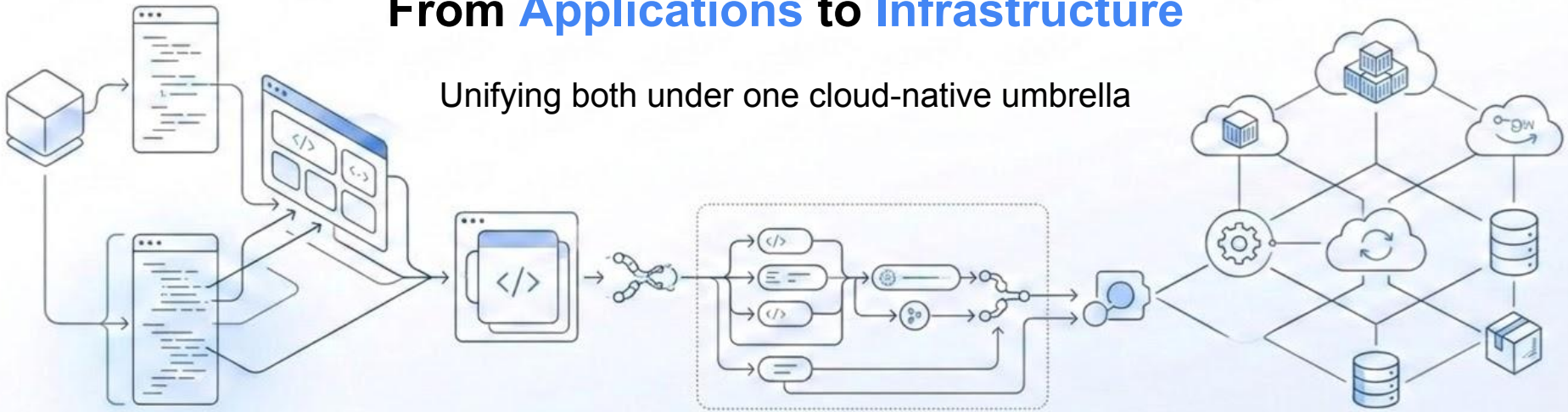
From Applications to Infrastructure

Unifying both under one cloud-native umbrella



From Applications to Infrastructure

Unifying both under one cloud-native umbrella

Julien Semaan
Head of Kubernetes Engineering
Kubex

Corey McGalliard
Engineering Manager
Akamai Cloud

Designing for the Tenant

Choosing the right multi-tenancy mode is a balance between isolation and efficiency

One Cluster Per Tenant

High isolation, higher overhead.

- Strong isolation
- Higher cost & operational overhead

Multi-Tenant Clusters

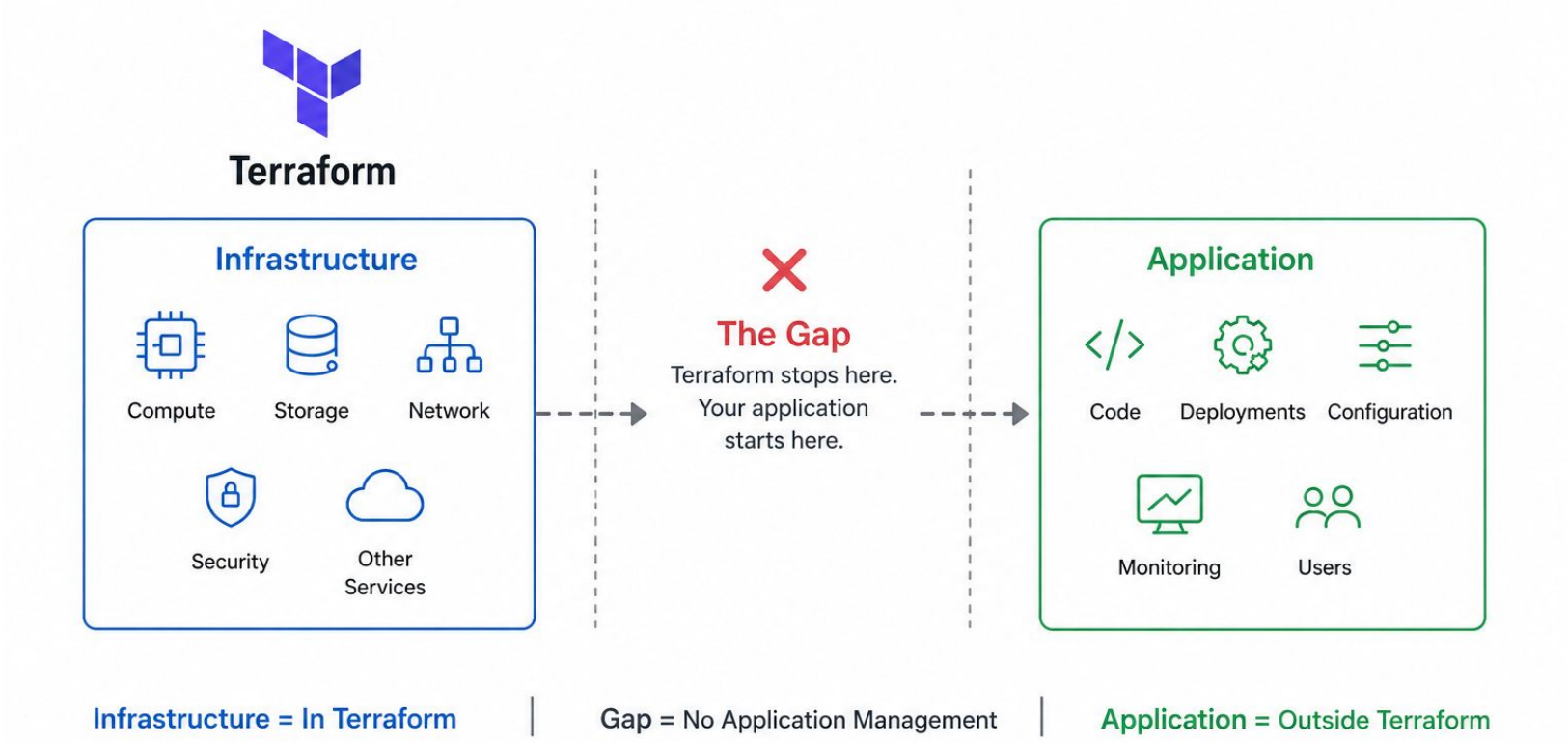
Shared resources, strict logical boundaries.

- Resource efficiency
- Requires strong isolation & governance

Right model, right balance. Align with tenant needs, scale goals, and operational capabilities.

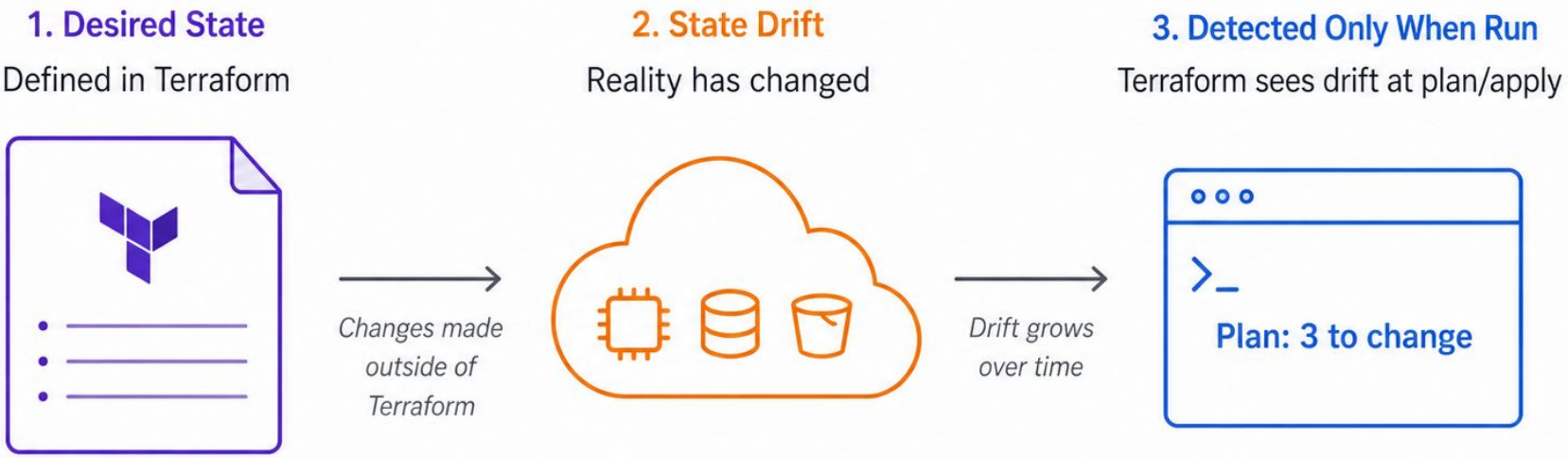
Terraform Manages Infrastructure, Not Your Application

Choosing the right multi-tenancy mode is a balance between isolation and efficiency



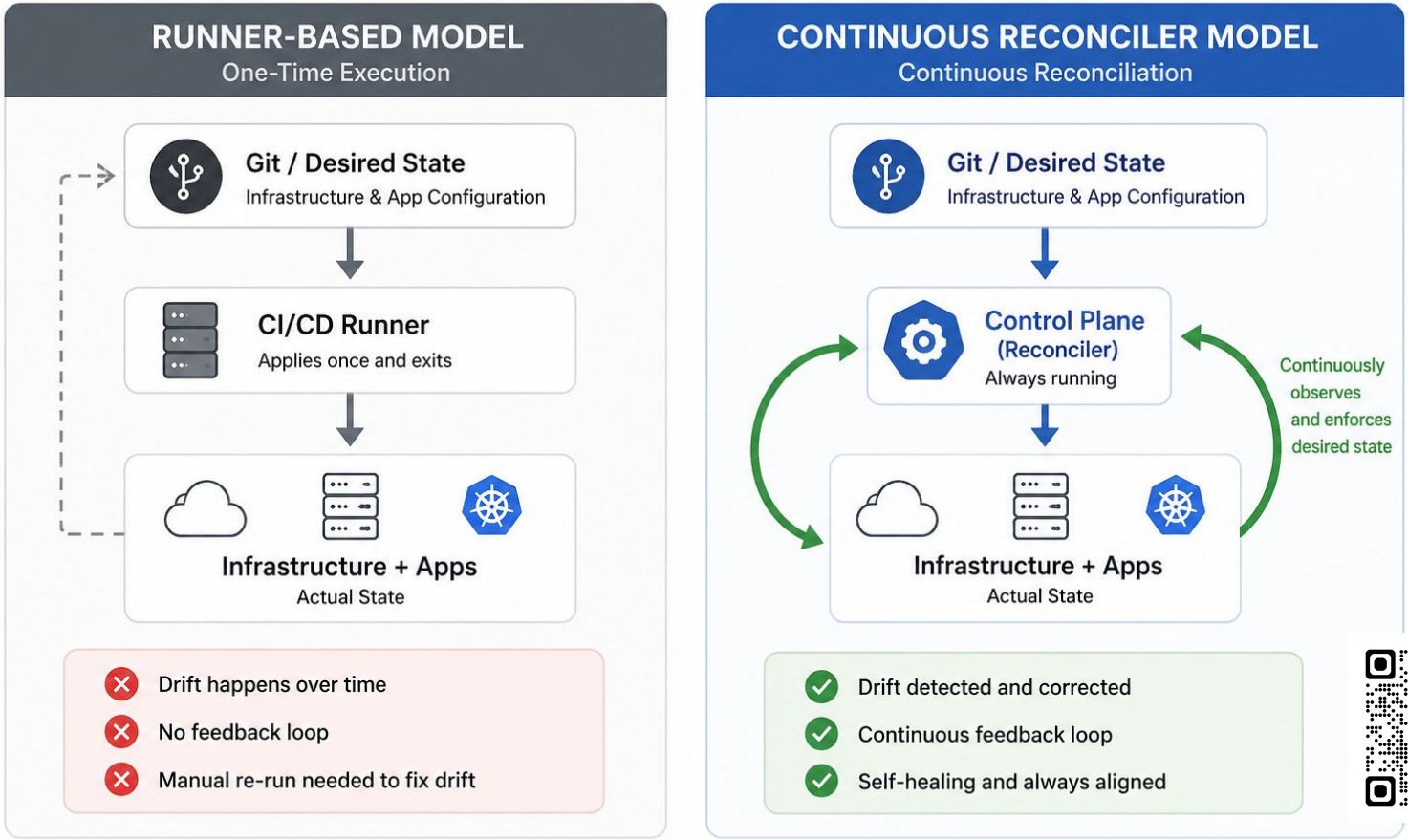
Problem: State Drift in Terraform

No continuous reconciliation -> Reality drifts from desired state

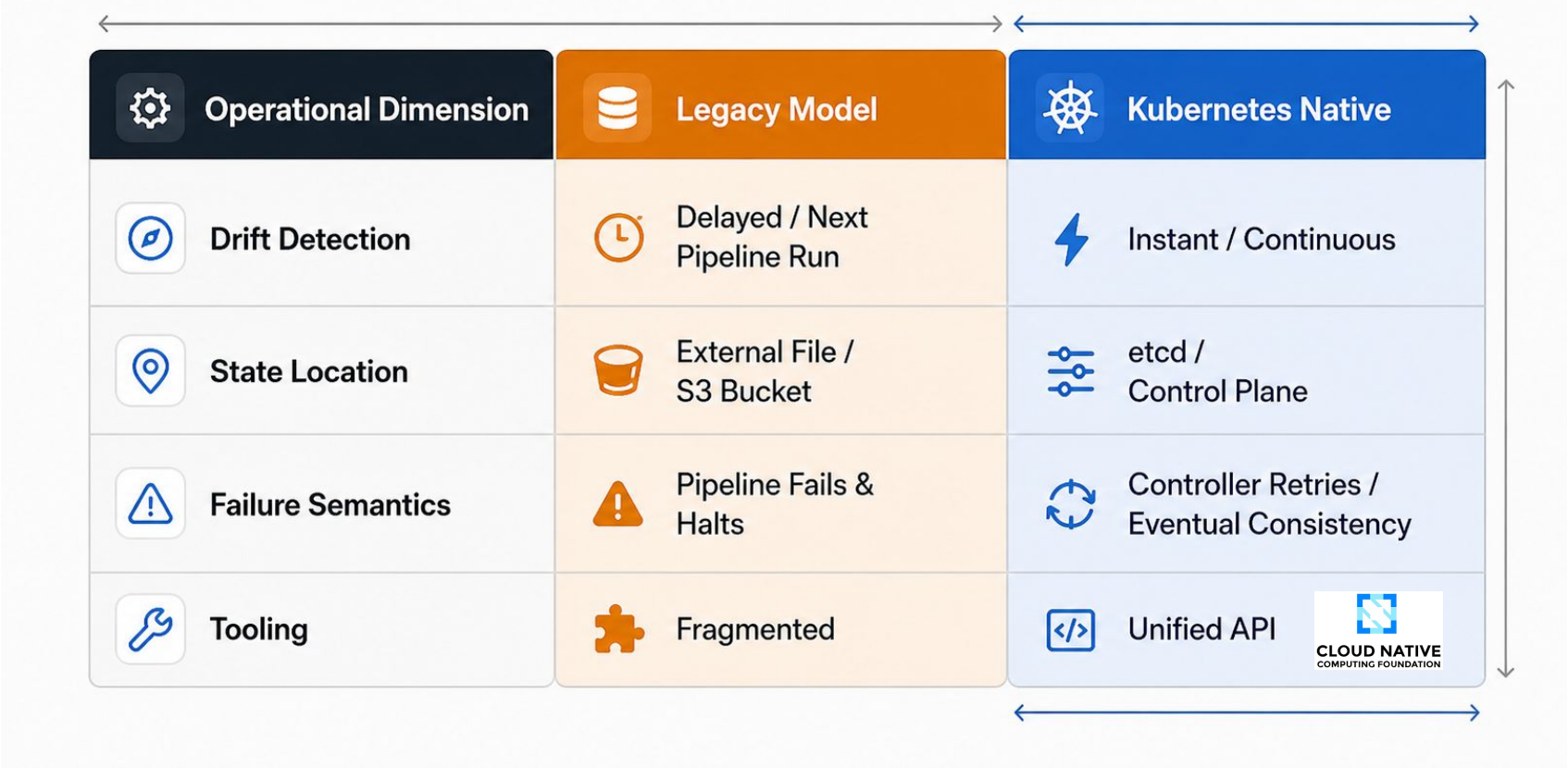


Terraform enforces your intent only when you run it—there is no continuous reconciliation.

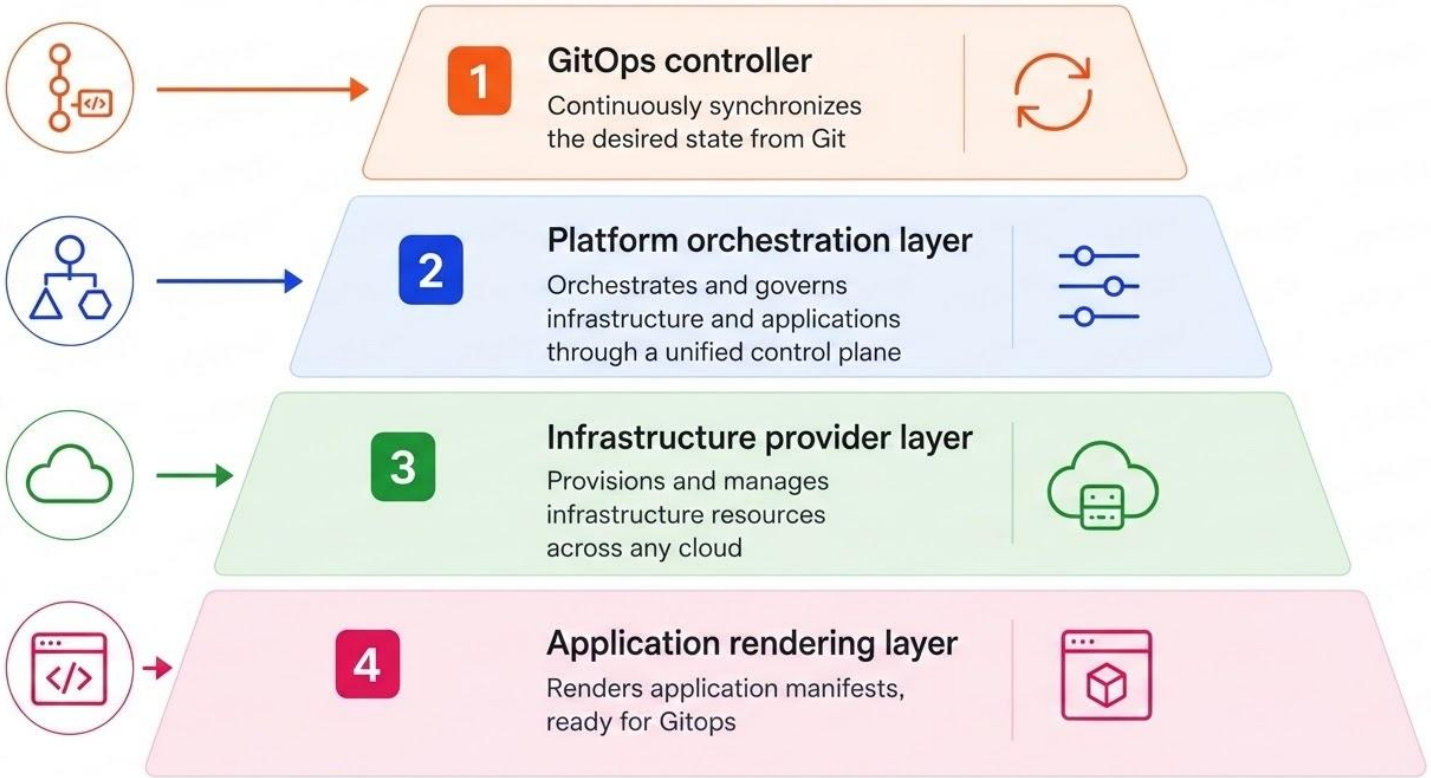
Continuous Reconciliation vs One-time Execution



Speaking the same Language



A Cloud-Native model to make it work



Example implementation

CLOUD NATIVE COMPUTING FOUNDATION

GitOps controller

Argo CD continuously synchronizes the desired state from Git

Platform orchestration layer

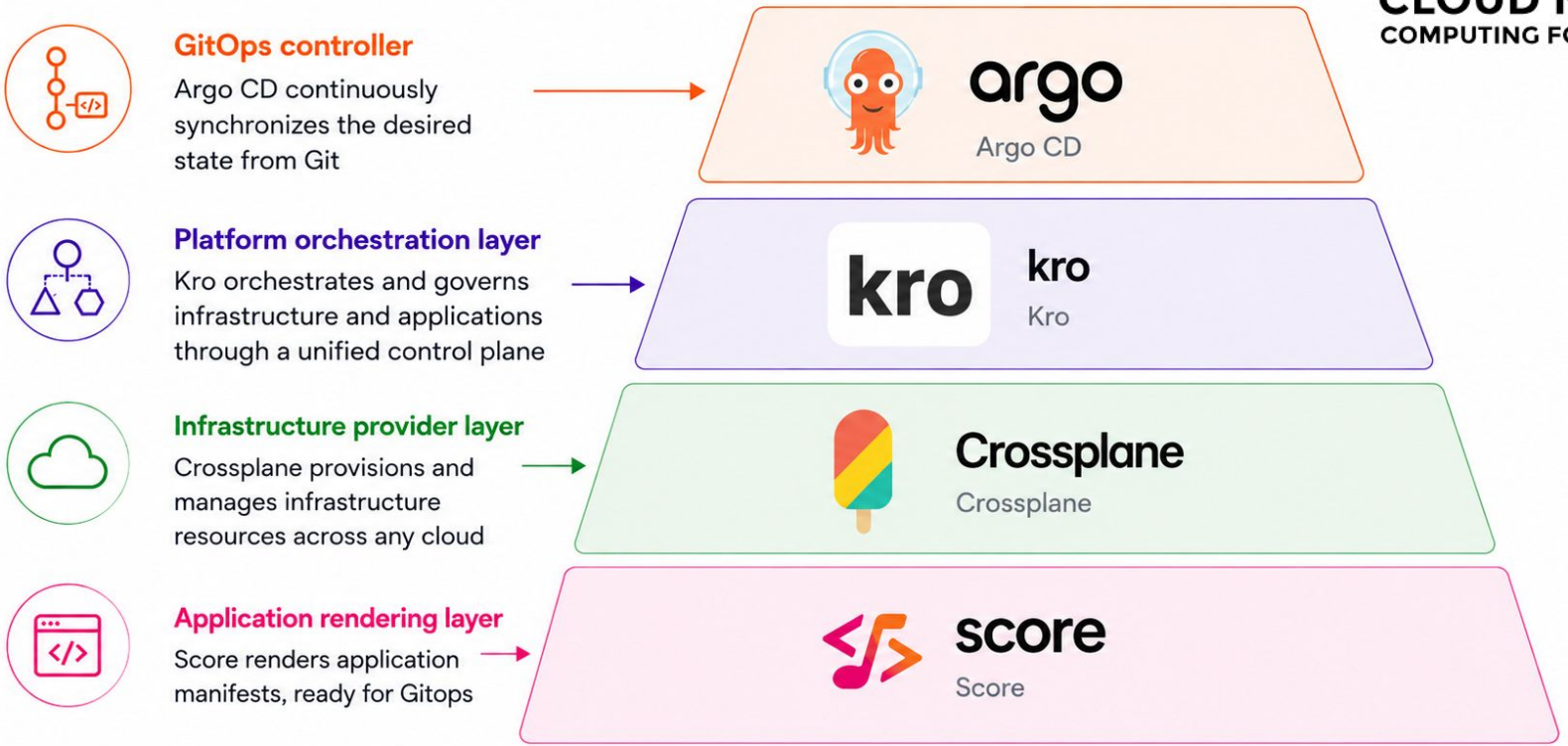
Kro orchestrates and governs infrastructure and applications through a unified control plane

Infrastructure provider layer

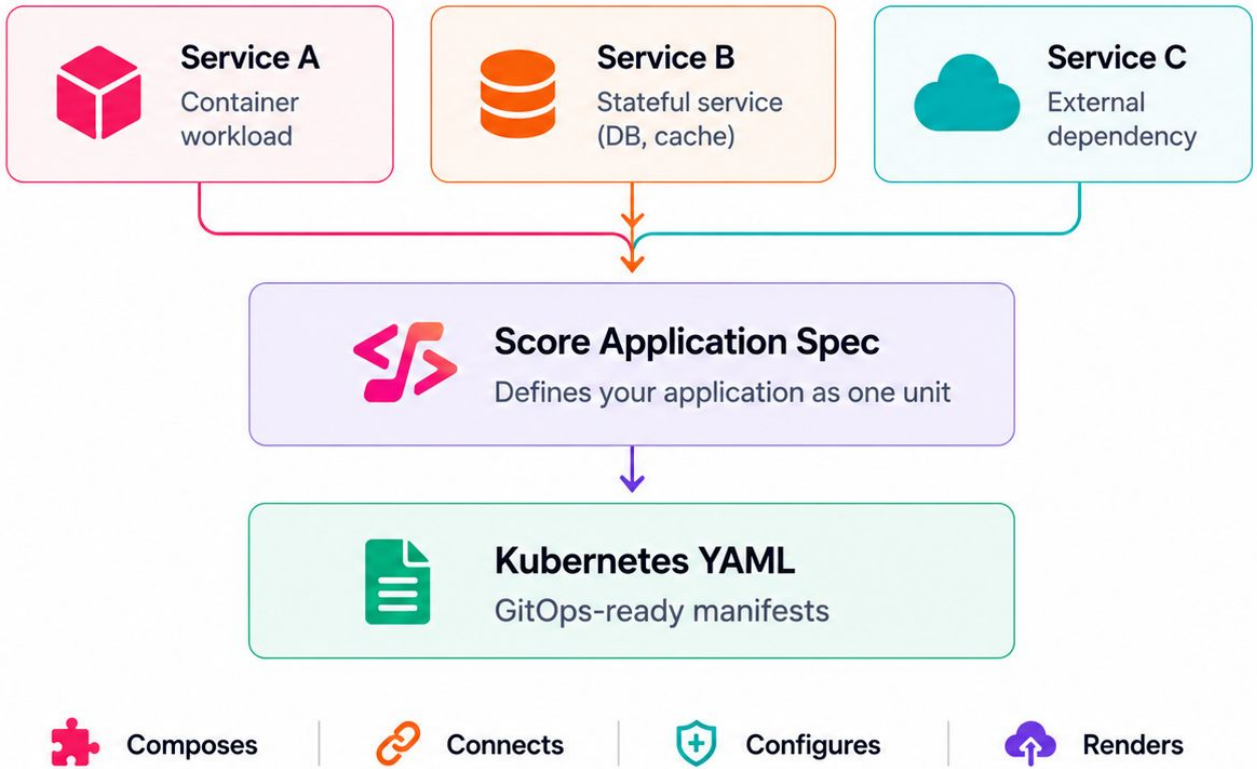
Crossplane provisions and manages infrastructure resources across any cloud

Application rendering layer

Score renders application manifests, ready for Gitops



score: Application rendering layer



Translates intent into Kubernetes manifests

Your Score YAML
Simple intent

```yaml
apiVersion: score.dev/v1b1
metadata:
  name: hello-world
  annotations:
    tags: "nodejs,http,website,javascript,postgres"
containers:
  hello-world:
    image: scorespec/sample-score-app:latest
    variables:
      -...>
resources:
  db:
    type: postgres
  dns:
    type: dns
  route:
    type: route
    params:
      host: ${resources.dns.host}
      path: /
      port: 8080
  service:
    ports:
      www:
        port: 8080
        targetPort: 3000
```

score

Score expands your intent into production-ready Kubernetes manifests

Complete Kubernetes YAML resources
Everything needed to run your app

Namespace
Isolates and scopes your application

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: hello-world
```

ConfigMap / Secret
Configuration and sensitive data

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: hello-world-config
```

Deployment
Runs and manages your container

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world
```

Service
Stable network endpoint for your app

```yaml
apiVersion: v1
kind: Service
metadata:
  name: hello-world
```

Ingress / HTTPRoute
HTTP routing and traffic management

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: hello-world
```

Database (Postgres)
Stateful service for your data

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: hello-world-db
```

DNS / ExternalName
Internal DNS for service discovery

```yaml
apiVersion: v1
kind: Service
metadata:
  name: hello-world-dns
```

Other supporting resources
ServiceAccount, RBAC, Probes, NetworkPolicy, etc.

...

Crossplane: Infrastructure Management

Multi-cloud

Use any provider and any service across AWS, Azure, GCP, and more.

aws

Terraform Providers

VMware vSphere

And more

Works with providers you already use.

What Crossplane manages

Database
Managed databases and data stores

Network
Network and connectivity resources

Cluster
Compute and container platforms

Model once, provision anywhere.

How Crossplane works

You define infrastructure in Kubernetes

Crossplane provisions and configures

Cloud provider APIs deliver and manage the resources

Infrastructure as code, with Kubernetes.

Crossplane Resources are powerful but verbose and complex

Infrastructure and code with many moving parts, deep nesting, and provider-specific fields

```yaml
# Provider
apiVersion: v1
kind: Secret
metadata:
  name: linode-creds
  namespace: crossplane-system

type: Opaque

stringData:
  credentials: |
    { "token": "REPLACE_WITH_LINODE_API_TOKEN" }

---

apiVersion: linode.upbound.io/v1beta1
kind: ProviderConfig
metadata:
  name: linode-provider

spec:
  credentials:
    source: Secret
    secretRef:
      namespace: crossplane-system
      name: linode-creds
      key: credentials
```

Verbose and hard to manage

- Many resource types to define
- Deeply nested and repetitive
- Provider-specific and hard to maintain

```yaml
---
apiVersion: lke.linode.upbound.io/v1alpha1
kind: Cluster
metadata:
  name: prod-lke
spec:
  deletionPolicy: Orphan
  providerConfigRef:
    name: linode-provider
  writeConnectionSecretToRef:
    name: prod-lke-connection
    namespace: crossplane-system
  forProvider:
    label: prod-lke
    region: ca-central
    k8sVersion: "1.31"
    controlPlane:
      highAvailability: true
    tags:
      - production
      - crossplane
      - lke
    pool:
      - type: g6-standard-4
        count: 3
        autoscaler:
          min: 3
          max: 6
      - type: g6-dedicated-8
        count: 2
        autoscaler:
          min: 2
          max: 4
```

KRO: Platform Orchestration

What is KRO?

KRO composes application definitions from Score (what to run) and infrastructure from Crossplane (where and how to run it) into a single platform resource.

KRO brings together

Score (Applications)
Defines what to run

Crossplane (Infrastructure)
Defines where and how to run it

Brings applications and infrastructure together.

How KRO works

One platform resource that composes everything.

How you use KRO

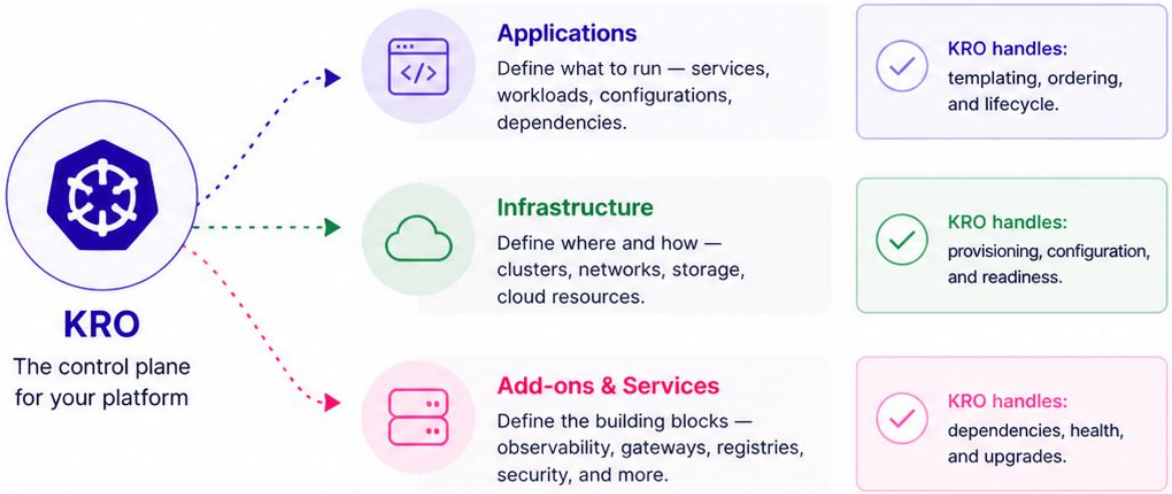
Create
Build your own simple Kubernetes custom resource that represents your platform intent.

Define
Define a simple custom resource (CRD) that captures the outcome you want KRO to deliver.

Compose
KRO composes it in the underlying custom resources using Score, Crossplane, and more.

Simple to define. Powerful to run.

KRO: Abstracting Complexities



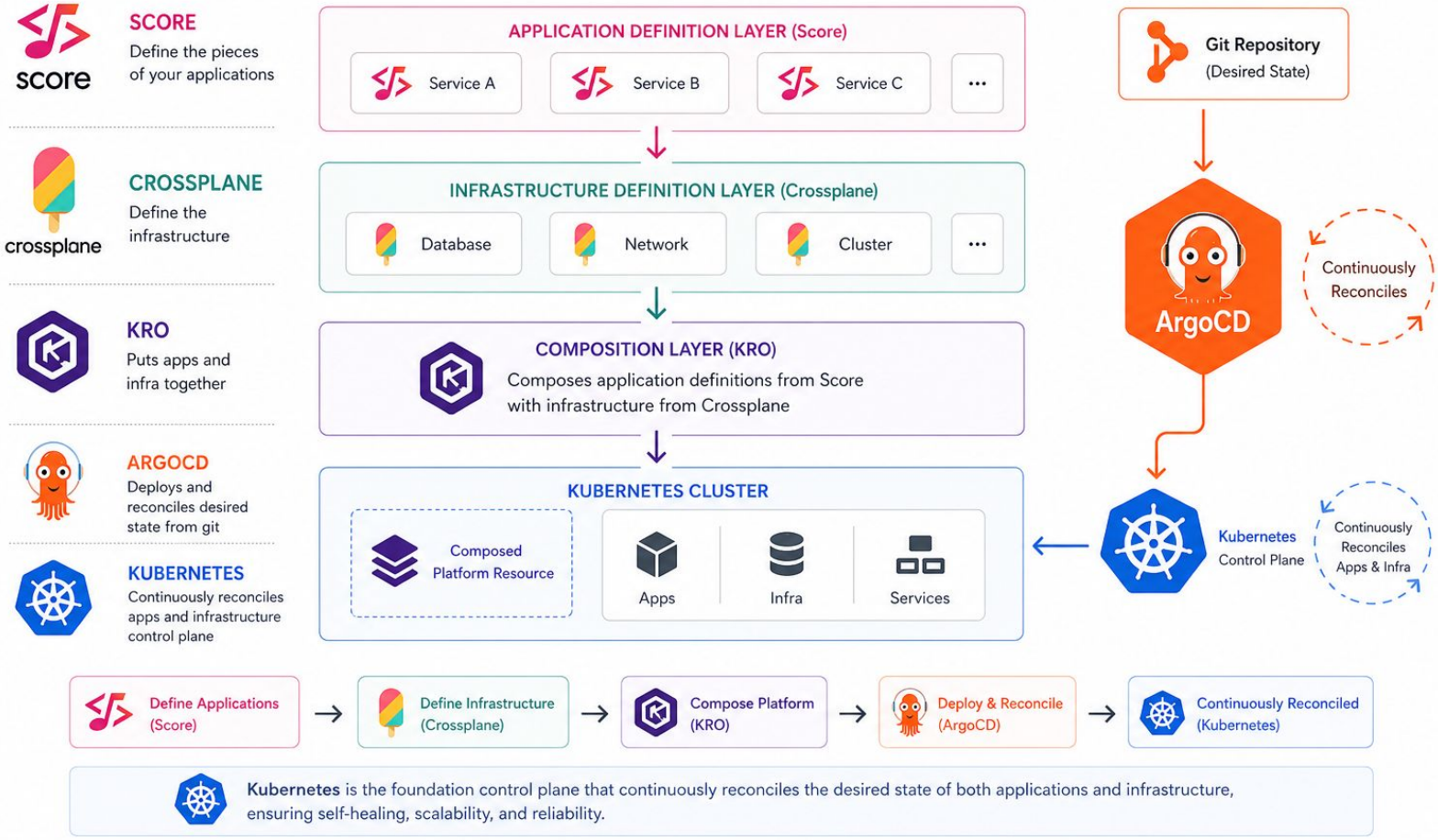
PlatformCluster.yaml

```yaml
apiVersion: api.platform.example.com/v1
kind: PlatformCluster
metadata:
  name: demo-cluster
  namespace: default
  labels:
    environment: dev
    team: platform-team
    purpose: development
spec:
  clusterSpec:
    clusterName: demo-cluster
    region: us-east
    kubernetesVersion: "1.35"
    nodePool:
      count: 3
      type: g6-standard-4
    labels:
      environment: dev
      team: platform-team
  components:
    gatewayFabric:
    enabled: true
    userApps:
      enabled: true
        - demo-app
   observabilityStack:
    enabled: true
    loki: true
    tempo: true
    grafana: true
    alloy: true
  harbor:
    enabled: false
  headlamp:
    enabled: true
```

ArgoCD: GitOps Operator

PLATFORM OVERVIEW

From application definition to continuously reconciled platform in your Kubernetes cluster



Demo

https://github.com/cmcgalliard/2026-oss-summit