KubeCon CloudNativeCon

OPEN SOURCE SUMMIT

China 2019

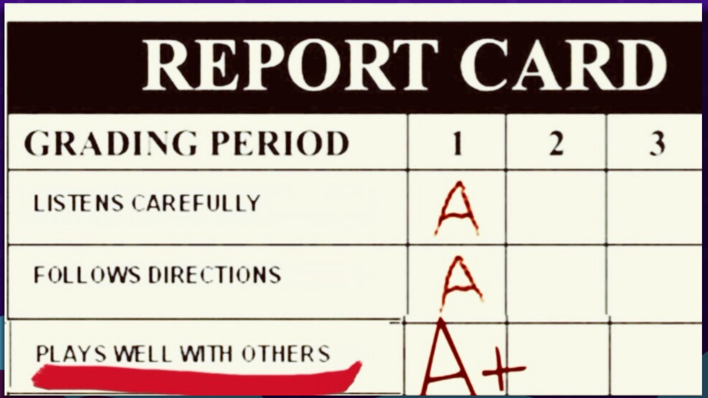
“Plays Well With Others”

Composability for Cloud Native Applications

Steve Judkins

Program Manager, Upbound

@infinitecompute



REPORT CARD

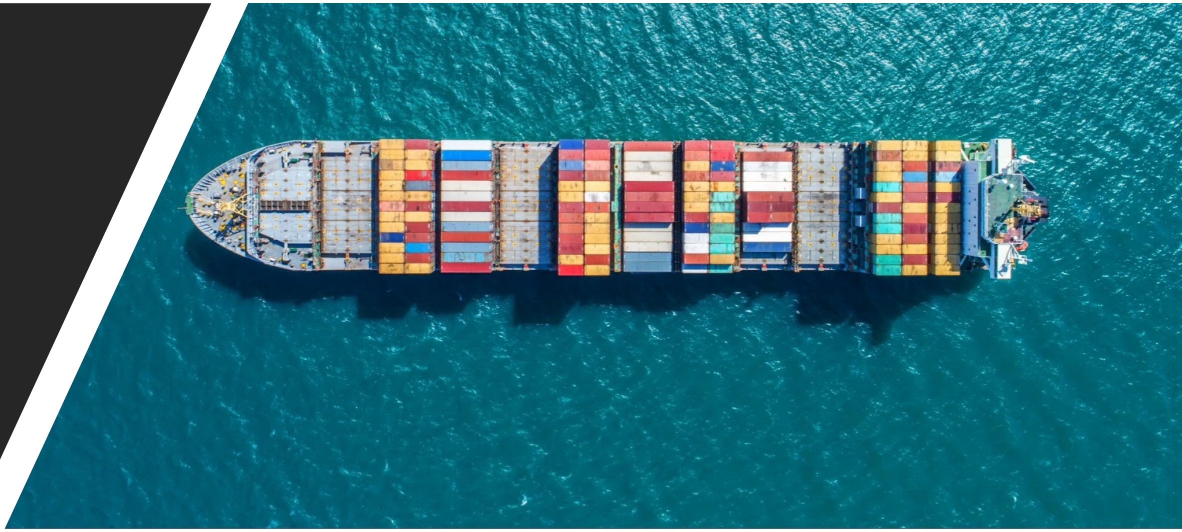
GRADING PERIOD	1	2	3
LISTENS CAREFULLY	A		
FOLLOWS DIRECTIONS	A		
PLAYS WELL WITH OTHERS	A+		

We ❤ Kubernetes

- It's easy to manage the services we build and deploy in a declarative way
- Active state controllers for reconciliation
- Containers for our services
- Proven scalability
- It is extensible!

Modern cloud native applications

- Leverage managed Kubernetes for your apps
- But use cloud managed services in production
  - Database replication and backups, DR, elasticity, etc.
- Use advanced cloud provider functionality like search, AI/ML, that is a pain to manage in cluster

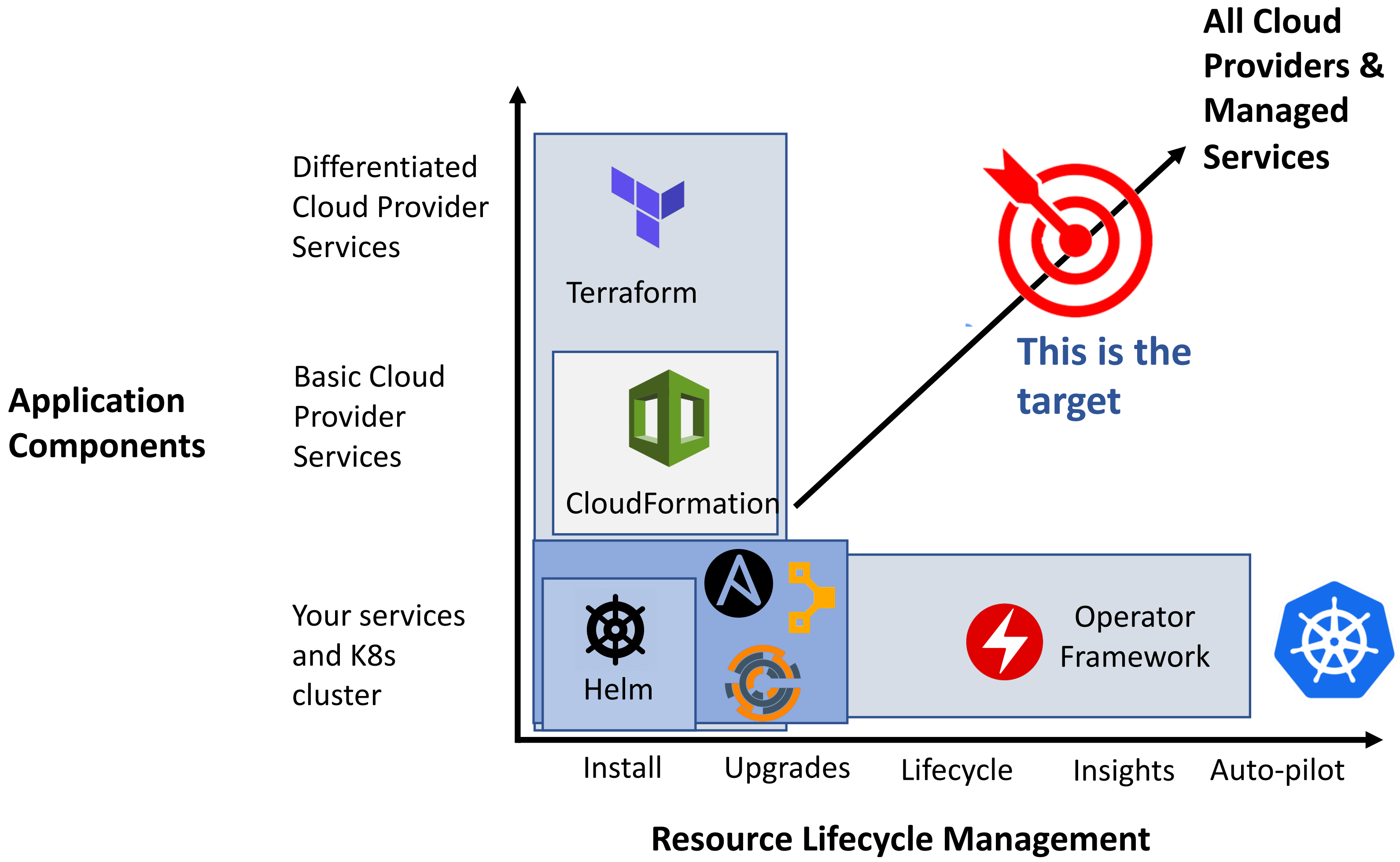


What’s wrong with this picture?

Modern applications are composed of more than just the services you write and own…

- You have dependencies on databases, buckets, pub/sub, search, monitoring, etc.
- But do you really want all these running in your own cluster in production?
  - Do you want to be paged at midnight? I didn’t think so!
- Also, your IT DevOps are using a completely different set of tools to provision & orchestrate cloud services
  - It’s a dumpster fire of tools!

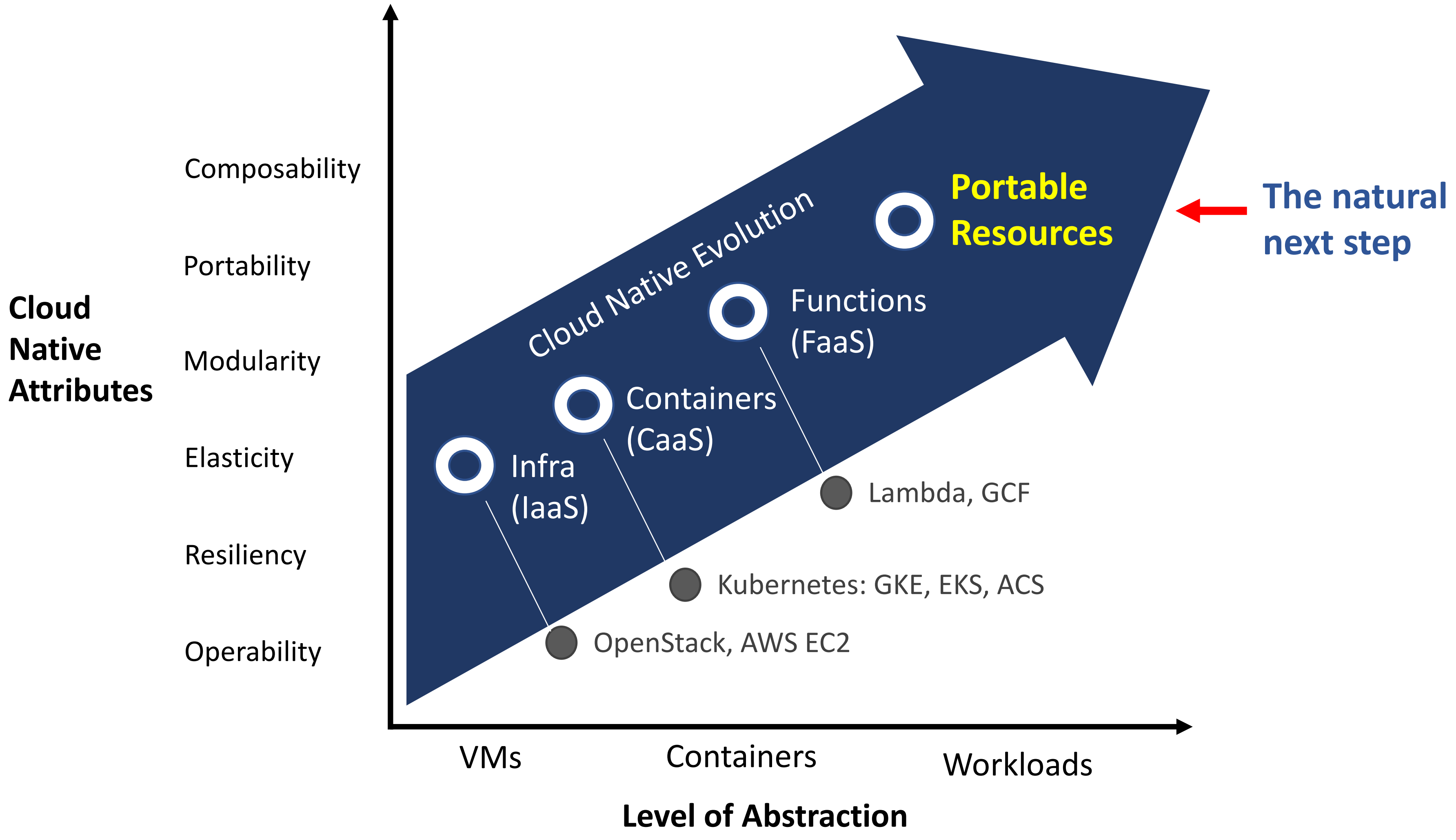
Infrastructure Orchestration


All Cloud Providers & Managed Services
Differentiated Cloud Provider Services
Terraform
Basic Cloud Provider Services
CloudFormation
Application Components
This is the target
Your services and K8s cluster
Helm
Operator Framework
Install
Upgrades
Lifecycle
Insights
Auto-pilot
Resource Lifecycle Management

Can we solve this in an elegant way?

- Based on Kubernetes engine
- That brings cloud provider services and infrastructure into Kubernetes
- One API to manage your infrastructure
- Provide portability for heterogeneous workloads beyond containers

Cloud Native Evolution



Cloud Native Attributes

Composability

Portability

Modularity

Elasticity

Resiliency

Operability

Cloud Native Evolution

Infra (IaaS)

Containers (CaaS)

Functions (FaaS)

Portable Resources

The natural next step

OpenStack, AWS EC2

Kubernetes: GKE, EKS, ACS

Lambda, GCF

VMs

Containers

Workloads

Level of Abstraction

Building on the Kubernetes Engine

- Declarative API
- `kubectl` native integration as well as other tools, libraries, and UI
- Rich ecosystem and community growing around Kubernetes
- Lets apply the lessons learned from container orchestration to multicloud workloads and resources

Resource lifecycle management

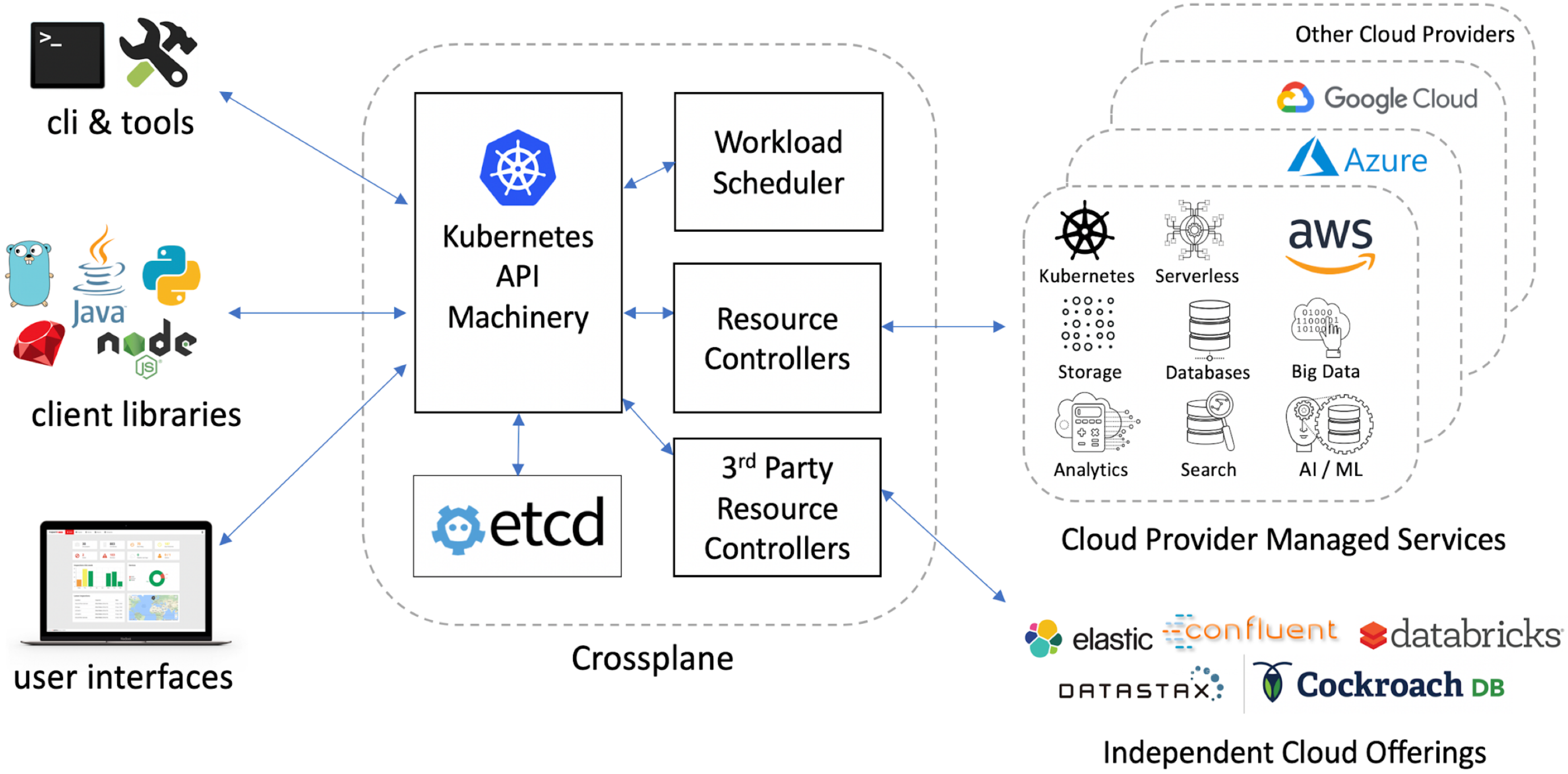
- **Custom Resources (CRDs):** model cloud provider services and infrastructure as well as independent cloud offerings
- **Custom controllers:** provision, configure, scale, monitor, upgrade, failover, backup, and more
- **Active reconciliation:** responds to external changes that deviate from the desired configuration

Portable resource abstractions

- Powerful “volume” abstraction in Kubernetes - portability of stateful applications
- What about other resources? databases, buckets, clusters, caches, message queues, data pipelines, AI/ML, etc.
- Let’s abstract those too!
- **Write once, run anywhere**

Crossplane

Open source multicloud control plane



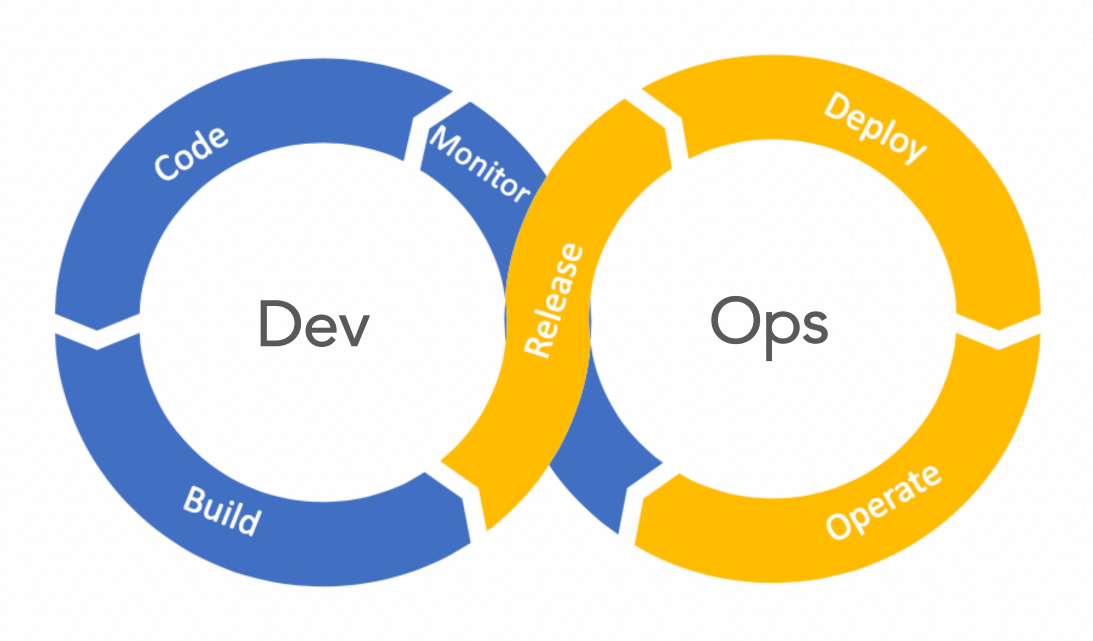
Separation of concerns

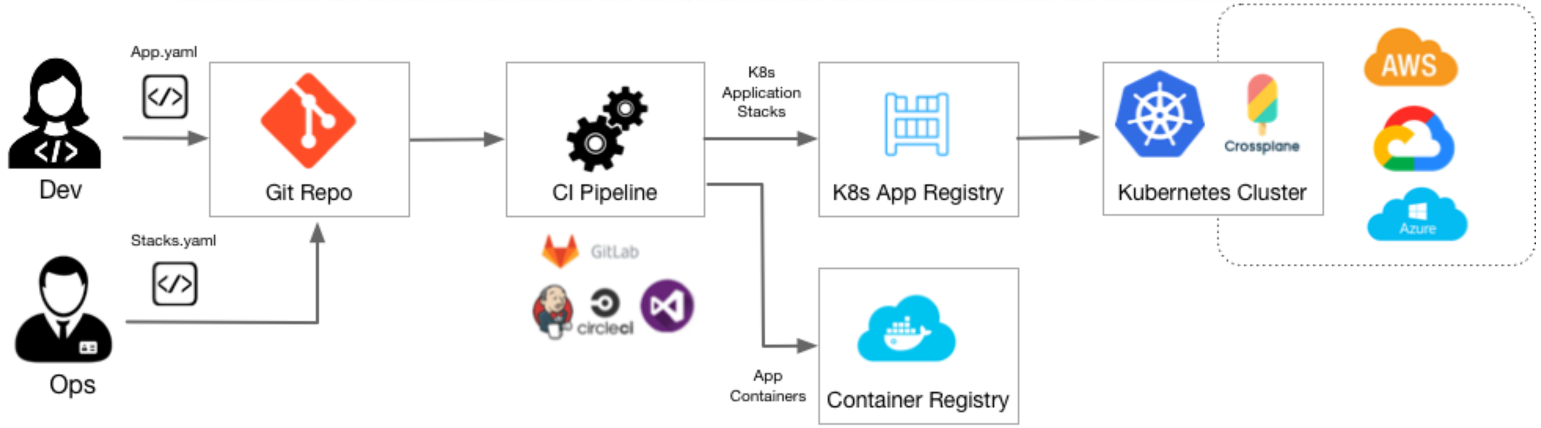
- **Developer** composes their app and resources in a general way
  - Not tightly coupled at app dev time
- **Administrator** defines environment specifics and policies
- Modeled as resource claims and resource classes
  - similar to PVC and StorageClass
- Dynamic (on-demand) provisioning of resources

Deconstructed image courtesy of Todd McLellan

GitOps for cloud native apps

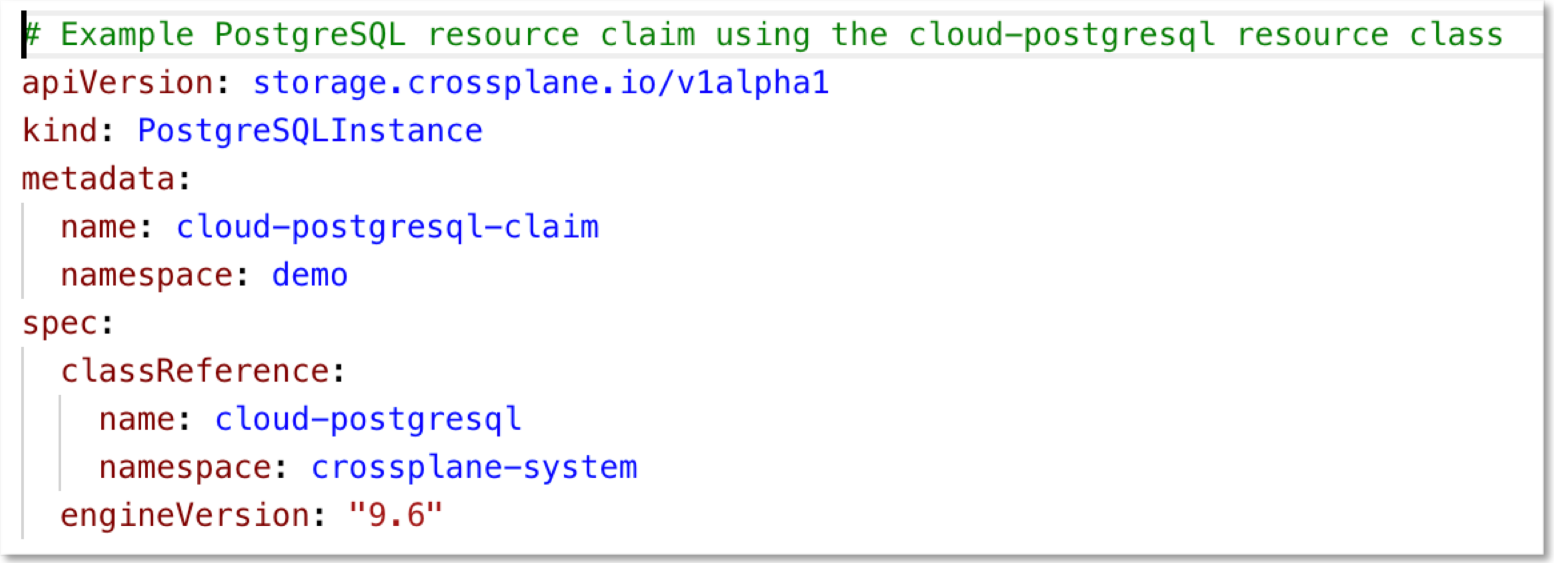
- App owner YAML
  - Resource Claims
  - Workloads
- Administrator YAML
  - Resource Classes
  - Providers
  - Concrete resources
- Dev and Ops converge
  - A single app definition for the stack





Resource Claim

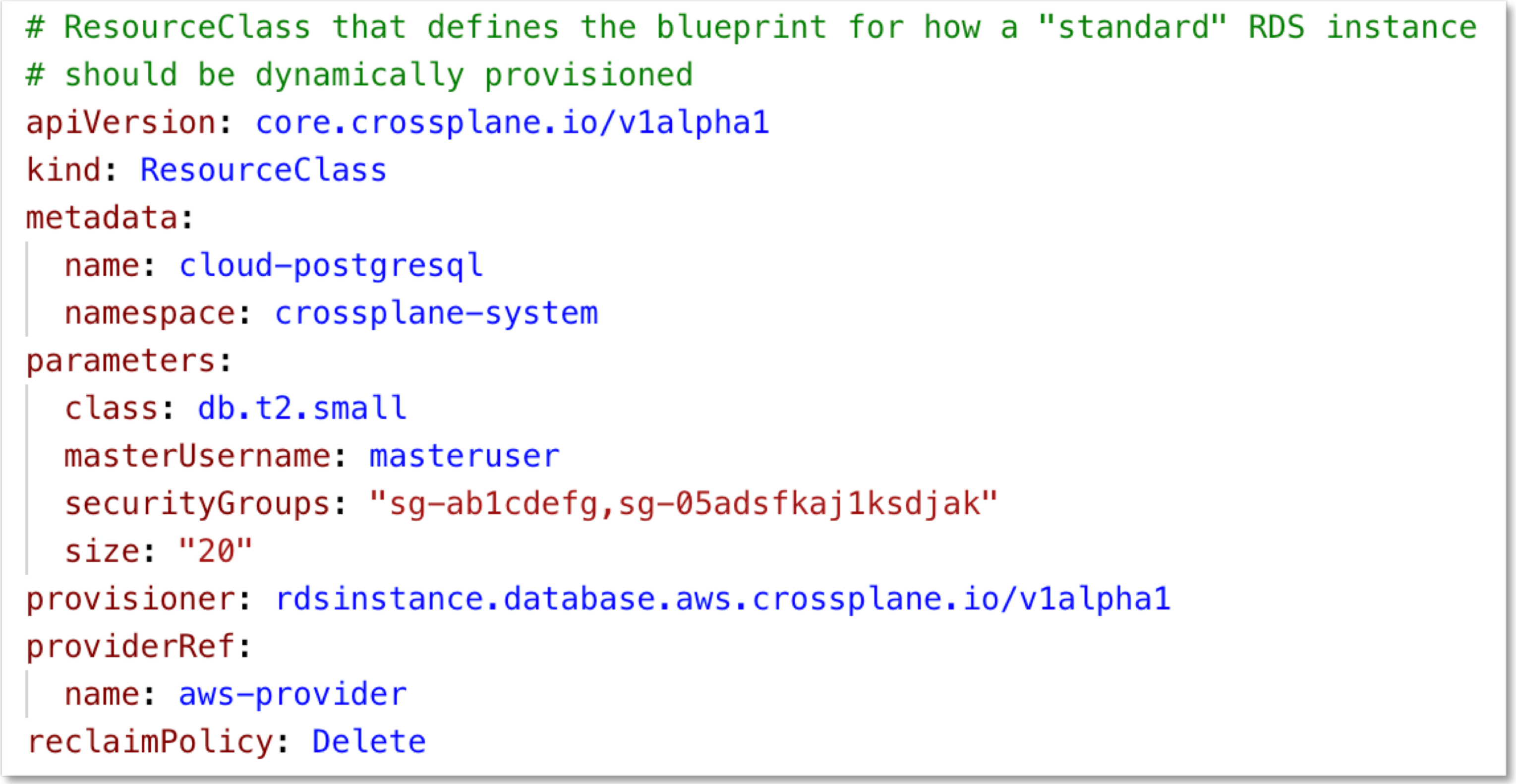
- App owner YAML
  - Resource Claims
  - Workloads
- App specifies a cloud postgreSQL dependency



```yaml
# Example PostgreSQL resource claim using the cloud-postgresql resource class
apiVersion: storage.crossplane.io/v1alpha1
kind: PostgreSQLInstance
metadata:
  name: cloud-postgresql-claim
  namespace: demo
spec:
  classReference:
    name: cloud-postgresql
    namespace: crossplane-system
  engineVersion: "9.6"
```

Resource Class

- Administrator YAML
  - Resource Classes
  - Providers
  - Concrete resources
- Administrator defines where PostgreSQL is dynamically provisioned
  - e.g. AWS RDS in example



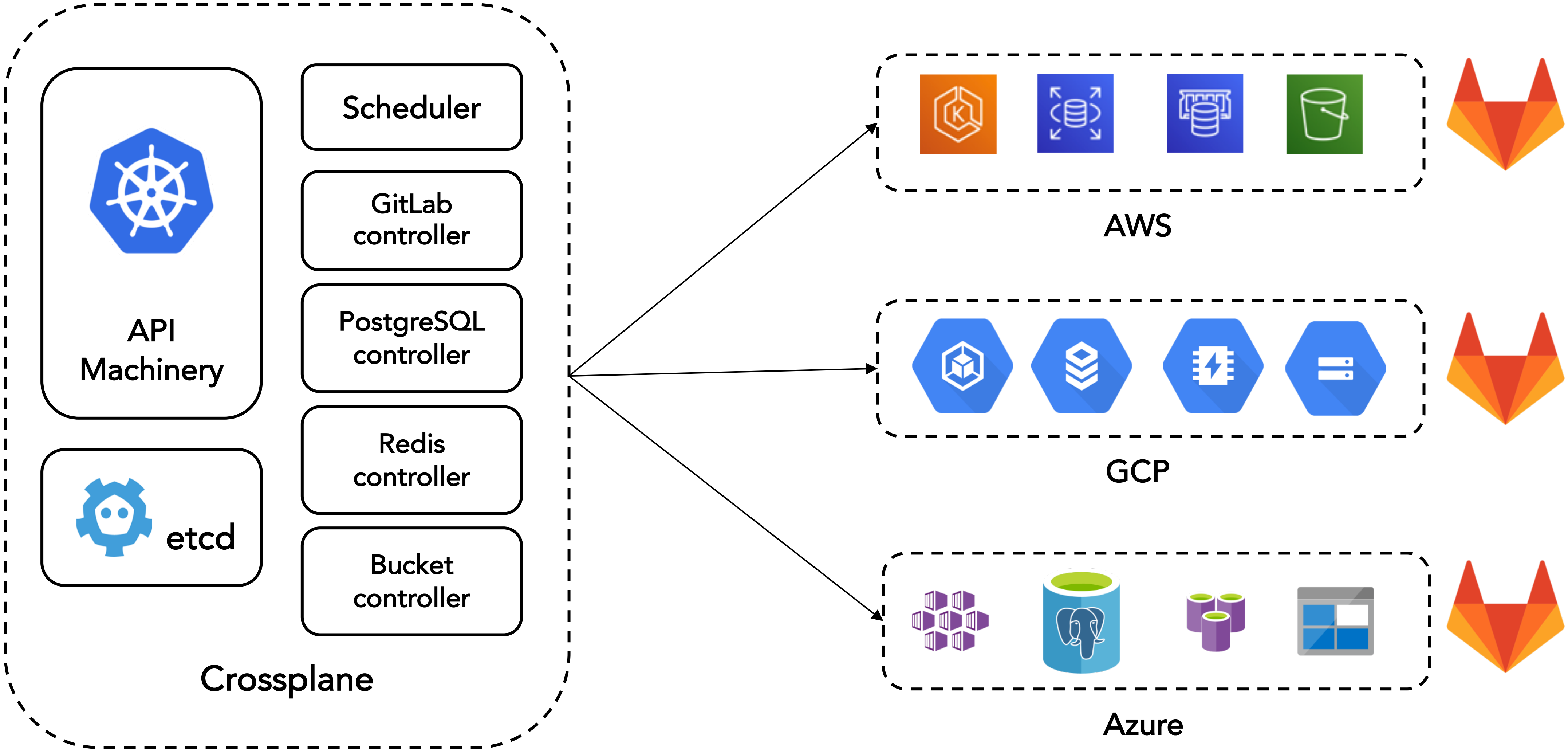
```yaml
# ResourceClass that defines the blueprint for how a "standard" RDS instance
# should be dynamically provisioned
apiVersion: core.crossplane.io/v1alpha1
kind: ResourceClass
metadata:
  name: cloud-postgresql
  namespace: crossplane-system
parameters:
  class: db.t2.small
  masterUsername: masteruser
  securityGroups: "sg-ab1cdefg,sg-05adsfkaj1ksdjak"
  size: "20"
provisioner: rdsinstance.database.aws.crossplane.io/v1alpha1
providerRef:
  name: aws-provider
reclaimPolicy: Delete
```

GitLab on Crossplane

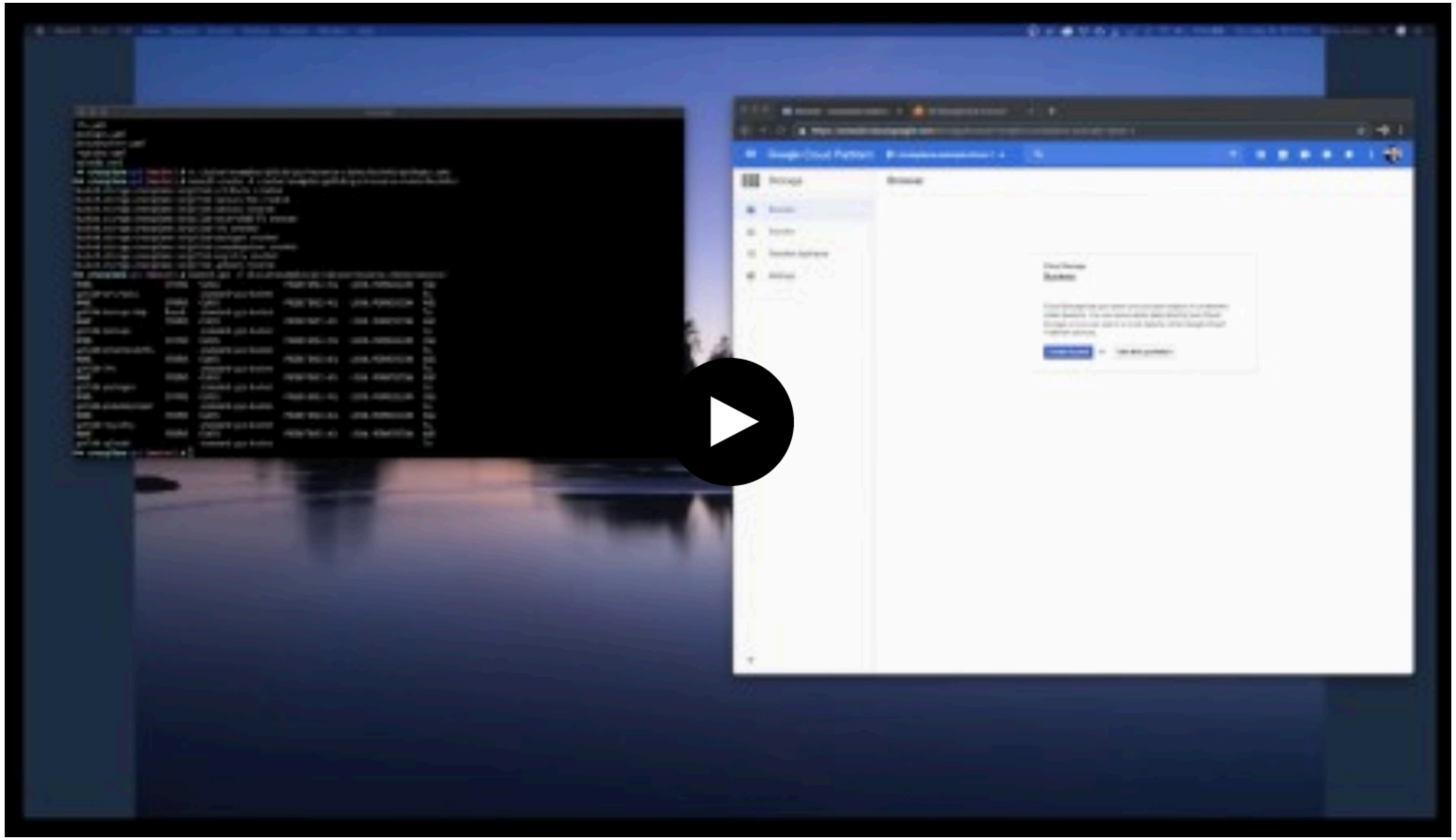
- Real world (complex) application
  - Currently a Helm chart
  - 4,800 lines of YAML, 14 Deployments, 3 Jobs, 9 Services, 16 ConfigMaps, etc.
- PostgreSQL, Redis, Object storage
- How can we make this better?
  - CRD - simple config experience
  - Custom controller to generate artifacts
  - Fully automated and portable multi-cloud deployment

GitLab on Crossplane

KubeCon CloudNativeCon OPEN SOURCE SUMMIT China 2019



Demo



How to get involved?

- Contribute to Crossplane
  - https://github.com/crossplaneio/crossplane/
  - https://crossplane.io/
- Slack - https://slack.crossplane.io/
- Twitter - @crossplane_io
- Forums - crossplane-dev on google groups
- Community Meetings - every other Tuesday 9am Pacific