Crossplane

Introduction & Deep Dive

https://crossplane.io

Ying Mo, IBM

What is Crossplane?

- **Framework** for building cloud native **control planes**
  - No need to write any code
- Cloud providers have been managing their infrastructure with control planes for years
  - Crossplane helps you build your own - with your own **opinions**
- Extensible backend to manage **any infrastructure** in **any environment**
- Configurable frontend to expose **declarative APIs** (abstractions) for developer **self-service**

CNCF Project for the Community

- Crossplane is a neutral place for vendors and individuals to come together in enabling control planes
- Launched in Dec 2018 by creators of CNCF graduated Rook project
  - Accepted into Sandbox in June 2020
  - First major “stable” milestone v1.0 released in Dec 2020
  - Moved to Incubation September 2021
  - v1.13 most recent release, v1.14 targeting late Oct
  - Progressing towards Graduation - we need your help adopters!

Project and Community Stats

9,000+
Stars

80,000+
Contributions

1,650+
Contributors

6,950+
Followers

9,500+
Members

42M+
Pulls

The Basics

Managed Resources

Crossplane

Managed Resources Example: AWS

Networking

Databases

Kubernetes Clusters

IAM

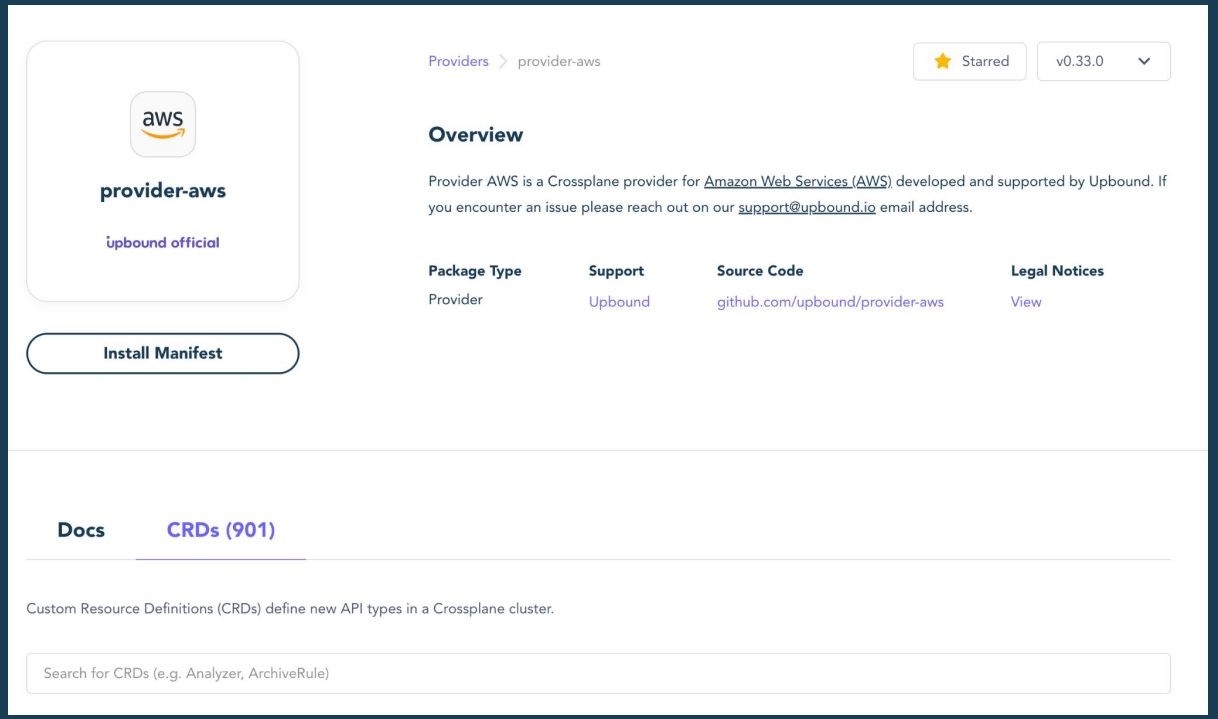
VMs

Message Queues

Caches

Certificates

...and much more...

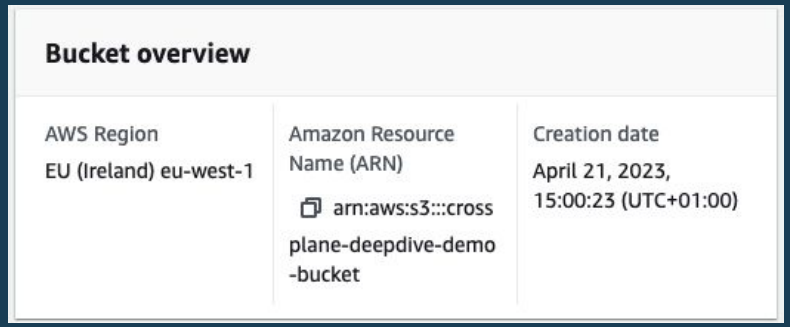


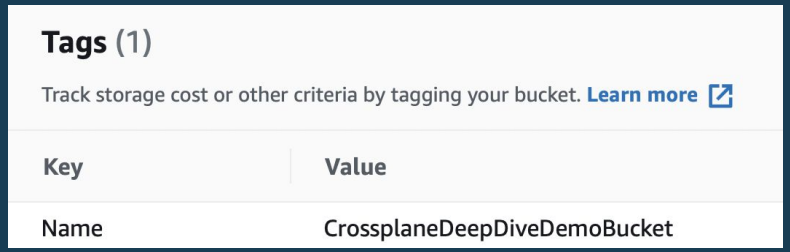
Crossplane

https://marketplace.upbound.io/providers/upbound/provider-family-aws

Managed Resources

```yaml
apiVersion: s3.aws.crossplane.io/v1beta1
kind: Bucket
metadata:
  name: crossplane-deepdive-demo-bucket
spec:
  forProvider:
    acl: private
    locationConstraint: eu-west-1
    paymentConfiguration:
      payer: BucketOwner
    versioningConfiguration:
      status: Enabled
    tagging:
      tagSet:
      - key: Name
        value: CrossplaneDeepDiveDemoBucket
```





Managed Resources

Status contains values returned from the remote API and the condition of the resources.

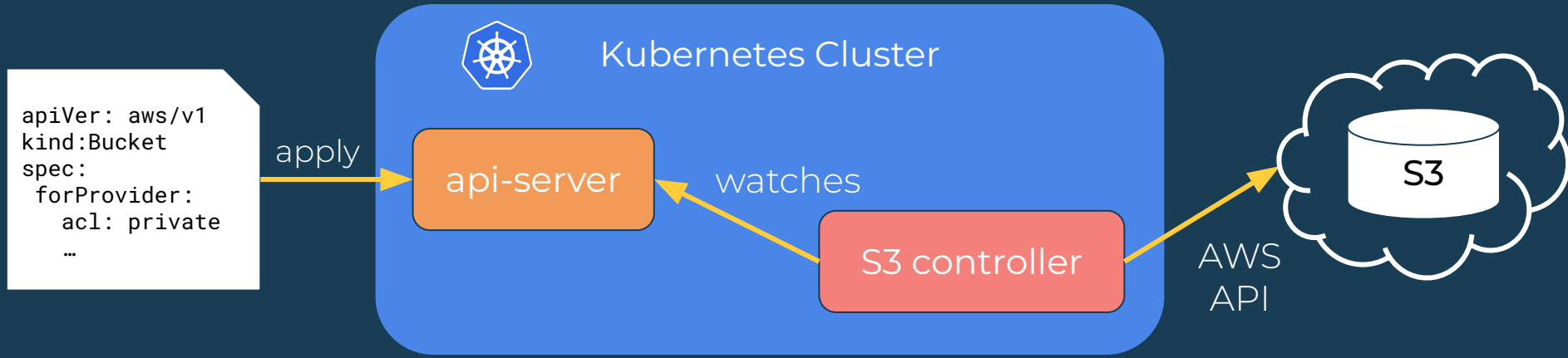
```
Status:
  At Provider:
    Arn:  arn:aws:s3:::crossplane-deepdive-demo-bucket

Events:
  Type    Age   From                          Message
  ----    ----  ----                          -------
  Normal  6m8s  bucket.s3.aws.crossplane.io  Successfully created external resource
```

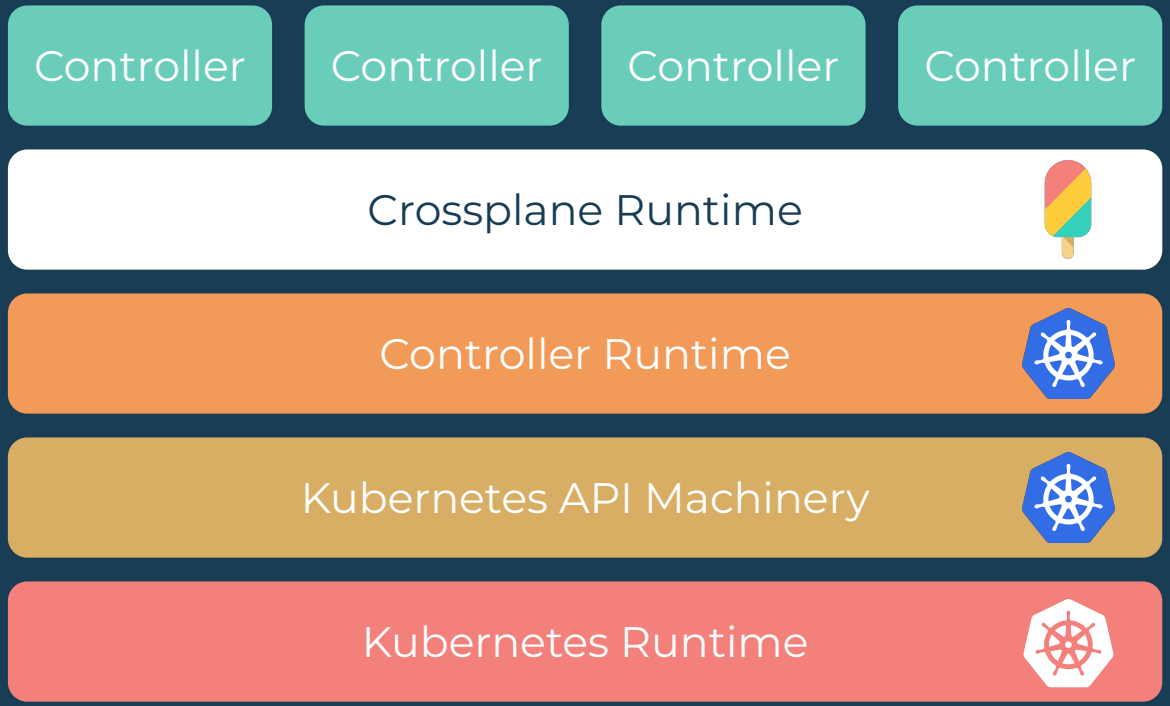
Managed Resources Generate K8s Events

Managed Resource Reconciliation

- Controllers reconcile these CRDs with cloud provider and on-prem APIs (e.g., GCP, AWS, or any API really)



Control Plane Internal Stack



Custom Logic

Manage External APIs
Create/Update/Delete

Event, Watch, Request,
Reconciliation

CRDs, OpenAPI,
Persistence (etcd)

Run Workloads, Ingress,
RBAC

Building Your Control Plane

Composition

Crossplane

Build your own Platform API

- Assemble granular resources. E.g. from multiple clouds.
- Expose as higher level self-service API for your app teams
  - **Compose** GKE, NodePool, Network, Subnetwork
  - **Offer** as a single Cluster resource (API) with limited config for developers to self-service
- Hide infrastructure complexity and include policy guardrails
- All with K8s API - compatible with kubectl, GitOps, etc.
- **No code** required, it's all **declarative**

Claim

XRD
Composite Resource Definition

Composition

Configuration

Managed Resource

Managed Resource

Managed Resource

Provider

Crossplane

Small PostgreSQL

XRD
Composite Resource Definition

AWS Composition

Configuration

RDS Instance

DBParameterGroup

SecurityGroup

Provider

Crossplane

Composite Resources

```yaml
apiVersion: apiextensions.crossplane.io/v1
kind: CompositeResourceDefinition
metadata:
  name: nosqls.database.example.com
spec:
  group: database.example.com
  names:
    kind: NoSQL
    plural: nosqls
  versions:
  - name: v1alpha1
    served: true
    referenceable: true
    schema:
      openAPIV3Schema:
        type: object
        properties:
```

First create Composite Resource Definition (XRD) to declare our custom platform API

Custom API Group

Standard openAPIV3 Schema

Compositions

```yaml
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  name: dynamo-with-bucket
spec:
  compositeTypeRef:
    apiVersion: database.example.com/v1alpha1
    kind: NoSQL
  resources:
    - name: dynamoDB
      base:
        apiVersion: dynamodb.aws.upbound.io/v1beta1
        kind: Table
```

Then we define a Composition which implements XRD

XRD reference

List of Managed Resources to Compose

Patches

Patches enable propagation of data from Composite Resource (XR) down to composed Managed Resources (MR)

```yaml
patches:
  - type: FromCompositeFieldPath
    fromFieldPath: "spec.readCapacity"
    toFieldPath: "spec.forProvider.readCapacity"
  - type: FromCompositeFieldPath
    fromFieldPath: "spec.location"
    toFieldPath: "spec.forProvider.region"
    transforms:
      - type: map
        map:
          EU: "eu-north-1"
          US: "us-east-2"
```

Copy of value from XR spec down to MR spec

Map transform to manipulate the config data

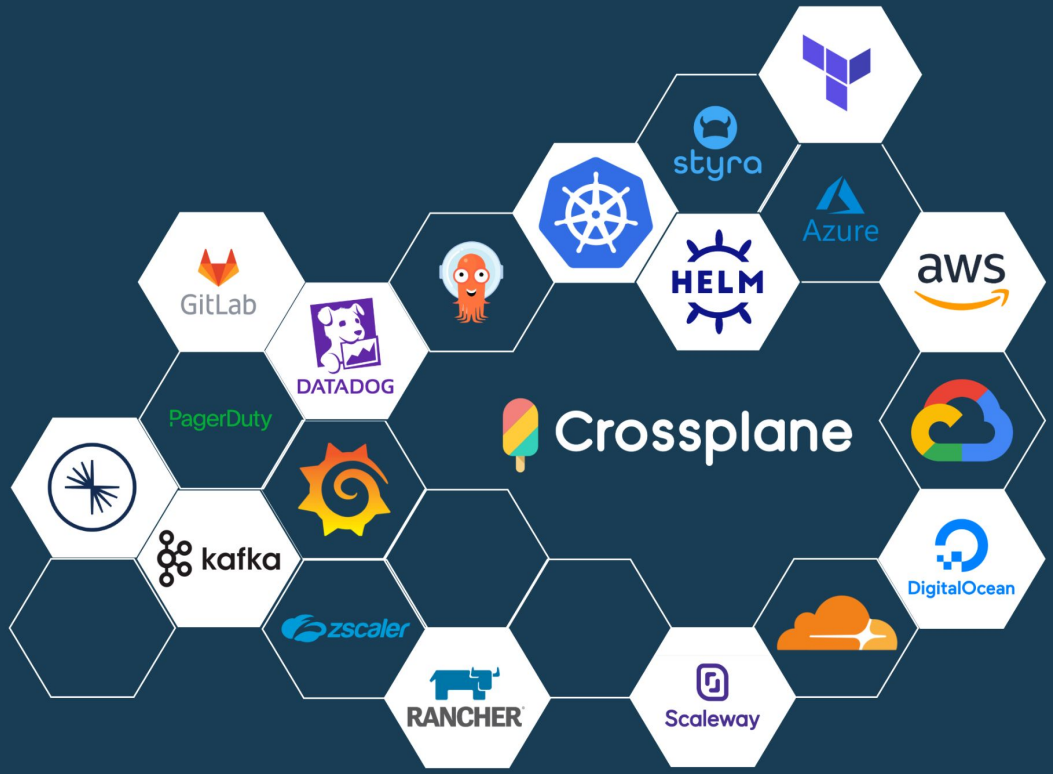
Extending Crossplane

Providers & Configurations

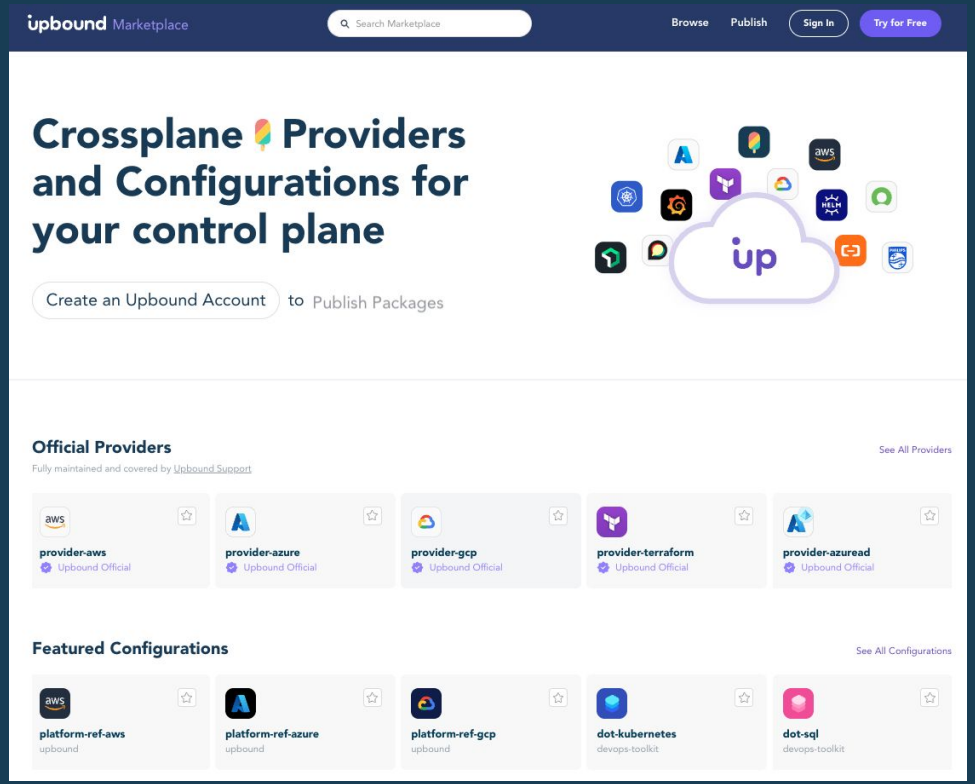
Current Extension Points

- Crossplane is a highly extensible framework
- **Providers**
  - You can build a provider to manage **anything** with an API
  - CRUD operations for cloud resources, on-prem services, etc.
- **Configurations**
  - Compose resources from providers
  - Define your control plane's declarative APIs and abstractions
  - These are what your devs see - it's how they consume the offerings of your control plane
- Both are Crossplane packages / opinionated OCI Images.

Crossplane Provider Ecosystem



Marketplace for all Extensions



Discover and share Crossplane extensions

Open to everyone

https://marketplace.upbound.io

Composition Functions

Current Limitations of Composition

- No iteration. No conditionals. No templates.
- No advanced logic other than simple patch & transforms
- List of resources is static
- No ability to call external APIs to get values
- and others...composition is not a programming language
- We didn't want to grow a DSL expressed in YAML
  - Need to reinvent a lot of wheels - testing, linting, etc
  - Infra DSLs tend to grow organically, but not cohesively
  - We're not language designers

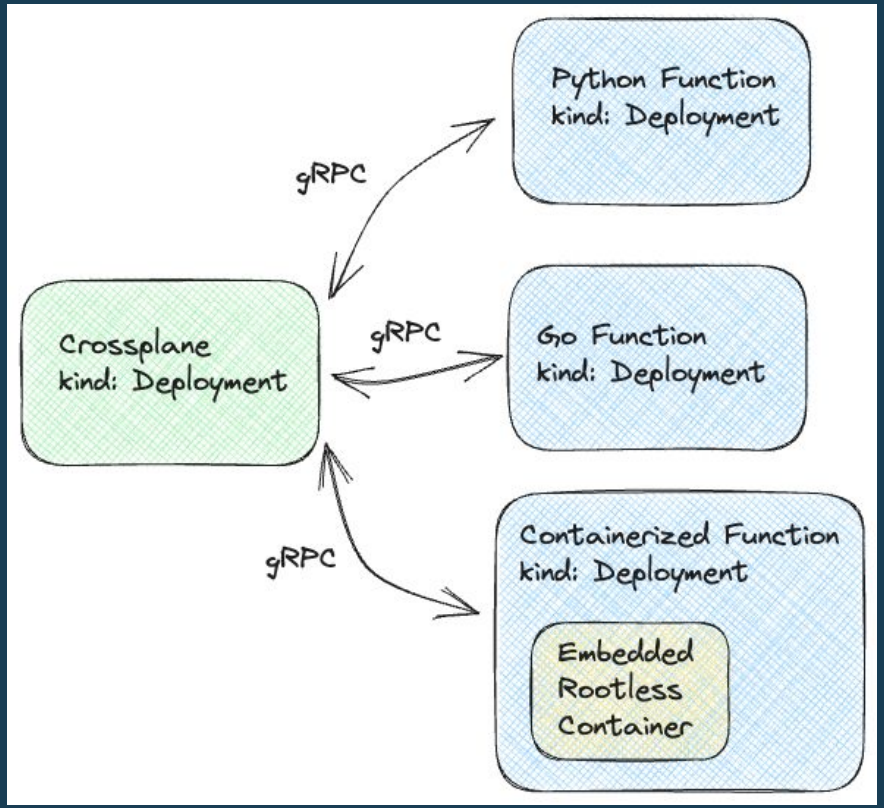
What can Functions do?

- First released as **alpha** in **v1.11.0**
- Evolving the architecture and experience while maturing to **beta** in **v1.14.0**
- Run a pipeline of simple functions
- Written in your language of choice with any logic your use case needs
- You don't **have** to write any code to start using functions
  - Reusable functions that are generally useful
  - e.g., helm/go templates
- Sweet spot between "no code" and building an entire controller
  - Focus on your platform's unique needs - Crossplane still does the heavy lifting of CRUD-ing resources, finalizers, owner refs, etc

How do Functions work?

- Packaged/distributed just like `Providers` and `Configurations`
- Each function is more like a “function server”
  - Long running processes, created as a `Deployment` and headless `Service`
- Crossplane talks gRPC to each function
  - `RunFunctionRequest`, `RunFunctionResponse`
  - secured by mutual transport layer security (mTLS)
- Pipeline of data passing from one function to the next, each modifying and validating the stream as needed
- Functions don’t interact with API server - Crossplane will take resulting set of resources and “make it so”

Functions Visualized



Building Functions

- SDKs
  - Libraries for common tasks, codify best practices, eliminate boilerplate
- Tooling
  - Scaffold a new Function with everything except your custom logic
    - `kubectl crossplane init function --template "crossplane/function-template-go"`
  - Build/Push to package registry
    - `kubectl crossplane function build`
    - `kubectl crossplane function push myorg/cool-func:v0.1.0`
  - Test and iterate locally
    - `kubectl crossplane composition render composition.yaml xr.yaml`

Using Functions

- Not everyone needs to write their own Functions code
  - There will be generic & reusable Functions for the 80% case
- General workflow
  - Find (or build) Functions you want to use
  - Install the Functions into your control plane with `kind: Function`
  - Reference the Functions in a `Composition`
  - Provide Function input in the `Composition`, where needed
- Mixing classic Patch & Transform logic alongside Functions is possible

Using Functions Example - `for` loop

```yaml
apiVersion: apiextensions.crossplane.io/v1beta1
kind: Composition
metadata:
  name: example
spec:
  compositeTypeRef:
    apiVersion: database.example.org/v1
    kind: XPostgreSQLInstance
  mode: Pipeline
  pipeline:
  - step: compose-xr-using-go-templates
    functionRef:
      name: go-templates
    input:
      apiVersion: example.org/v1
      kind: GoTemplate
      source: Inline
      inline: |
        {{- range $i := until ( .desired.composite.resource.spec.count ) }}
        ---
        apiVersion:  rds.aws.upbound.io/v1beta1
        kind: Instance
        spec:
          forProvider:
            engine: postgres
            engineVersion: "13.7"
            ...etc...
        {{- end }}
  - step: validate-composed-resources
    functionRef:
      name: cel-validation
```

Ordered Deletion

Deletion ordering problem

- Kubernetes is **eventually consistent**
  - Works great when **creating** multiple resources that have dependencies, e.g. `VPC` and `Subnet`
  - Just keep retrying until dependencies are created…success!
  - Loose coupling, less complexity, resilient
- Eventual consistency doesn't always work for **deletions**
  - Can result in orphaned managed resources
  - e.g., Helm `Release` deployed into EKS `Cluster`
  - Deleting `Cluster` first prevents proper clean-up of `Release` and all its resources

`Usage` API

- New `Usage` type being introduced in Crossplane v1.14
  - Alpha level for at least one release to get feedback and iterate
- Declare dependency relationships between Crossplane resources
- Relationships captured in a `Usage` object are enforced by an admission webhook
  - `Usage` of A by B will block deletion of A until B is deleted first
  - e.g., `admission webhook "nousages" denied the request: This resource is in-use by Usage Release/my-chart`

Example Usage Dependency

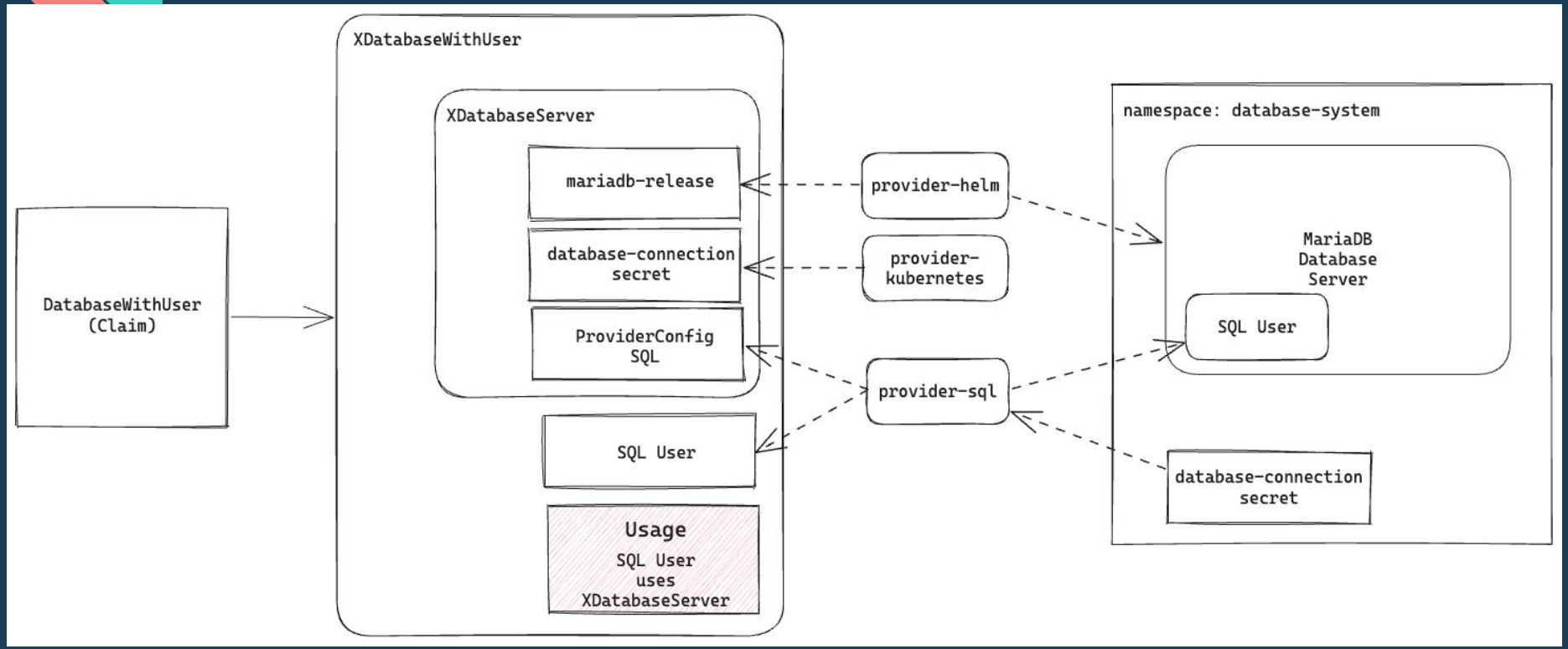
```yaml
apiVersion: apiextensions.crossplane.io/v1alpha1
kind: Usage
metadata:
  name: release-uses-cluster
spec:
  reason: "Release uses Cluster"
  of:
    apiVersion: eks.upbound.io/v1beta1
    kind: Cluster
    resourceRef:
      name: my-cluster
  by:
    apiVersion: helm.crossplane.io/v1beta1
    kind: Release
    resourceRef:
      name: my-prometheus-chart
```

Resource Protection with Usage

- Protect resources forever by omitting by field

```yaml
apiVersion: apiextensions.crossplane.io/v1alpha1
kind: Usage
spec:
  reason: "Production Database - never delete"
  of:
    apiVersion: rds.aws.upbound.io/v1beta1
    kind: Instance
    resourceRef:
      name: my-cluster
```

Demo Usage - SQLUser depends on DBServer



Crossplane

https://github.com/jbw976/usage-demo

Demo
Usage

https://github.com/jbw976/usage-demo

Community is everything

Get Involved

- Website: https://crossplane.io/
- Docs: https://crossplane.io/docs
- GitHub: https://github.com/crossplane/crossplane
- Slack: https://slack.crossplane.io/
- Blog: https://blog.crossplane.io/
- Twitter: https://twitter.com/crossplane_io
- Youtube: Crossplane Youtube

Calling all Crossplane Adopters!

🚨 We'd love to hear about your adoption of Crossplane, please share your story in ADOPTERS.md in the crossplane/crossplane repo 🚨