KubeCon | CloudNativeCon
North America 2024

How Google Built a New Cloud on Top of Kubernetes

Jie Yu & Prashanth Venugopal
(with help from Saad Ali)
Google

cloud

Use Existing Tech?

Google stack built for planet scale

Building these technologies from scratch is very expensive.

Google has already invested in open sourcing many of these building blocks.

Would open source help?

Many of the building blocks existed

- Resource Manager... Kubernetes
- VM Manager... Kubevirt
- Service Mesh... Istio
- Image management... Harbor
- Dataplane... Cilium

Pros

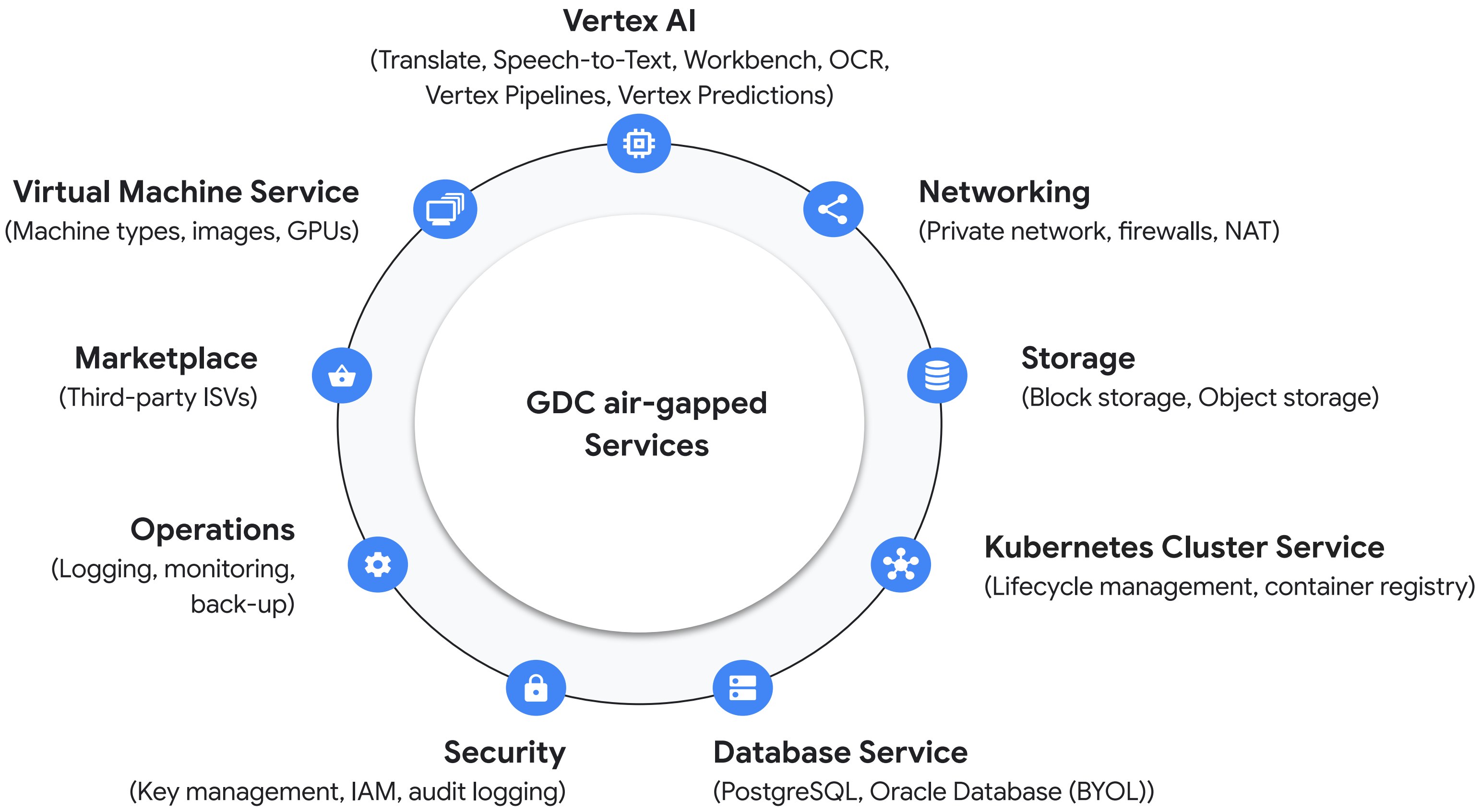
- Time to market
  - Building blocks exist
- Continuity for Customers
  - Pop open the hood, understand how it works, and be able to operate it themselves if necessary.
- Google is a big open source contributor
  - Lots of open source expertise
  - Leveraging existing investment
- OSS evolving quickly

Cons

- Requires rewriting proprietary services
- Missing functionality
- Unclear alignment with tenancy model.

GDC air-gapped is a hardware + software solution that delivers modern cloud technologies without connectivity to public cloud at any time.

GDC air-gapped

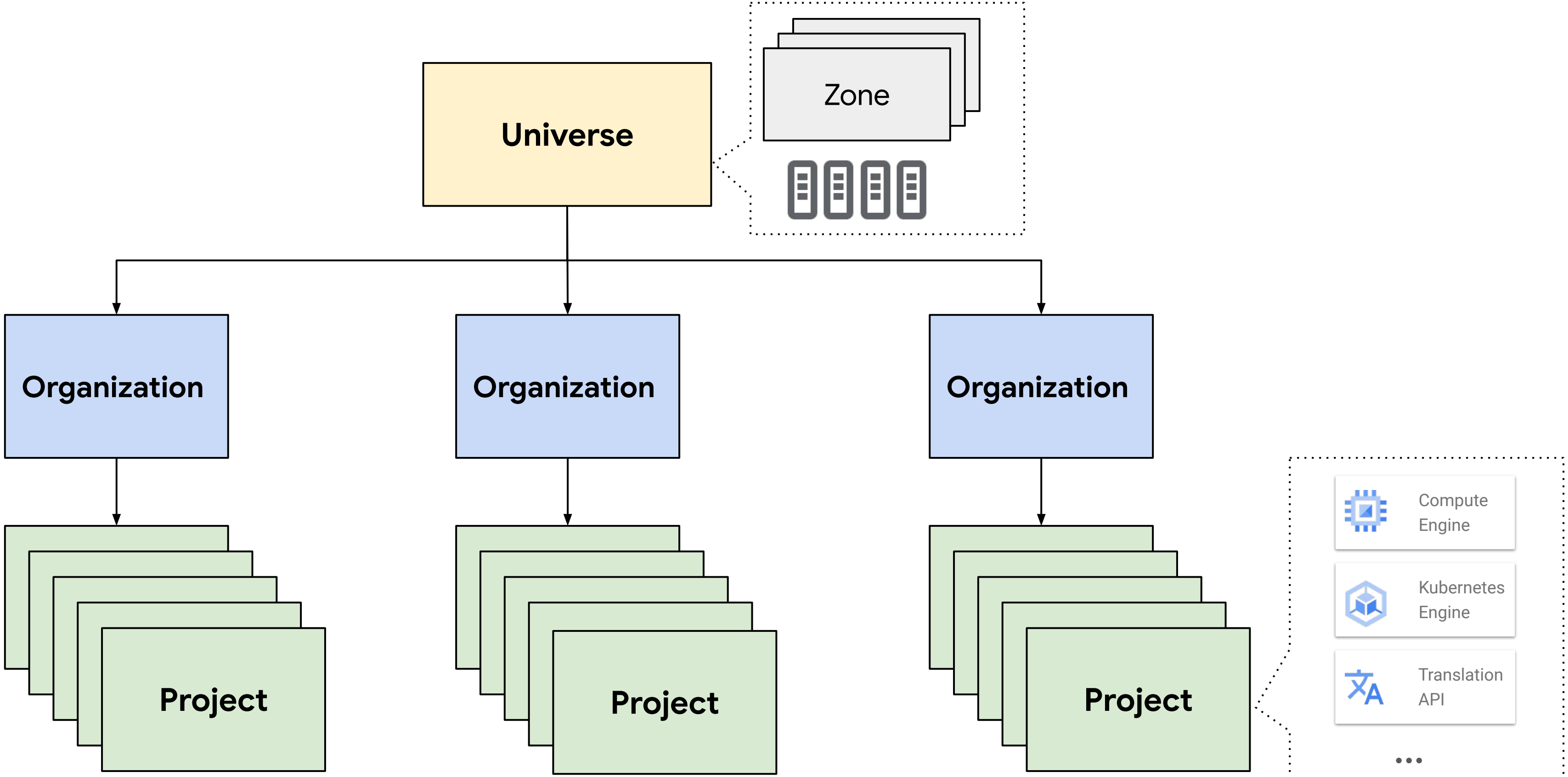

Vertex AI
(Translate, Speech-to-Text, Workbench, OCR, Vertex Pipelines, Vertex Predictions)
Virtual Machine Service
(Machine types, images, GPUs)
Networking
(Private network, firewalls, NAT)
Marketplace
(Third-party ISVs)
GDC air-gapped Services
Storage
(Block storage, Object storage)
Operations
(Logging, monitoring, back-up)
Kubernetes Cluster Service
(Lifecycle management, container registry)
Security
(Key management, IAM, audit logging)
Database Service
(PostgreSQL, Oracle Database (BYOL))

Goals of This Talk

Share 3 Key Design Principles

- Multi-cluster and namespace sameness
- KRM based API machinery and service controller pattern
- Treat containers and VMs equally

Key Concepts



Design Principle #1

Leverage...

Multi-Cluster

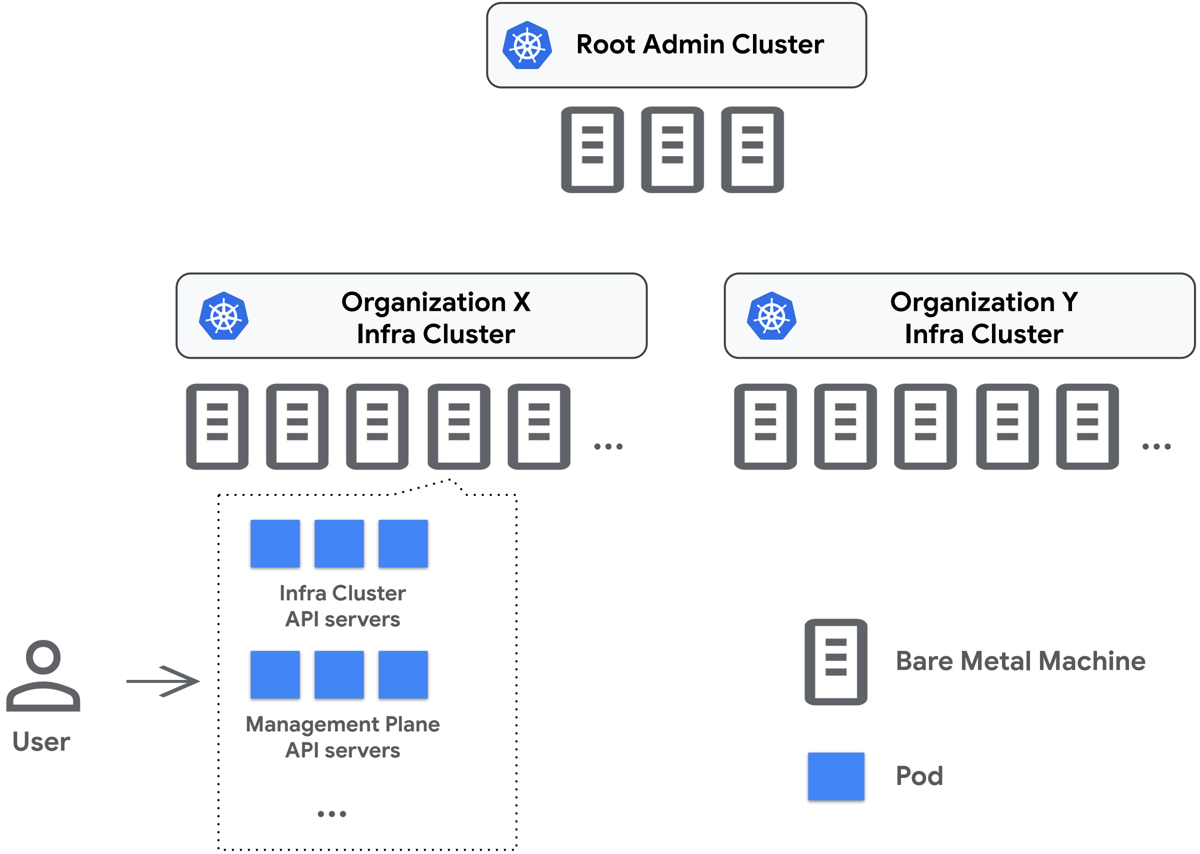
and

Namespace Sameness

Why Multi-Cluster?

- Need different types and shapes of clusters
    - Customers want Kubernetes clusters of arbitrary shape/size
    - VM Service needs Bare Metal Clusters
    - Other internal Services need cluster with special tuning
- Organization might span multiple zones/regions
- Need to support rack scale ⟶ DC scale
    - Cannot be all bare metal clusters
- Fault tolerance and reducing blast radius
- Better security isolation

Multi-Cluster Architecture



Root Admin Cluster

Organization X Infra Cluster

Organization Y Infra Cluster

...

Infra Cluster API servers

Management Plane API servers

...

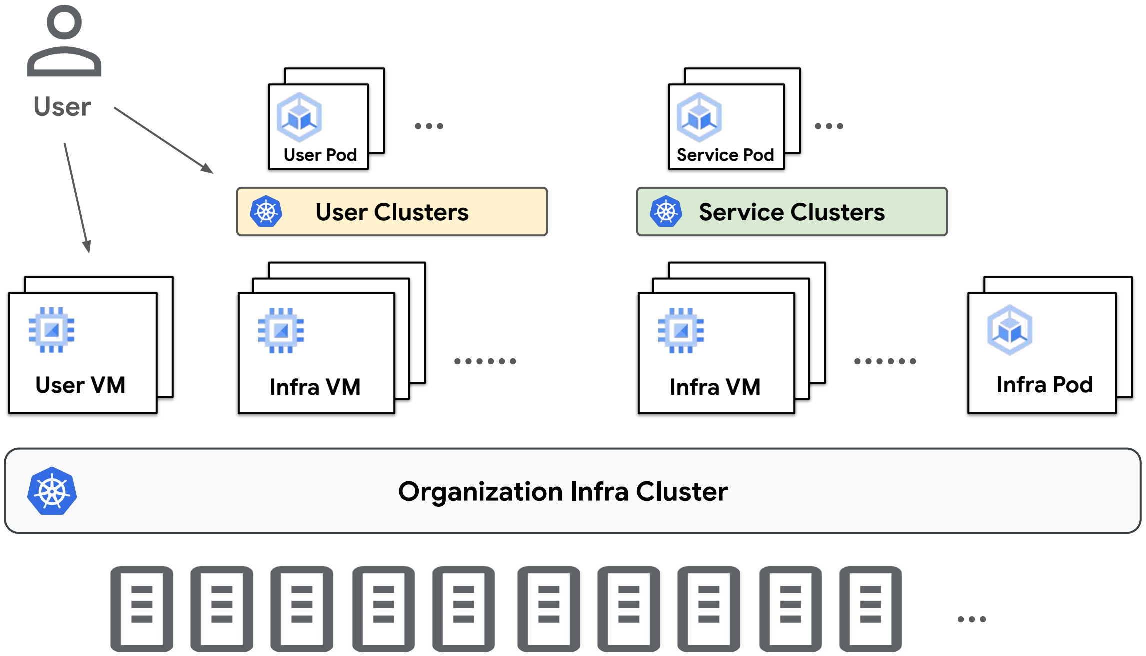
User

Bare Metal Machine

Pod

- **Root Admin Cluster**. One per private cloud zone, a 3 node Kubernetes cluster on bare metal servers, managing hardware resources and configurations
- **Organization Infra Cluster**. One per organization, dynamic sized and runs on bare metal servers
- **Dedicated Management Plane API Servers** per organization for CRUDL cloud resources

Multi-Cluster Architecture



- VMs scheduled and run on the Infra Cluster
  - **User VMs.** End user facing VMs
  - **Infra VMs.** Used for creating Kubernetes cluster nodes. Not visible to end users.

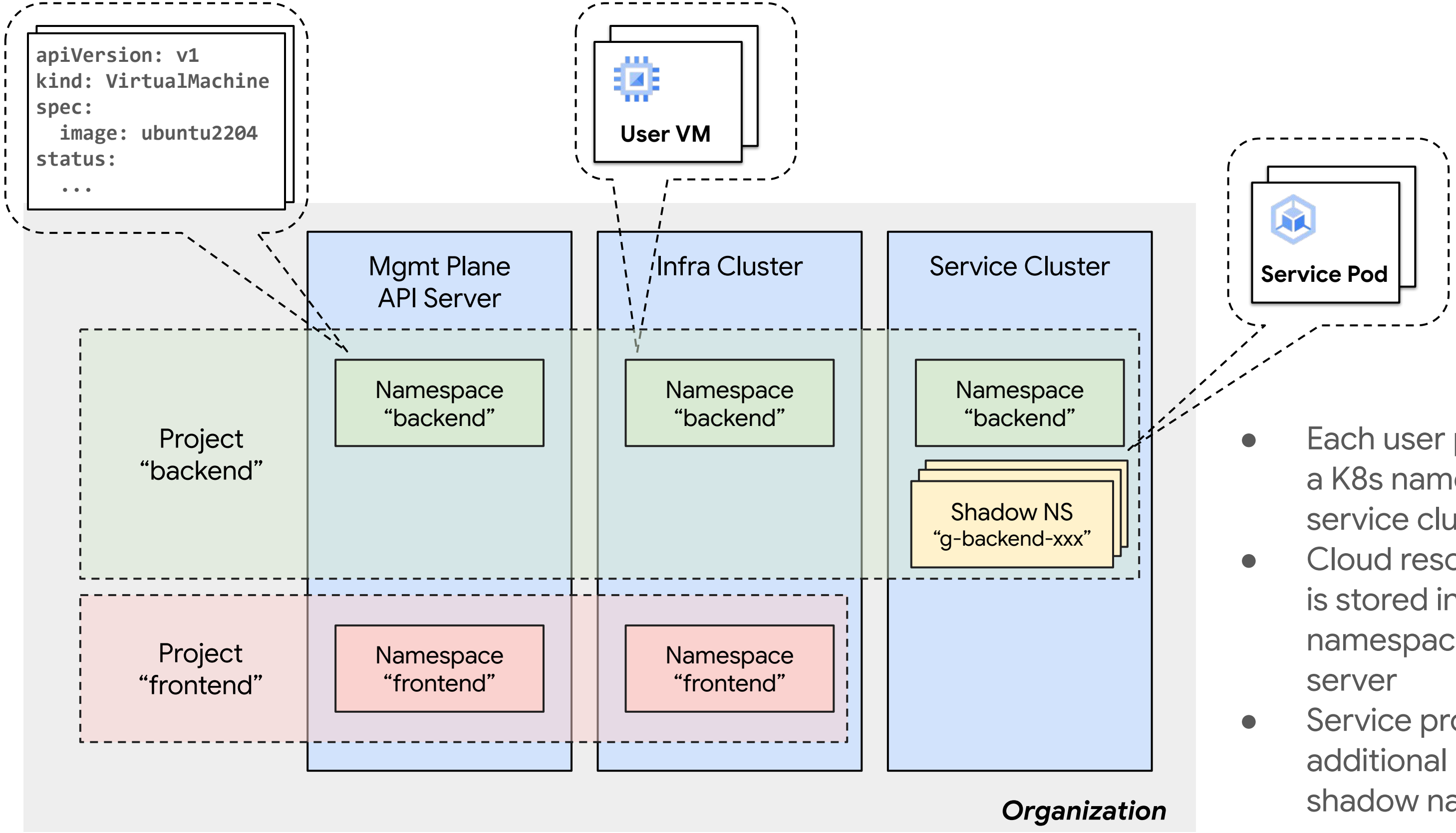
- Cluster service supports
  - **User Clusters.** End user facing Kubernetes clusters running on VMs
  - **Service Clusters.** Service producer owned Kubernetes clusters running on VMs

- Infra and service clusters not visible or accessible to end users

Namespace Sameness within An Organization

- Project maps to a namespace across clusters in an organization
- The namespace hosts cloud resources and workloads for the project spanning across multiple clusters with different types
  - E.g., VMs are in the bare metal infra cluster
- Consistent policies for that namespace across those clusters so that it feels like workloads are in the same cluster
  - E.g., administrative policies like IAM, project policies, and network policies
- Shadow namespaces for the project for service instances

Namespace Sameness



```
apiVersion: v1
kind: VirtualMachine
spec:
  image: ubuntu2204
status:
  ...
```

- Each user project is associated with a K8s namespace across infra and service clusters
- Cloud resources for the user project is stored in the corresponding K8s namespace management plane API server
- Service producers can request additional per service instance shadow namespace for data plane components

Design Principle #2

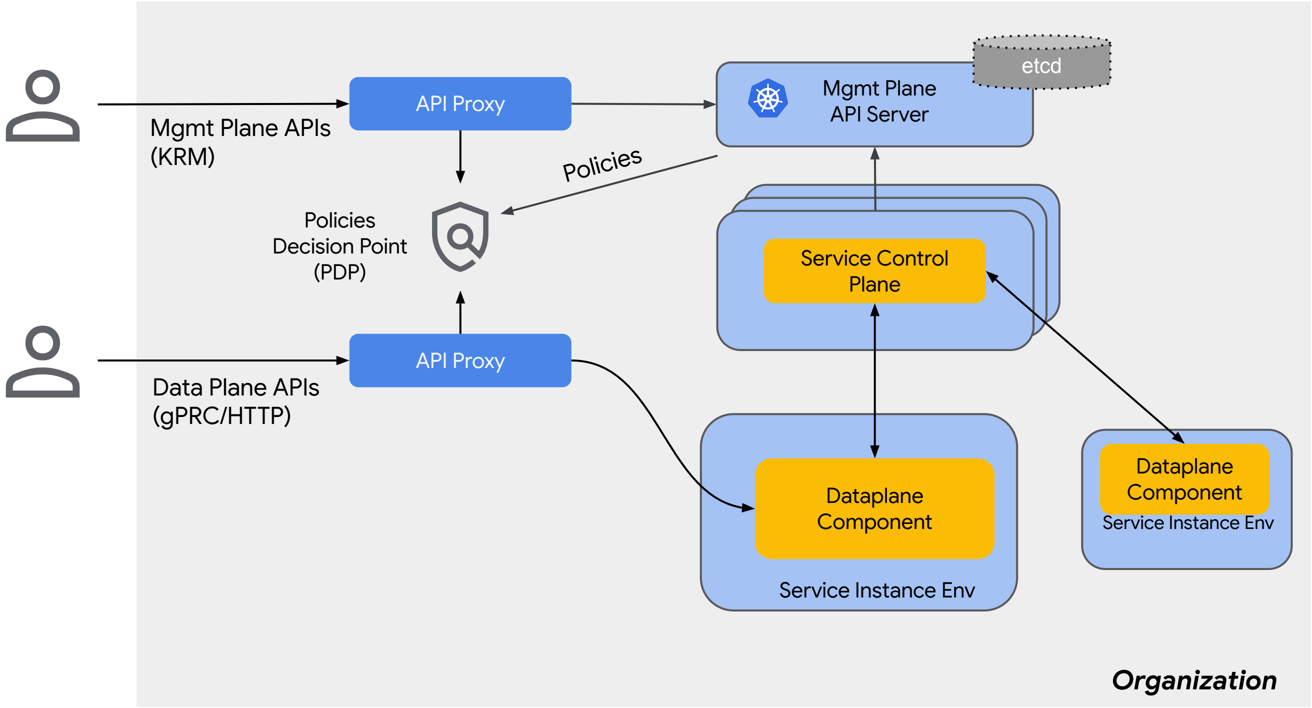
Leverage…

KRM-based API Machinery

and

Controller Pattern

API Platform and Controller Pattern

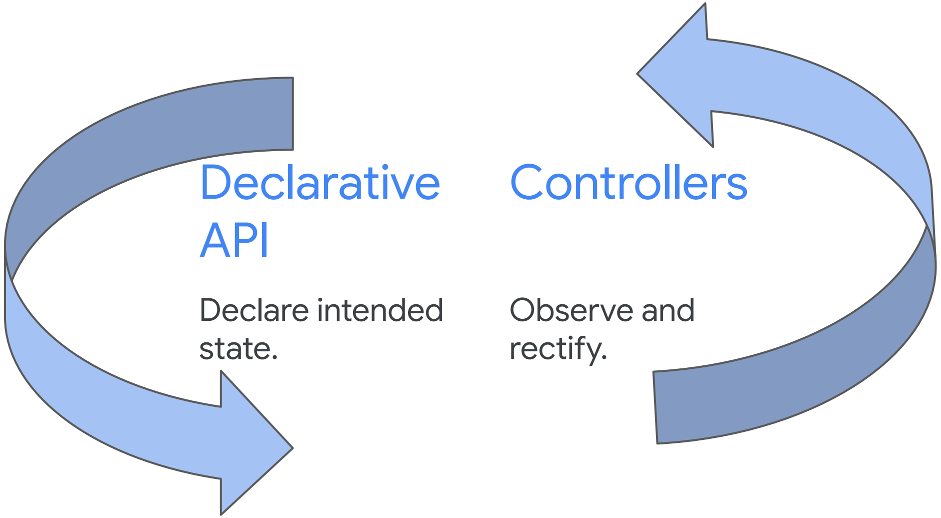


- **Management Plane APIs**
  - Follow declarative Kubernetes Resource Model (KRM)[1] style
- **Data Plane APIs**
  - Follow industry standard protocols if exists (e.g., OCI registry, SQL)
  - gRPC based Google APIs[2] for Google services
- **API Proxies**
  - Policy and horizontal enforcements
  - Policies stored in management plane API server

[1] https://github.com/kubernetes/design-proposals-archive/blob/main/architecture/resource-management.md
[2] https://google.aip.dev/

Controller Pattern

- Controller Pattern
  - All components work in parallel to drive the system to desired state
- Our system employs the Controller Pattern in all layers of the stack:
  - Managed 1p services
  - Internal infrastructure management
  - Hardware configuration
- Many benefits, most importantly **Automatic Recovery**



Declarative API

Declare intended state.

Controllers

Observe and rectify.

Design Principle # 3

Treat containers and VMs equally

Typical Networking Components in a Cloud

- **Virtual Networking Layer that plumbs workloads to the physical networking layer**
- **Expose workload related Networking knobs to the customer**

Connectivity

- IPAM for **VM and Container** workloads
- East-West Connectivity within a Project/Org
- North-South Connectivity to the external world
    - Ingress (LB)
    - Egress

Services

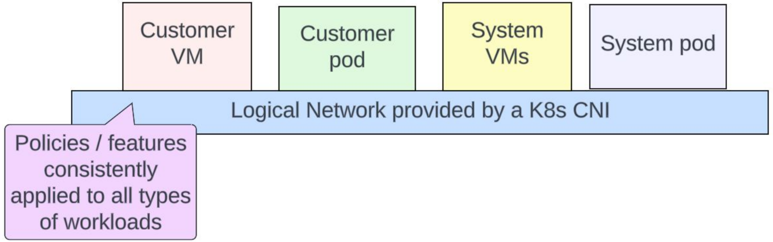
- L4 Loadbalancing
    - Internal to the organization
    - External to the organization
- L7 Loadbalancing (only for 1P services)
    - Internal to the organization
    - External to the organization
- DNS

Security (FW)

- Project Level Security
- Node Level Security
- Cluster Level Security

The Cloud we aspired towards…….

- Core Principle
  - ***Treat containers and VMs equally***
    - Consistent Policies and features for both VMs and Pods
    - Imagine a K8s service with both VMs and containers as backends…..
    - Imagine specifying networking security (Fw) using the same APIs for both VMs and containers….
    - Imagine exposing VM based or containerized First-Party Services using the same APIs…


Customer VM
Customer pod
System VMs
System pod
Logical Network provided by a K8s CNI
Policies / features consistently applied to all types of workloads

Imagine making it easier for customers to move to a truly cloud-native environment………

Introducing "Kubernetes Defined Networking (KDN)"

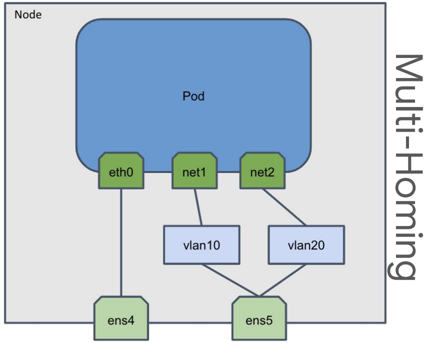
- Premise
  - Software Defined Networking (SDN) is typically used in the context of VMs
  - Kubernetes deployments involve orchestrating a network fabric across multiple kubernetes nodes **(built-in SDN)**
  - Kubernetes is an Ideal Platform to build an SDN stack to support VMs and containers
- What is KDN ?
  - A K8s based approach to define / deploy and manage a network fabric for both VM based and Containerized applications
  - A networking fabric orchestration system typically needs a management plane, control plane and a Dataplane.
  - In the KDN world,
    - Kubernetes Resource Model (KRM) based Management plane built around the K8s API server
    - K8s controllers based Control plane
    - K8s container Dataplane extended for VM networking support
  - Leverages the portable nature of Kubernetes to provide an SDN functional block on any platform

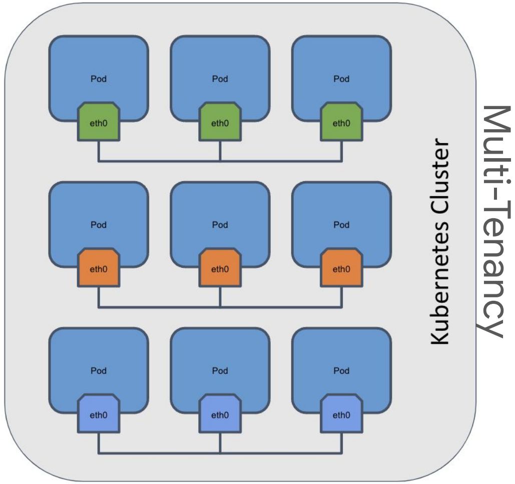
SDN vs KDN vs K8s Networking

SDN	KDN	K8s Networking
Focusses on VMs	Treat containers and VMs equally	Focusses on Containers
Many variations in API depending on the Technology	Consistent API surface	Consistent API surface
N/W Isolation Constructs are available	Native N/W Isolation is supported (multi-homing use case)	Not Available
Vendor specific NF integration	Both vendor agnostic and vendor specific NF integration	Both vendor agnostic and vendor specific NF integration
Feature rich n/w fabric for fault tolerant architecture	Feature rich n/w fabric for fault tolerant architecture	Feature richness across fault tolerant boundaries (Multi-Cluster) is lacking
Isolation via N/W multitenancy	Available through Multi-Networking; abstracted as Projects and VPC boundaries	Not Available

OK...I am hooked, I want KDN.... What next...

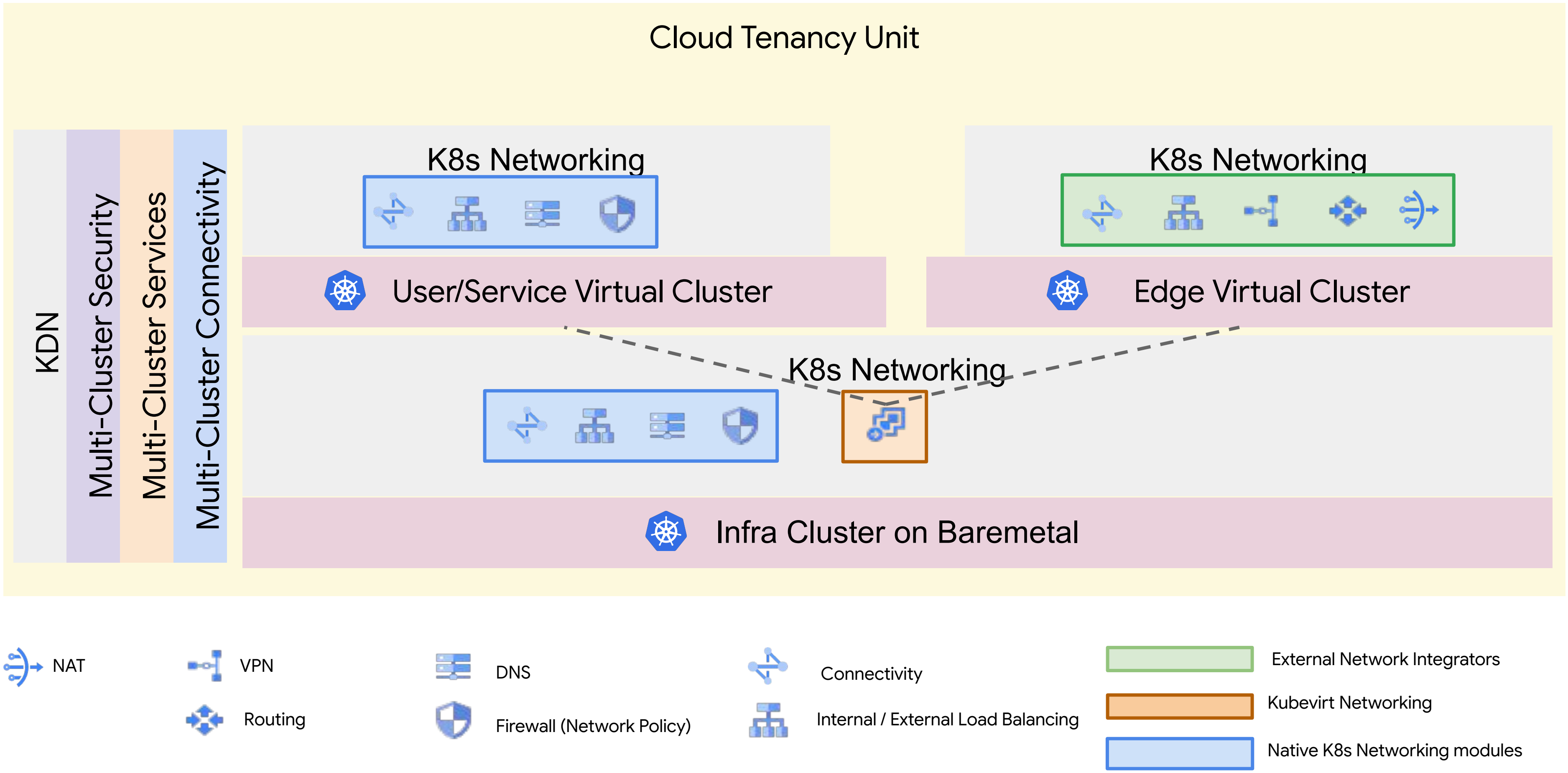
- Learnings from building a cloud network on a K8s substrate
  - Native Network API in K8s
  - Multi-homing Use-cases
  - Multi-tenancy Use-cases
  - Networking Isolation in both Physical and Virtual environments
- Effort to bring such APIs / Functionality to K8s
  - K8s Multi-Networking Workgroup
  - Kubecon 2023 Presentation





How does one build all this.... ?

Network Layering


Cloud Tenancy Unit
KDN
Multi-Cluster Security
Multi-Cluster Services
Multi-Cluster Connectivity
K8s Networking
User/Service Virtual Cluster
K8s Networking
Edge Virtual Cluster
K8s Networking
Infra Cluster on Baremetal
NAT
VPN
Routing
DNS
Firewall (Network Policy)
Connectivity
Internal / External Load Balancing
External Network Integrators
Kubevirt Networking
Native K8s Networking modules

Takeaways: Building a Cloud with CNCF Ecosystem

How did CNCF Ecosystem help?

- Don’t have to start from scratch.
    - Don’t need to right many key components: e.g. k8s API machinery, etc..

How did CNCF Ecosystem get in the way?

- Lack of rich multi-cluster support in OSS; assumption single-cluster
- VMs are not yet well supported in ecosystem

Q&A

KubeCon | CloudNativeCon

North America 2024