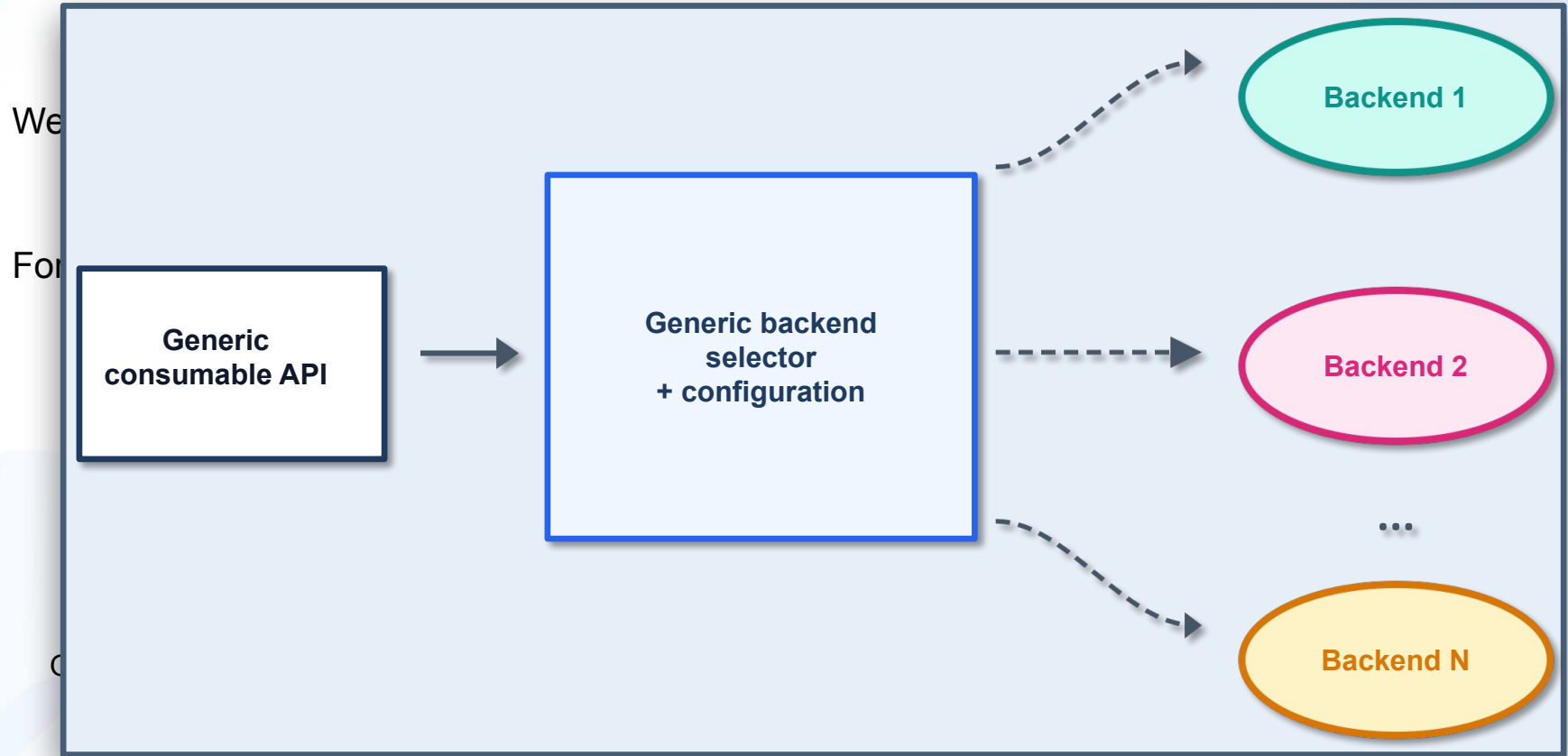
For example:



Other examples:

- GatewayClass (gateway.networking.k8s.io/v1),
- RuntimeClass (node.k8s.io/v1),
- You can create your own with CRDs...

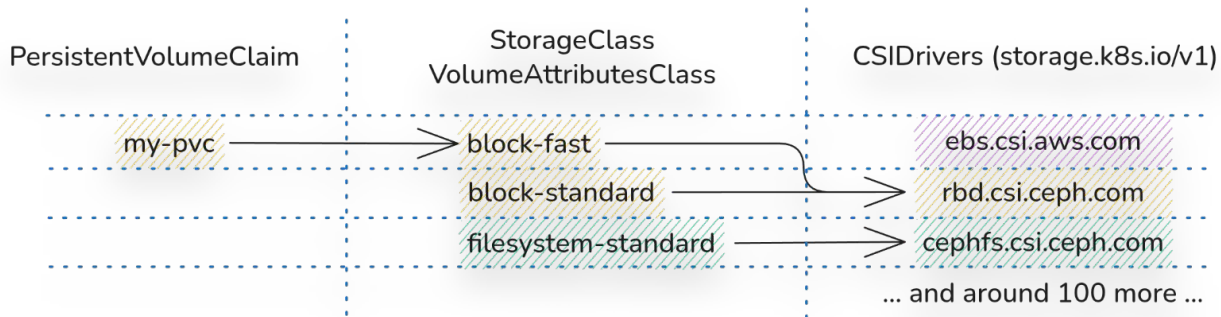
X implementations : M provider : N consumers



X implementations : M provider : N consumers

Well-established concept in vanilla Kubernetes!

For example:



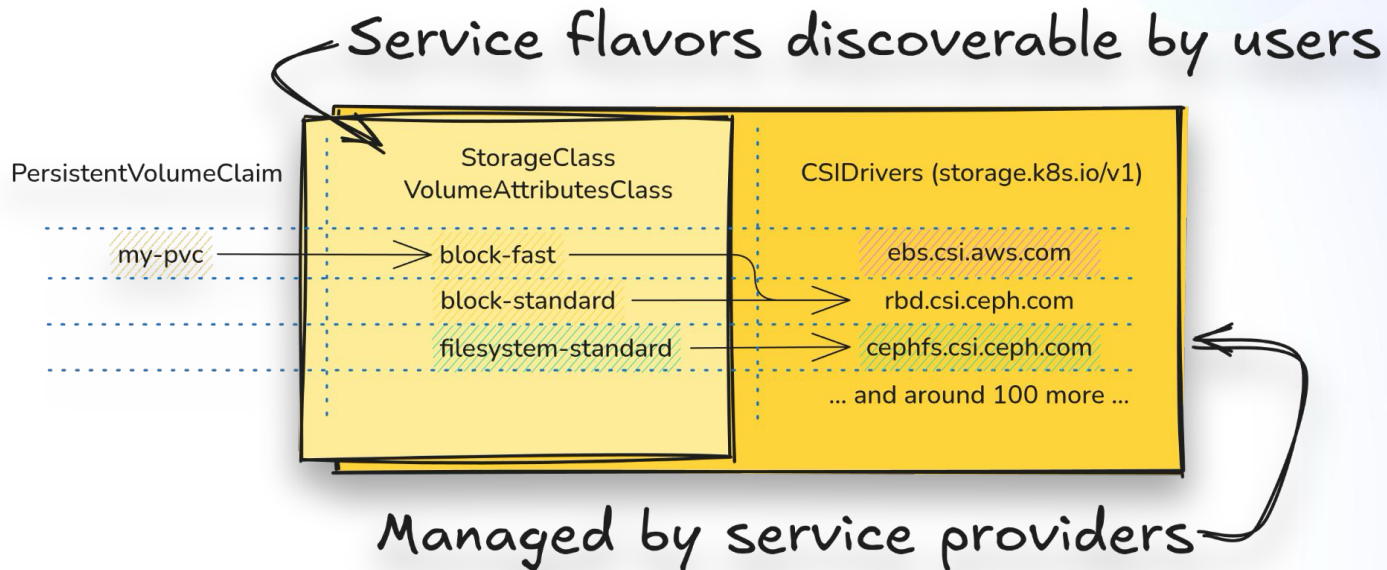
Other examples:

- GatewayClass (gateway.networking.k8s.io/v1),
- RuntimeClass (node.k8s.io/v1),
- You can create your own with CRDs...

X implementations : M provider : N consumers

Well-established concept in vanilla Kubernetes!

For example:



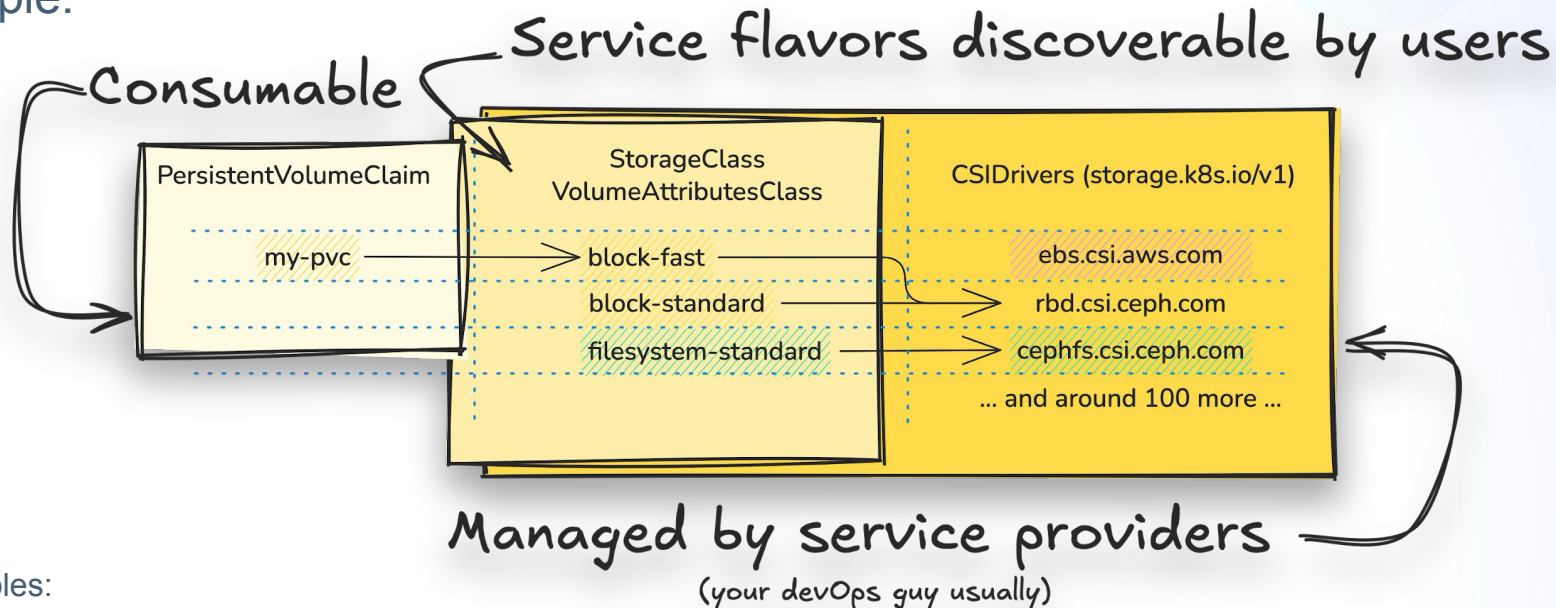
Other examples:

- GatewayClass (gateway.networking.k8s.io/v1)
- RuntimeClass (node.k8s.io/v1)
- You can create your own with CRDs...

X implementations : M provider : N consumers

Well-established concept in vanilla Kubernetes!

For example:



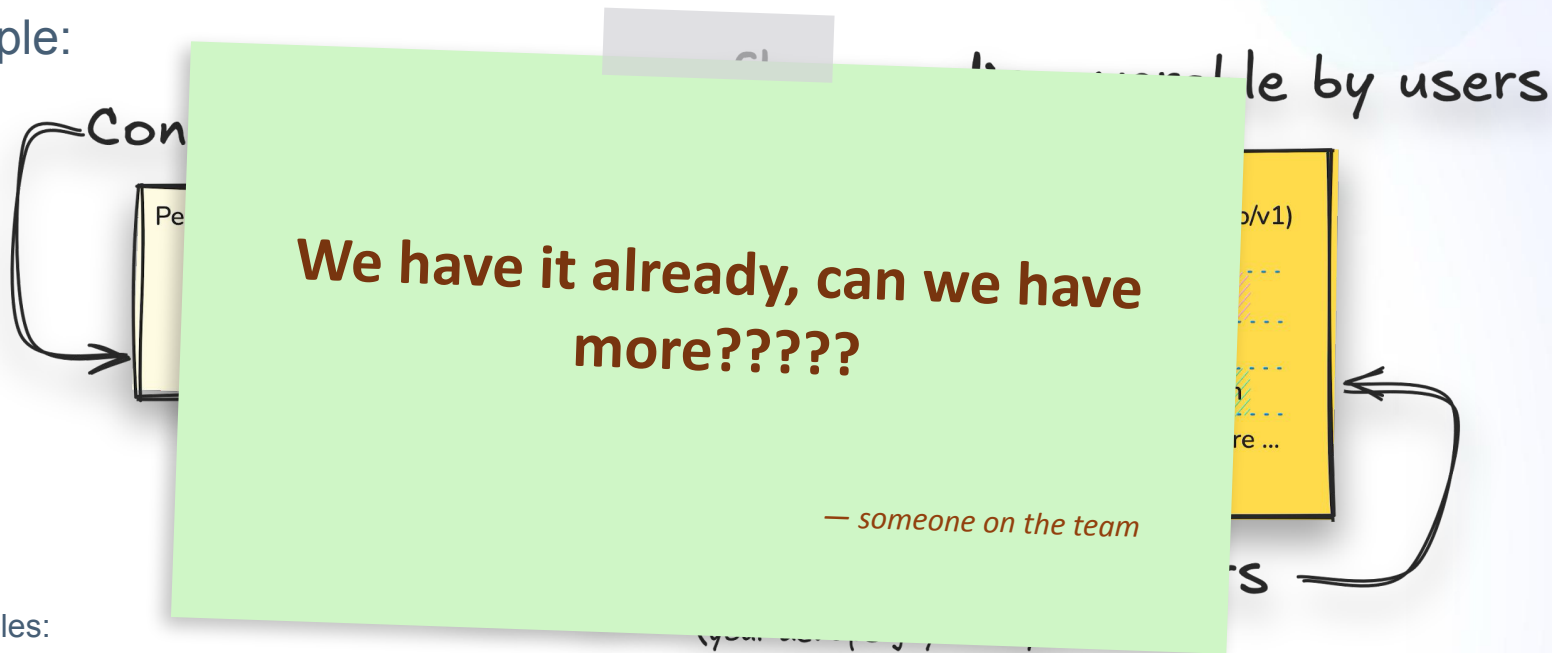
Other examples:

- GatewayClass (gateway.networking.k8s.io/v1)
- RuntimeClass (node.k8s.io/v1)
- You can create your own with CRDs...

X implementations : M provider : N consumers

Well-established concept in vanilla Kubernetes!

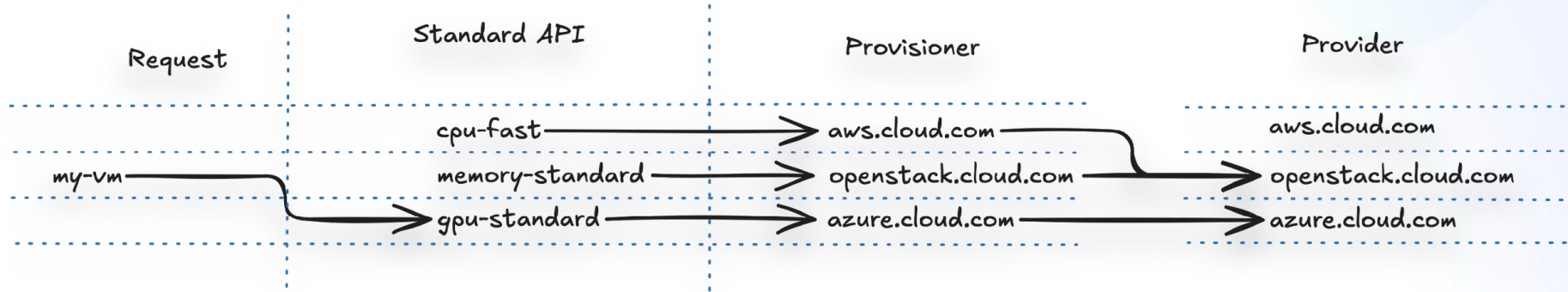
For example:



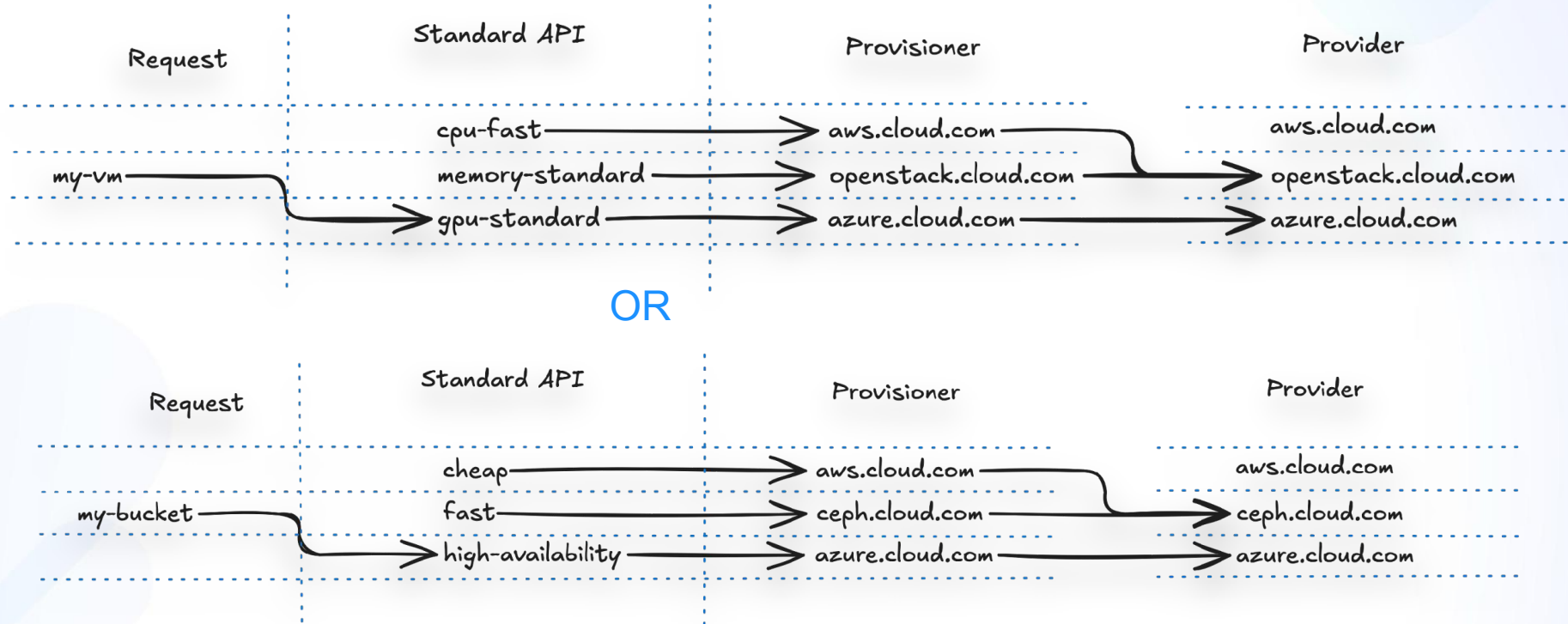
Other examples:

- GatewayClass (gateway.networking.k8s.io/v1)
- RuntimeClass (node.k8s.io/v1)
- You can create your own with CRDs...

X implementations : M provider : N consumers



X implementations : M provider : N consumers





```
$ reality-check.sh
```

```
WARNING: One can build this today already.
```

```
ERROR: But oh boy, it's "break a leg" effort.
```

```
WARNING: And it's super brittle, heavy yaml based.
```

```
FATAL: Add AI hallucinations on top and you have  
full-stack-yaml-based-engineering team  
maintaining this.
```

```
$ █
```

high-availability

azure.cloud.com

azure.cloud.com

Service provider levels - Proposed

Level	Name	How State Moves	Who "Repoints" it?
Level 1	Re-Install	Start from zero. Manual: You build a new server.	Consumer owned migration
Level 2	Data Sync	Restore from Backup. Manual: You update DNS/IPs/Provider	Consumer owned migration with System help
Level 3	Live Migration	Memory/Disk mirroring.	Orchestrator: The Hypervisor handles the handoff.
Level 4	Platform Abstraction	Global State Mesh.	The System: It moves itself based on cost/health

Service provider levels - Proposed

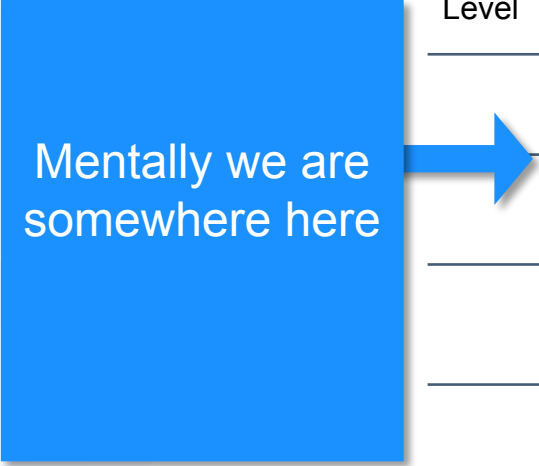
To achieve this,
we need to talk same
language on the
same platforms



Level	Name	How State Moves	Who "Repoints" it?
Level 1	Re-Install	Start from zero. Manual: You build a new server.	Start from zero. Manual: You build a new server.
Level 2	Data Sync	Restore from Backup. Manual: You update DNS/IPs/Provider	Restore from Backup. Manual: You update DNS/IPs/Provider
Level 3	Live Migration	Memory/Disk mirroring.	Orchestrator: The Hypervisor handles the handoff.
Level 4	Platform Abstraction	Global State Mesh	The System: It moves itself based on cost/health

Service provider levels - Proposed

Mentally we are
somewhere here



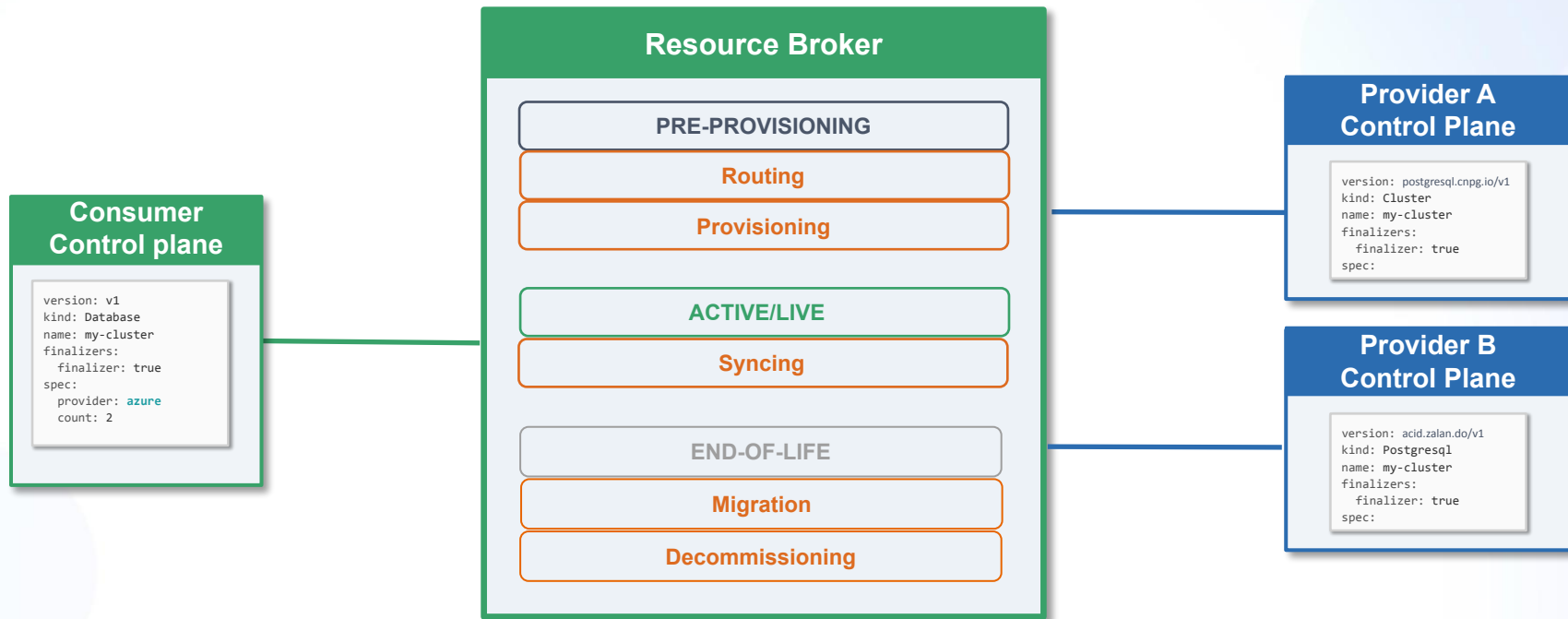
Level	Name	How State Moves	Who "Repoints" it?
Level 1	Re-Install	Start from zero. Manual: You build a new server.	Start from zero. Manual: You build a new server.
Level 2	Data Sync	Restore from Backup. Manual: You update DNS/IPs/Provider	Restore from Backup. Manual: You update DNS/IPs/Provider
Level 3	Live Migration	Memory/Disk mirroring.	Orchestrator: The Hypervisor handles the handoff.
Level 4	Platform Abstraction	Global State Mesh	The System: It moves itself based on cost/health

Service provider levels - Proposed

Mentally we are
somewhere here

Level	Name	How State Moves	Who "Repoints" it?
Level 1	Re-Install	Start from zero. Manual: You build a new server.	Start from zero. Manual: You build a new server.
Level 2		Restore from Backup. Manual: You update DNS/IPs/Provider	Restore from Backup. Manual: You update DNS/IPs/Provider
Level 3		Copy/Disk mirroring.	Orchestrator: The Hypervisor handles the handoff.
Level 4		State Mesh	The System: It moves itself based on cost/health

Resource Broker — intelligent dispatch & migration



KRM-native controller



Old service stays alive
until new is healthy

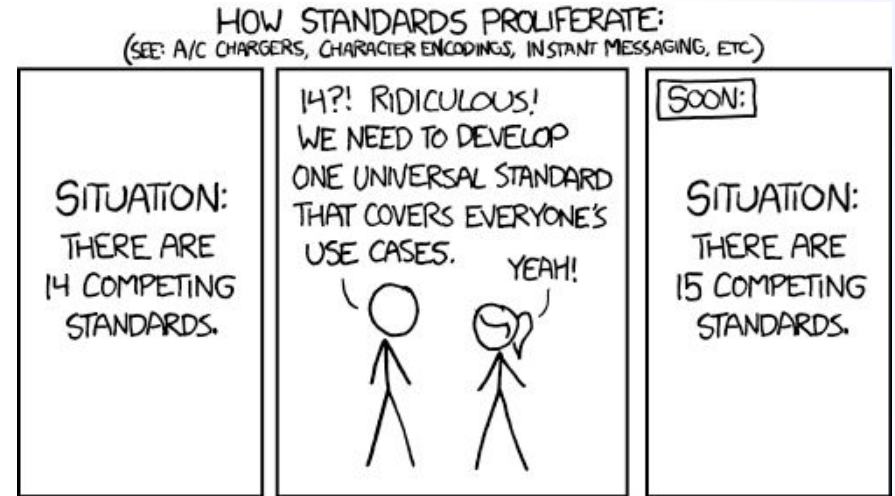


Pluggable migration
strategies/hooks per provider

No new standard - platform gives you tools, not opinions for the standard.

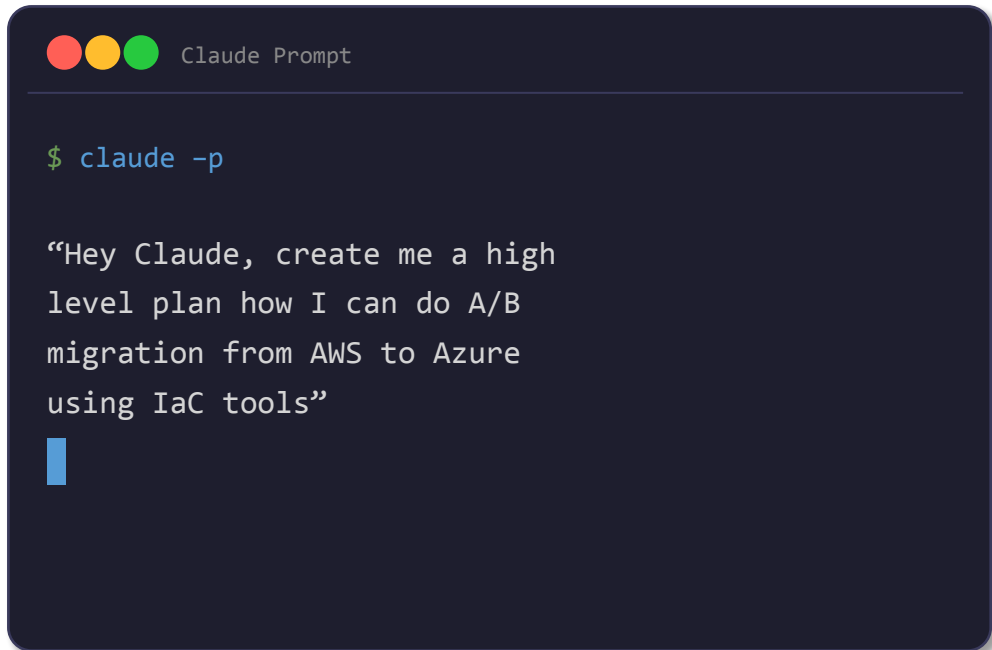
Platform gives you tools to create your own ways of working

When we get across the platforms - we cross this bridge when we get to it.



How would IaC/DevOps of the world would do that now?

GOAL : To migrate cluster with its state from AWS to Azure using existing IaC tools.



```
Claude Prompt

$ claude -p

“Hey Claude, create me a high
level plan how I can do A/B
migration from AWS to Azure
using IaC tools”
█
```

How would IaC/DevOps of the world do this today?

Tool-Agnostic Blue/Green Provider Swap

PREPARE

1

Audit current environment

Document dependencies

Define target architecture

Create IaC for both providers

BOTH ENVIRONMENTS LIVE

TF: workspaces | CP: compositions

DEPLOY GREEN

2

Provision target infra

Deploy workloads

Verify health checks

BOTH ENVIRONMENTS LIVE

TF: apply | CP: kubectl apply claim

MIGRATE & SHIFT

3

Sync persistent data

Gradual traffic shift

Validate & monitor

ROLLBACK IF NEEDED

DNS weights / Ingress / GLB

CLEANUP

4

100% traffic on Green

Decommission Blue

Update documentation

MIGRATION COMPLETE

TF: destroy | CP: delete claim

KEY PRINCIPLES

Never destroy Blue
before Green is proven

Data sync happens
WHILE both are live

Traffic shift is
gradual & reversible

Rollback = shift
traffic back to Blue

Tool choice affects HOW
not WHAT you do

How would IaC/DevOps of the world do this today?

Tool-Agnostic Blue/Green Provider Swap

PREPARE

DEPLOY GREEN

CLEANUP

4

**1+ month project
at minimum**

With AI or without...

& reason for DevOps presence

Audit current environment

Document dependencies

Define target architecture

Create IaC for both providers

BOTH ENVIRONMENTS

TF: workspaces | CP: compositions

Green

Blue

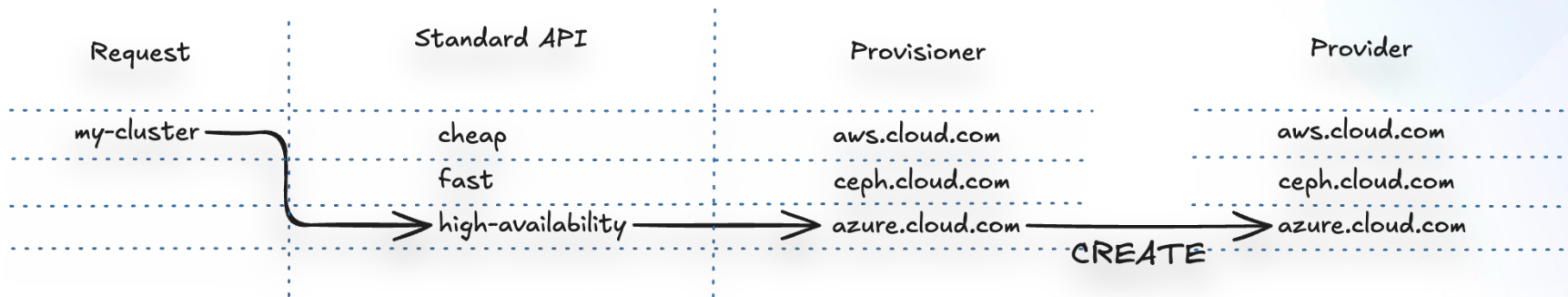
Validation

OPERATION COMPLETE

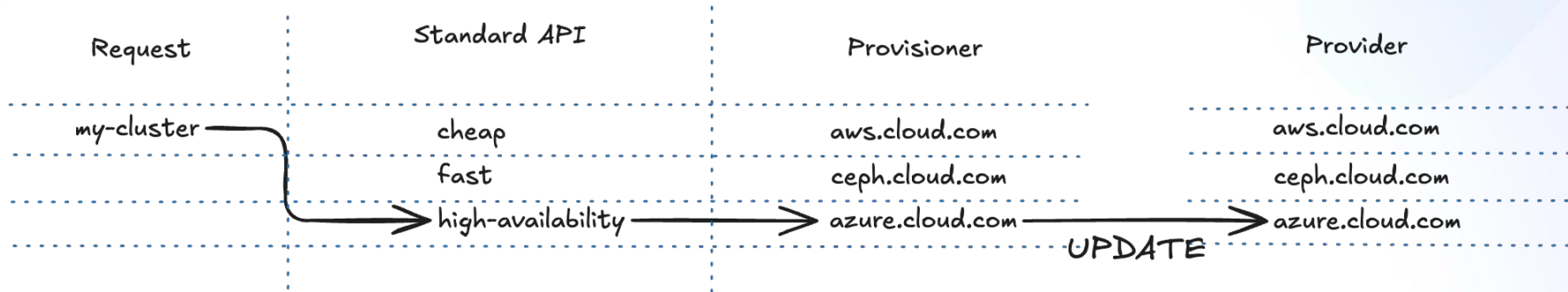
delete claim

**Never destroy Blue
before Green is proven**

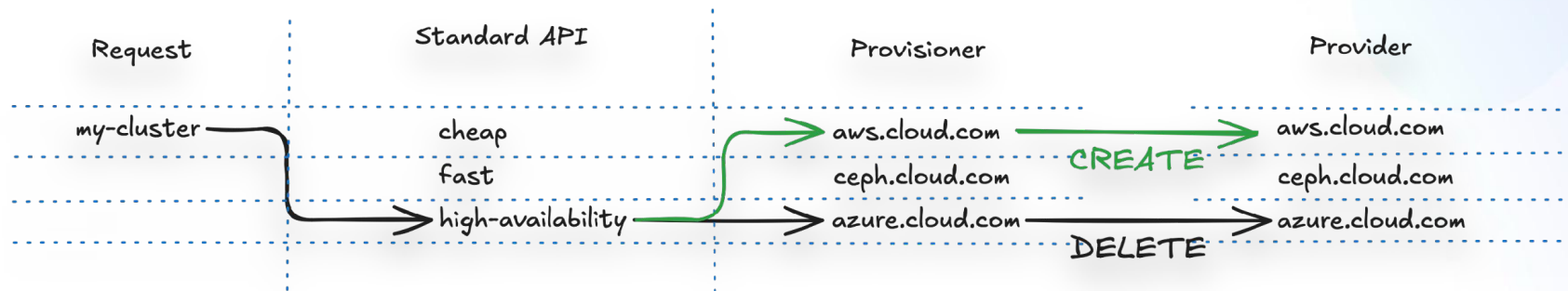
**Tool choice affects HOW
not WHAT you do**



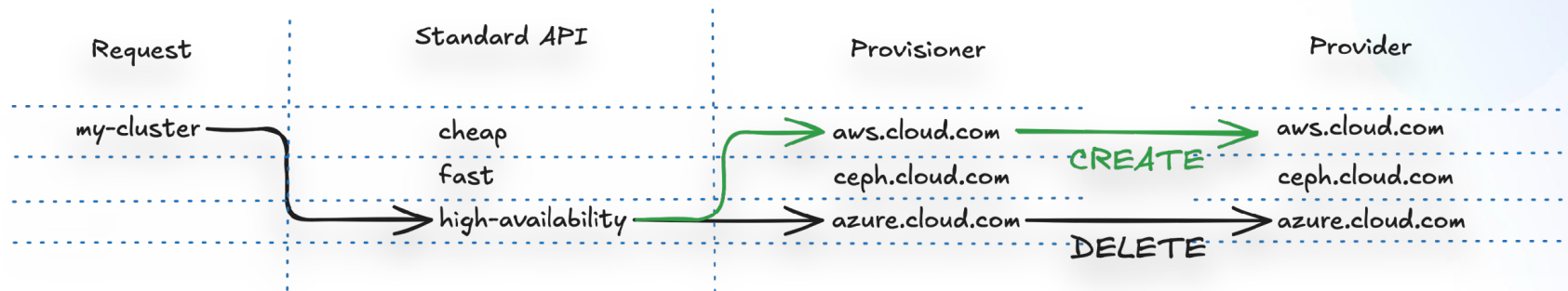
```
kind: Cluster
name: my-cluster
finalizers:
  finalizer: true
spec:
  provider: azure
  count: 2
```



```
kind: Cluster
name: my-cluster
finalizers:
  finalizer: true
spec:
  provider: azure
  count: 2 -> 3
```

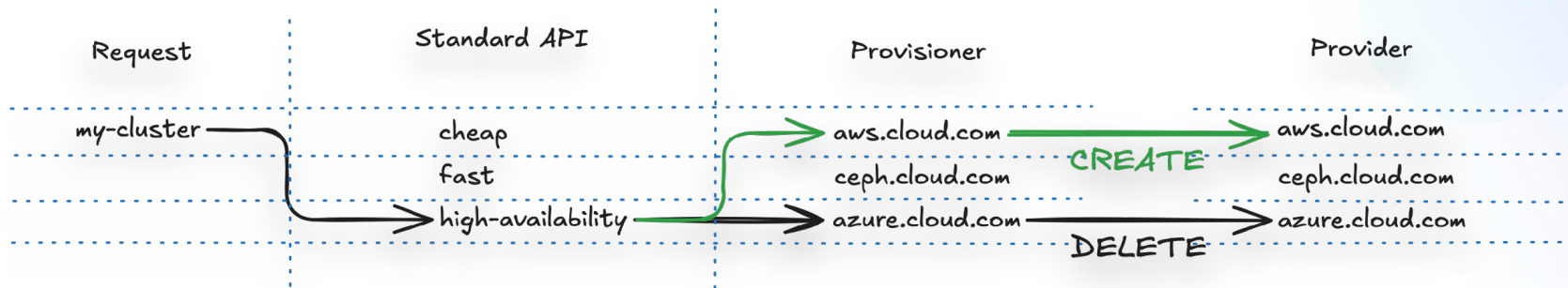


```
kind: Cluster
name: my-cluster
finalizers:
  finalizer: true
spec:
  provider: azure -> aws
  count: 3
```



```
kind: Cluster
name: my-cluster
finalizers:
  finalizer: true
spec:
  provider: azure -> aws
  count: 3
```

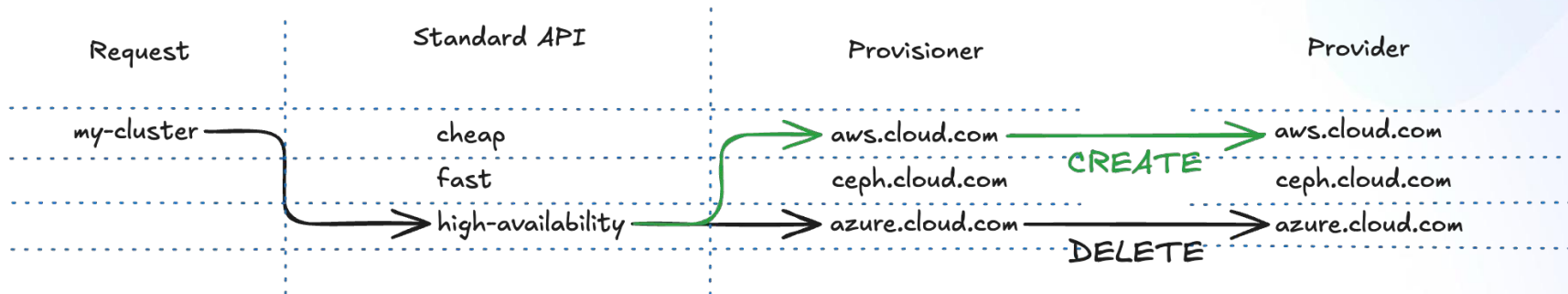
See the problem?



```
kind: Cluster
name: my-cluster
finalizers:
  finalizer: true
spec:
  provider: azure -> aws
  count: 3
```

See the problem?

How to handle finalizer when only one object exists?

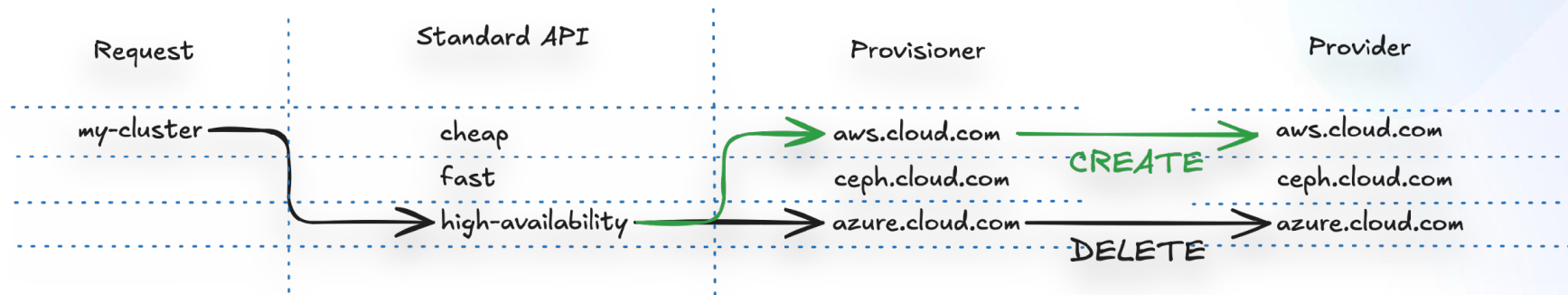


```
kind: Cluster
name: my-cluster
finalizers:
  finalizer: true
spec:
  provider: azure -> aws
  count: 3
```

See the problem?

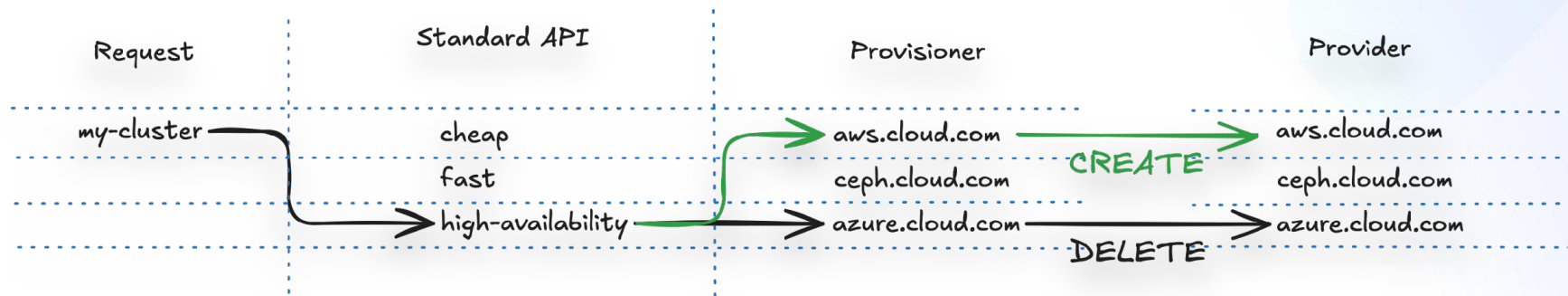
How to handle finalizer when only one object exists?

How to handle delete events?



```
kind: Cluster
name: my-cluster-2
finalizers:
  finalizer: true
spec:
  provider: aws
  count: 3
```

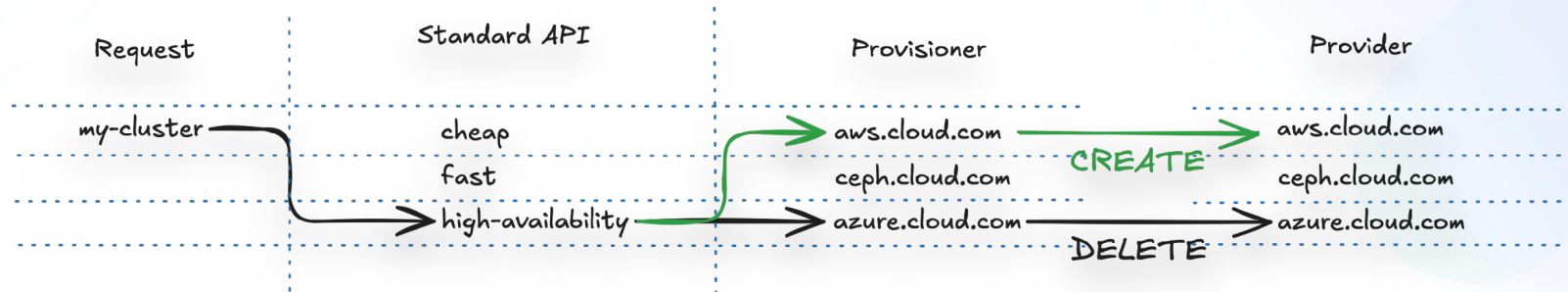
```
kind: Cluster
name: my-cluster-1
finalizers:
  finalizer: true
spec:
  provider: azure
  count: 3
```



```
kind: Cluster
name: my-cluster-2
finalizers:
  finalizer: true
spec:
  provider: aws
  count: 3
```

```
kind: Cluster
name: my-cluster-1
finalizers:
  finalizer: true
spec:
  provider: azure
  count: 3
```

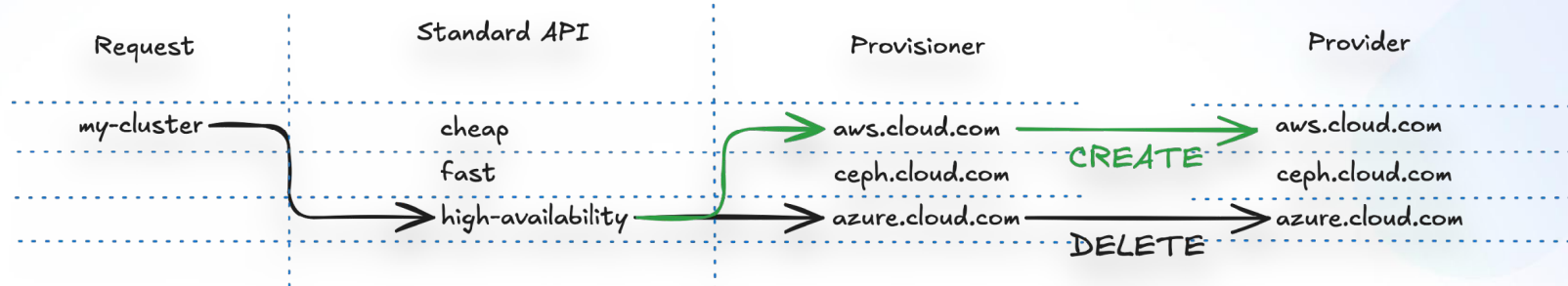
- No difference from what we do now
- UX is poor



```
kind: Cluster
name: my-cluster
finalizers:
  finalizer: true
spec:
  provider: aws
  count: 3
```

```
kind: Cluster
name: my-cluster
finalizers:
  finalizer: true
spec:
  provider: aws
  count: 3
```

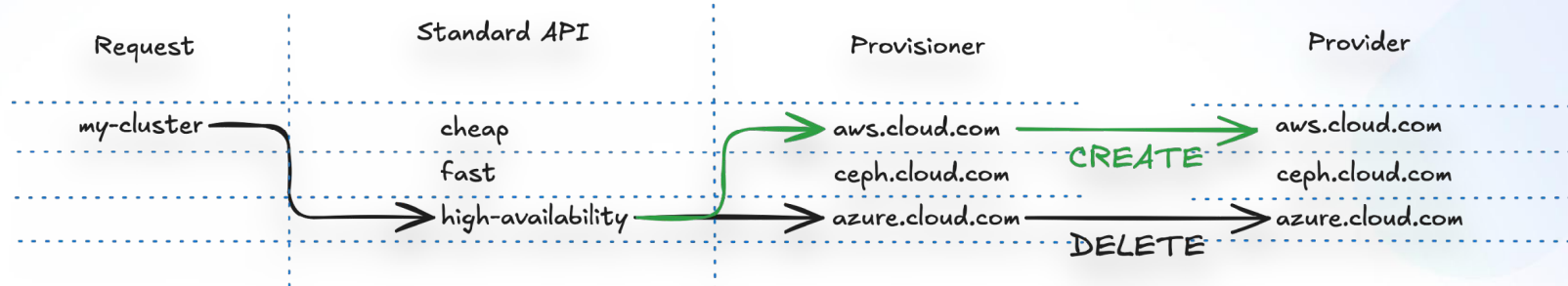
```
kind: Cluster
name: my-cluster
finalizers:
  finalizer: true
spec:
  provider: azure
  count: 3
```



```
kind: Cluster
name: my-cluster
finalizers:
  finalizer: true
spec:
  provider: aws
  count: 3
```

```
kind: Cluster
name: my-cluster
finalizers:
  finalizer: true
spec:
  provider: aws
  count: 3
```

```
kind: Cluster
name: my-cluster
finalizers:
  finalizer: true
spec:
  provider: azure
  count: 3
```

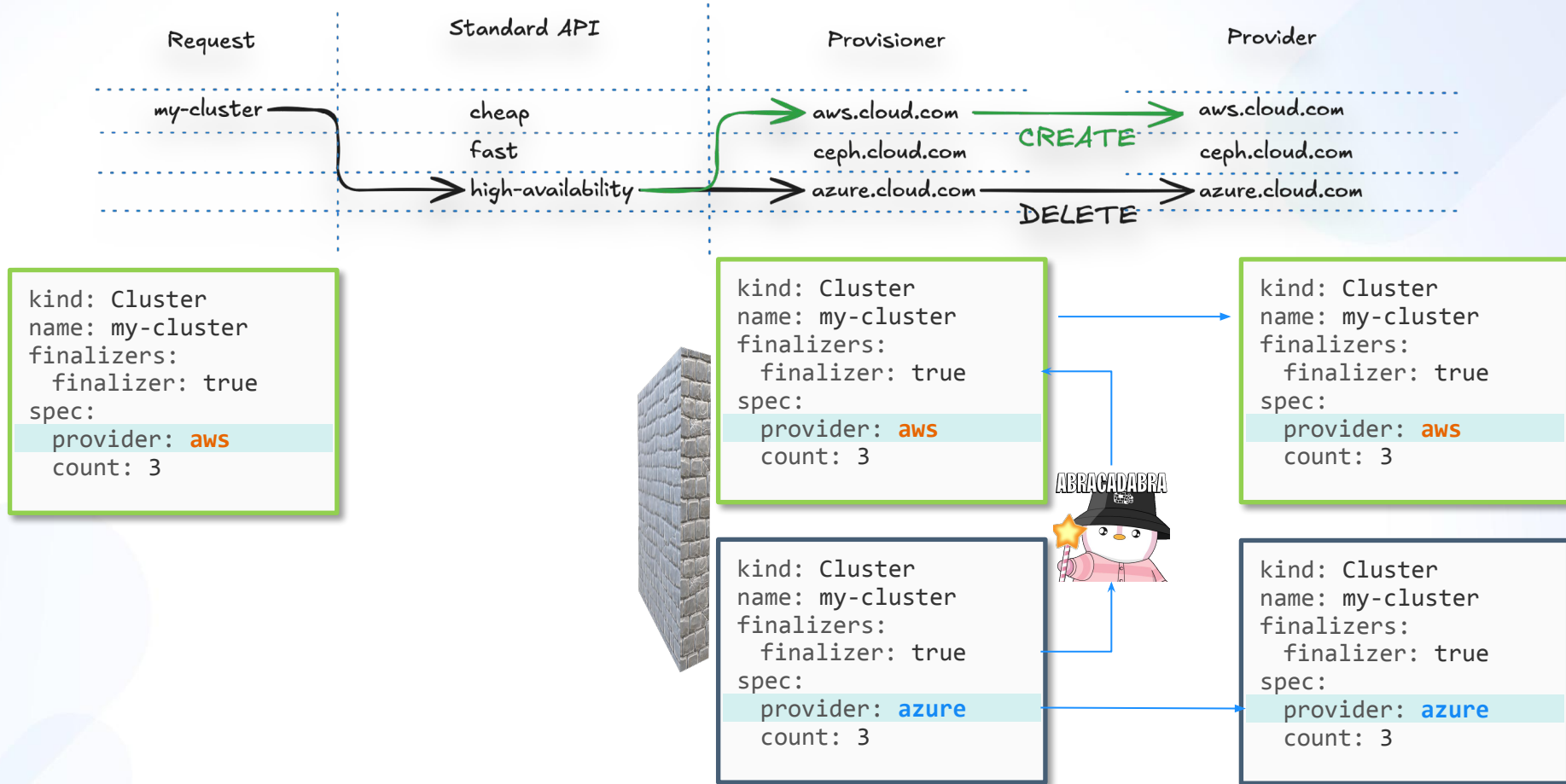


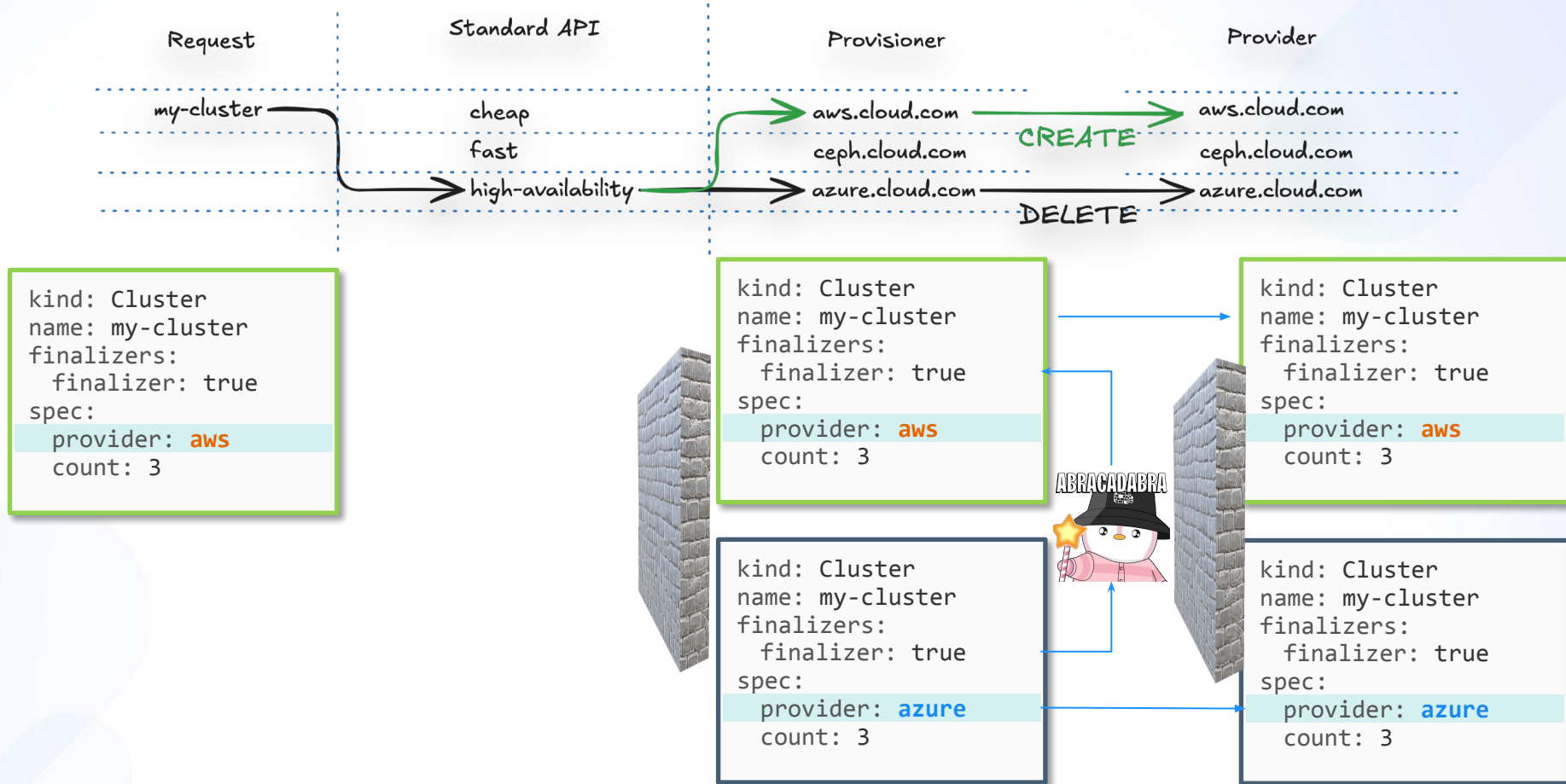
```
kind: Cluster
name: my-cluster
finalizers:
  finalizer: true
spec:
  provider: aws
  count: 3
```

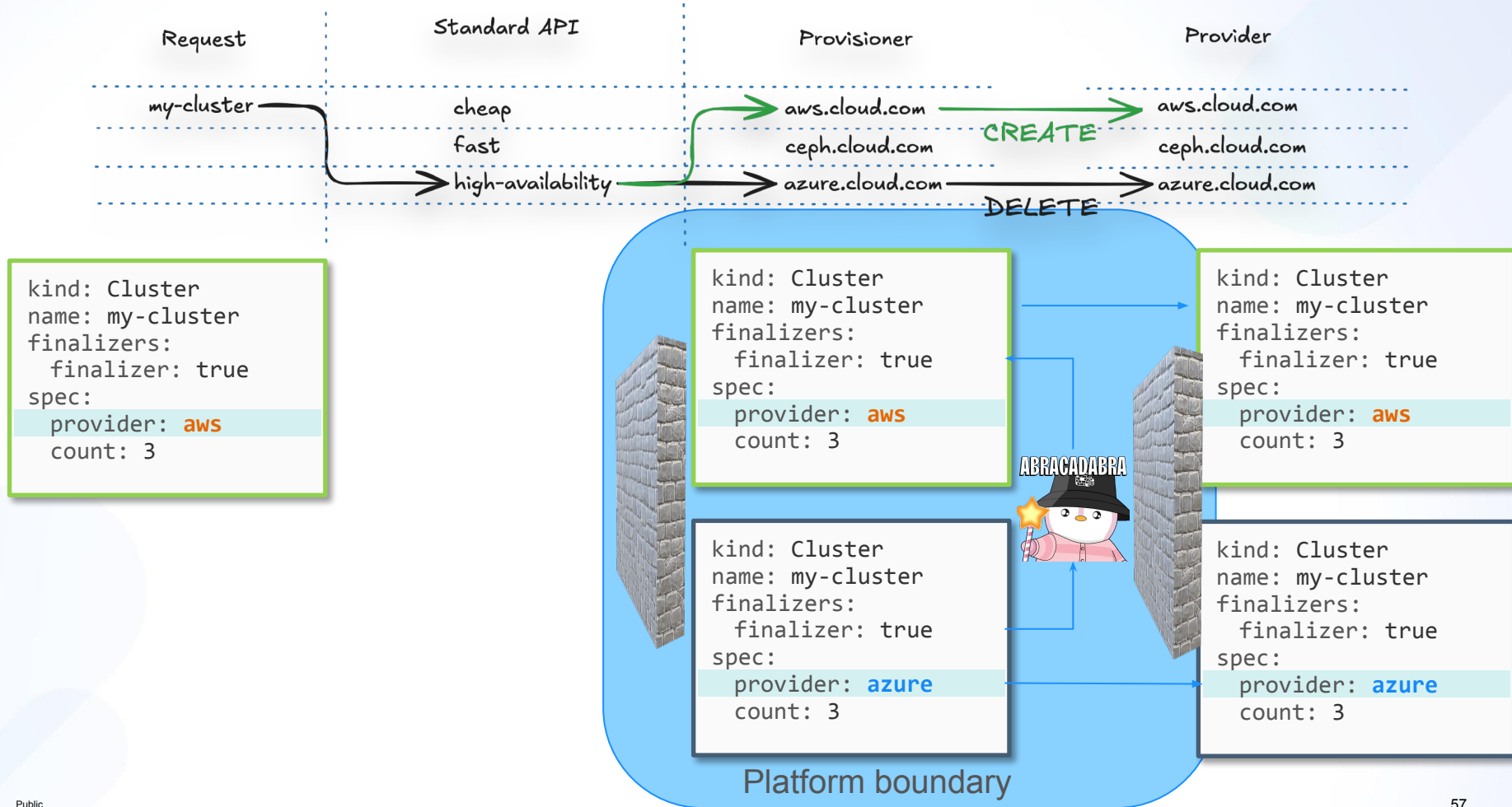
```
kind: Cluster
name: my-cluster
finalizers:
  finalizer: true
spec:
  provider: aws
  count: 3
```

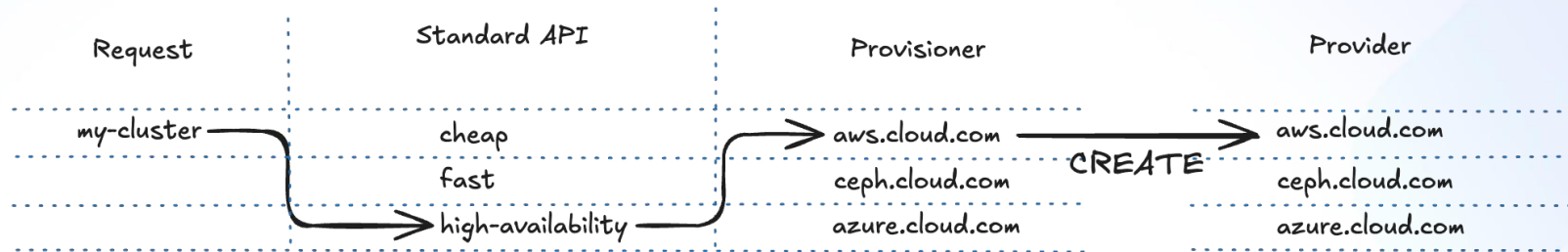
```
kind: Cluster
name: my-cluster
finalizers:
  finalizer: true
spec:
  provider: azure
  count: 3
```











```
kind: Cluster
name: my-cluster
finalizers:
  finalizer: true
spec:
  provider: aws
  count: 3
```

```
kind: Cluster
name: my-cluster
finalizers:
  finalizer: true
spec:
  provider: aws
  count: 3
```

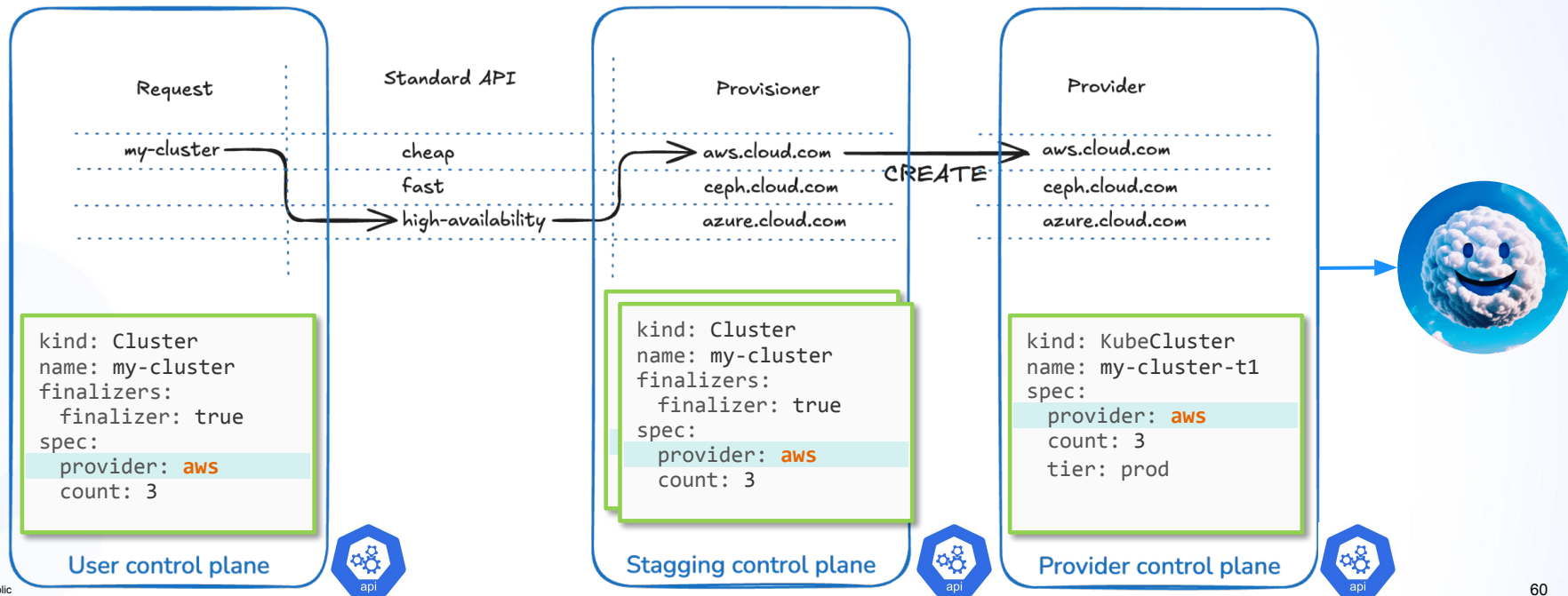
```
kind: Cluster
name: my-cluster
finalizers:
  finalizer: true
spec:
  provider: aws
  count: 3
```

What we learned?

- For full 2(3...) vendors portability - you need to have multiple object copies not to break KRM machinery
- It has to be abstracted from user (Why do I have 3 clusters?)
- It's not easy (else we would already have it everywhere)
- Separation by control planes

What we learned?

- Architecture like this are only possible if you have cheap control-planes at your disposal to be used

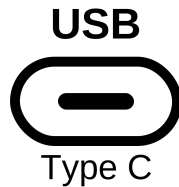


Part 5: Demo

Summary

What are polymorphic APIs?

An abstraction layer for services and resources



Why do we need polymorphic APIs?

m consumer : 1 service : n providers : o flavours

Simplification of operations on both sides of the API

You want to build a cloud :)

How can we implement polymorphic APIs?

kcp as Service Control Plane

KRM as lingua franca



+ KRM =



NœoNephos |  **THE LINUX FOUNDATION** | Europe

kcp.io



 **ApeiRA** | 

platform-mesh.io



**Funded by
the European Union**
NextGenerationEU

Supported by:



Federal Ministry
for Economic Affairs
and Energy

on the basis of a decision
by the German Bundestag

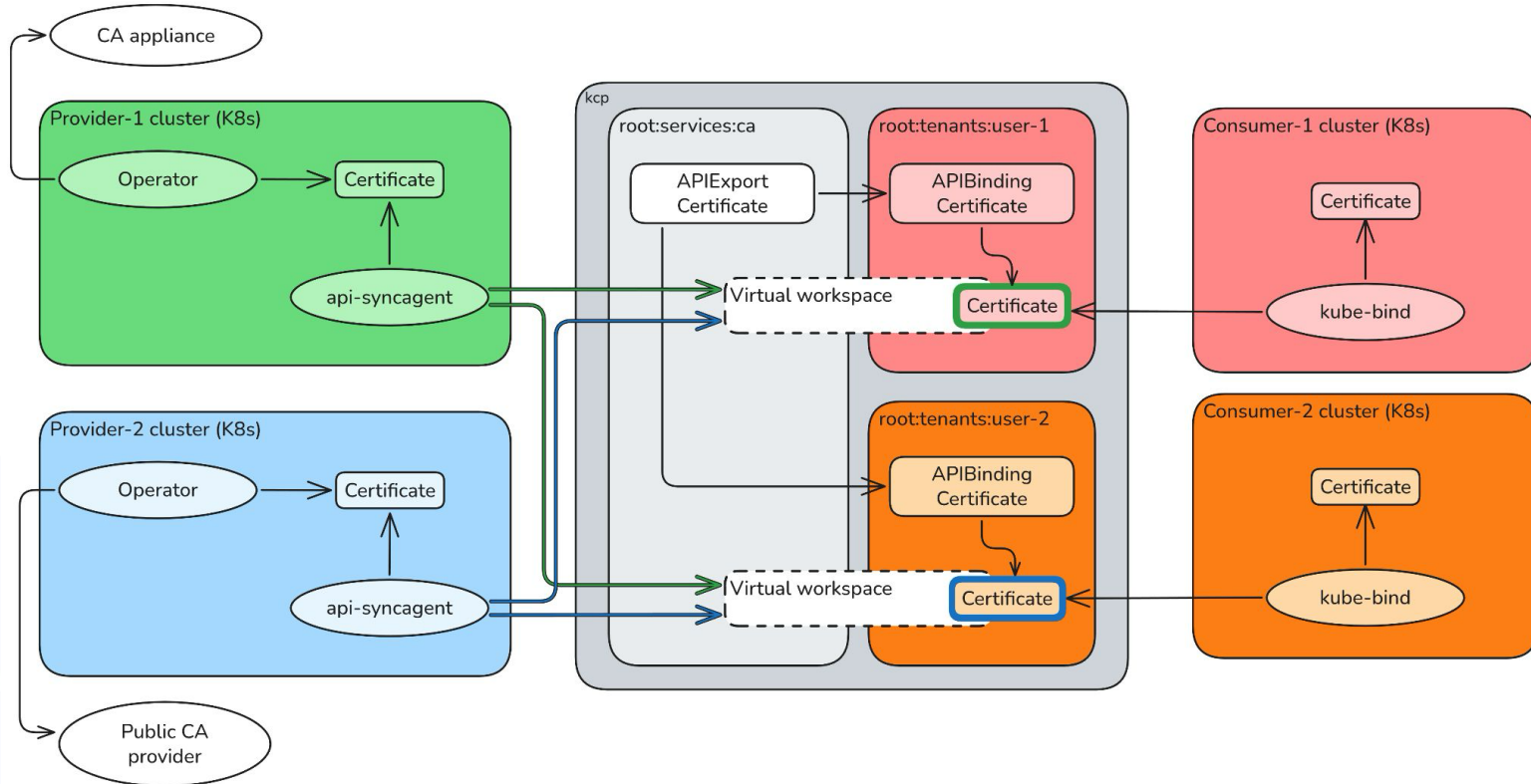
All links also on:
platform-mesh.io/get-involved

Questions

Feedback



M providers : N consumers



Same service. Completely different APIs.

The API Fragmentation Problem

Provider A · Zalando Operator

```
apiVersion: acid.zalan.do/v1
kind: postgresql
metadata:
  name: my-db
spec:
  numberOfInstances: 2
  postgresql:
    version: "15"
  teamId: myteam
  volume:
    size: 10Gi
  users:
    appuser:
      - superuser
```

Provider B · CloudNativePG

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: my-db
spec:
  instances: 2
  imageName:
    postgresql:15
  storage:
    size: 10Gi
  bootstrap:
    initdb:
      database: app
```

Platform Mesh · Generic API

```
apiVersion: services.platform-
kind: PostgresDatabase
metadata:
  name: my-db
spec:
  instances: 2
  version: "15"
  storage: 10Gi
  # Provider selected
  # by policy – not
  # by the consumer
```

Consumer writes **one manifest**. Platform chooses the provider. Migration = **change a policy**, not rewrite YAML.