BUILDING FOR THE ROAD AHEAD

DETROIT 2022

Orchestrating Interconnected Apps Across Geographically Distributed Kubernetes Clusters
Introducing Nephio

John Belamaric
Sr Staff Software Engineer, Google
Thursday, October 27 2022

Orchestrating Interconnected Apps Across Geographically Distributed Kubernetes Clusters

Introducing Nephio

KubeCon | CloudNativeCon

North America 2022

BUILDING FOR THE ROAD AHEAD

DETROIT 2022

KubeCon | CloudNativeCon North America 2022

BUILDING FOR THE ROAD AHEAD

DETROIT 2022

Thursday, October 27
3:25pm - 4:00pm

https://sched.co/182H0

John Belamaric

Sr Staff Software Engineer

Google

Who am I?

- Senior tech leader in Anthos and GKE at Google Cloud
- Nephio SIG Automation Chair
- Kubernetes SIG Architecture Co-chair
- Emeritus approver of Kubernetes SIG Network
- Led development of CoreDNS/Kubernetes integration
- Led development of pluggable IPAM in OpenStack Neutron
- Former tech lead / architect of a successful commercial enterprise network automation product
  - Automates configuration of tens of thousands of switches, routers, and other devices

Imagine deploying a set of complex, interconnected workloads across an array of geographically distributed sites.

cloud.google.com/about/locations

So. Many. Problems.

Do we run clusters in every site, or clusters that span sites?

How do you decide where to run each workload?

How do we specialize the configs for each site?

How do we make sure those configs conform to our policies?

How do we deliver the configs to the right clusters, and make sure they don't drift?

What happens when we add a site - how do we know which interconnected workloads need to be reconfigured?

How do we know what to change in each of those workloads?

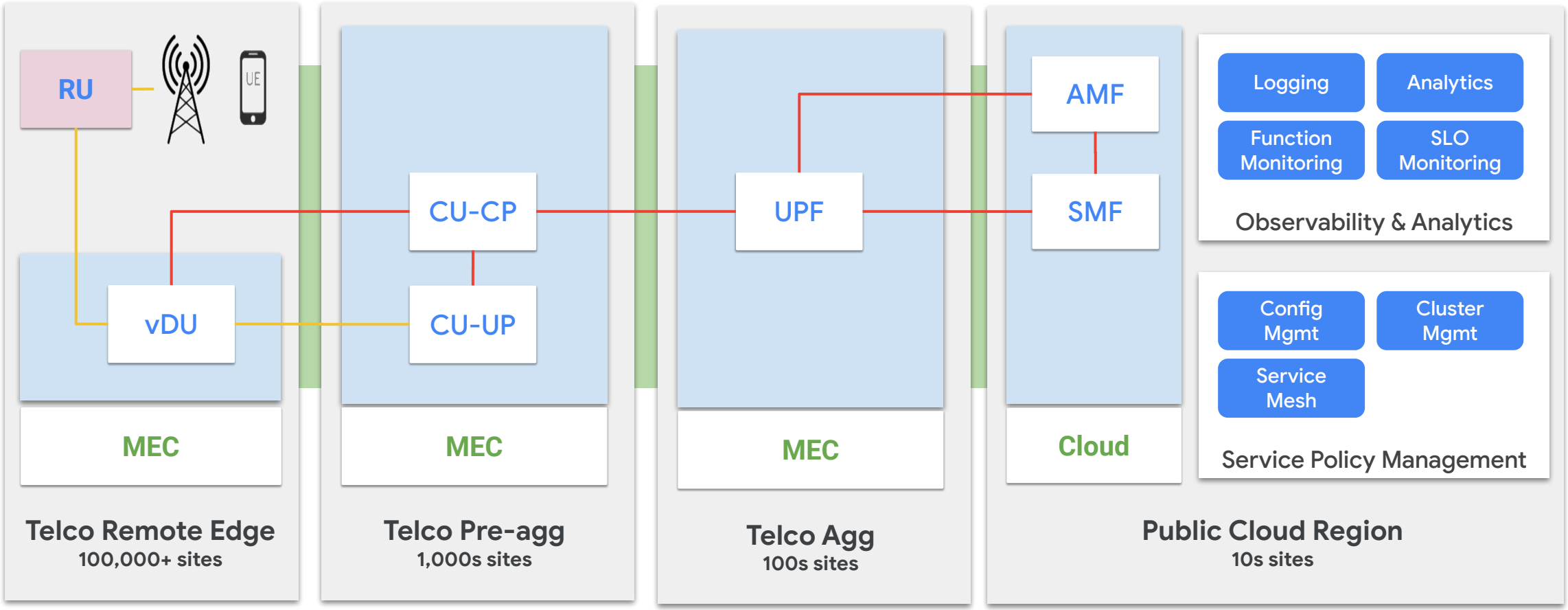
Do we just need to change Kubernetes manifests, or do the configuration files of the workloads themselves need to be changed?

How do we upgrade workloads?

How do we enforce policies?

…and on, and on…

Simplified / Minimal 5G Network



A Simple Request

I want a private 5G core in the Northeastern US for my fleet of 20,000 trucks.
Bandwidth use will be low since this is only used for tracking status reports and occasional voice.
Mid-quality latency is acceptable.

What does it take to roll this out? ***Some*** of it, for **Day 1**:

- Identify the sites - edge and cloud regions - that are available and applicable
- Determine which workloads should run where and what to connect to what
- Determine the infrastructure needed - the clusters and nodes
- Configuring the networking between sites, allocate subnets, IPs, VLAN tags, VRFs, etc.
- Configuring the underlying nodes for specialized telco requirements
- Configuring the workload specifications - the basic Kubernetes manifests for them
- Configuring the workloads themselves to know about each other.

Complexity, Complexity, Complexity

Day 2 adds more complexity:

- Monitor that the stated intent is still expressed
  - Workloads are up and running
  - Configuration hasn't drifted
- Handle changes to topology
  - Spin up a new aggregation site, adding a UPF
  - UPF needs to talk to an SMF
  - Each of these needs to be configured to see each other
- Resize workloads as topology changes
  - As we add UPFs, we need to vertically scale the SMF.
- Enable upgrade of workloads and infrastructure with progressive rollout

It Gets Worse…

Each layer is managed by different systems!

- Topology: Manually encoded in powerpoint slides and spreadsheets
  - Maybe end-to-end orchestration workflows
- Cloud infrastructure: Cloud Provider APIs
  - Maybe Terraform, scripts, or e2e orchestration
- Networking within and between sites: manual router configuration
  - Maybe some automation via vendor or other proprietary technologies
- Nodes: K8s extensions, manual or scripted kernel and other node configs
- Workload specifications: stored in Kubernetes manifests, maybe in Git or scripts
- Workloads configs: proprietary, vendor-specific network element managers

What do we do? Where do we start?

Reduce Complexity

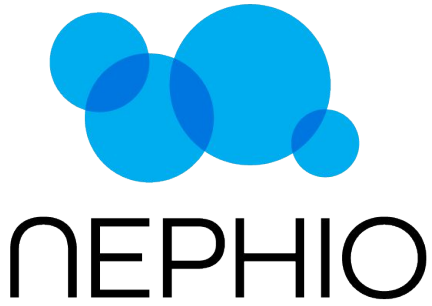
- Consolidate on a **single, unified platform for automation**
  - Across infrastructure, workloads, workload configs, vendors and deployment tiers.
- **Declarative configuration with active reconciliation** to support days one and two.
  - And distribute state (intent) across geography for resilience
- Configuration that can be **cooperatively managed** by machines and humans.
  - Machine-manipulable configuration is fundamental to automation.

Kubernetes Everywhere

- **Uniform automation tooling** for topology, infra, workloads, and workload config
  - No out-of-band changes!
- Foundation for **intent-based, declarative management** with active reconciliation
- Widely adopted and understood
- Existing K8s-based point solutions in many of the layers
- Support for custom schemas and controllers
- Strong extensibility
- Rich ecosystem

Configuration-as-Data (CaD)

- A new approach to configuration management
  - Represent config in a well-defined, structured data model (KRM!)
  - Configuration lives in versioned storage, separate from the live state
  - Tools operate on the config - do not intermingle code and configuration
  - Clients interact with config via APIs, not directly on storage
- Machine manageable configurations
- Enables iterative, multi-actor workflows to operate and validate configurations
- Automated changes, bulk operations, and human-initiated modifications co-exist peacefully
- Automatic system validation of configuration before applying to live state
- Reusable, well-tested functions operate on configuration rather than embedding code inside the configuration
- Implemented in open source projects kpt, Porch, and Config Sync



The Linux Foundation and Google Cloud Launch Nephio to Enable and Simplify Cloud Native Automation of Telecom Network Functions

New Open Source Project at the LF brings Cloud, Telecom and Network functions providers together in a Kubernetes world

San Francisco—April 12, 2022 Today, the Linux Foundation, the nonprofit organization enabling mass innovation through open source, announced the formation of project Nephio in partnership with Google Cloud and leaders across the telecommunications industry. The Linux Foundation provides a venue for continued ecosystem, developer growth and diversity, as well as collaboration across the open source ecosystems.

Mission Statement

Nephio’s goal is to deliver carrier-grade, simple, open, Kubernetes-based cloud-native intent automation and common automation templates that materially simplify the deployment and management of multi-vendor cloud infrastructure and network functions across large scale edge deployments. Nephio enables faster onboarding of network functions to production including provisioning of underlying cloud infrastructure with a true cloud native approach, and reduces costs of adoption of cloud and network infrastructure.

Very High Level Nephio Architecture

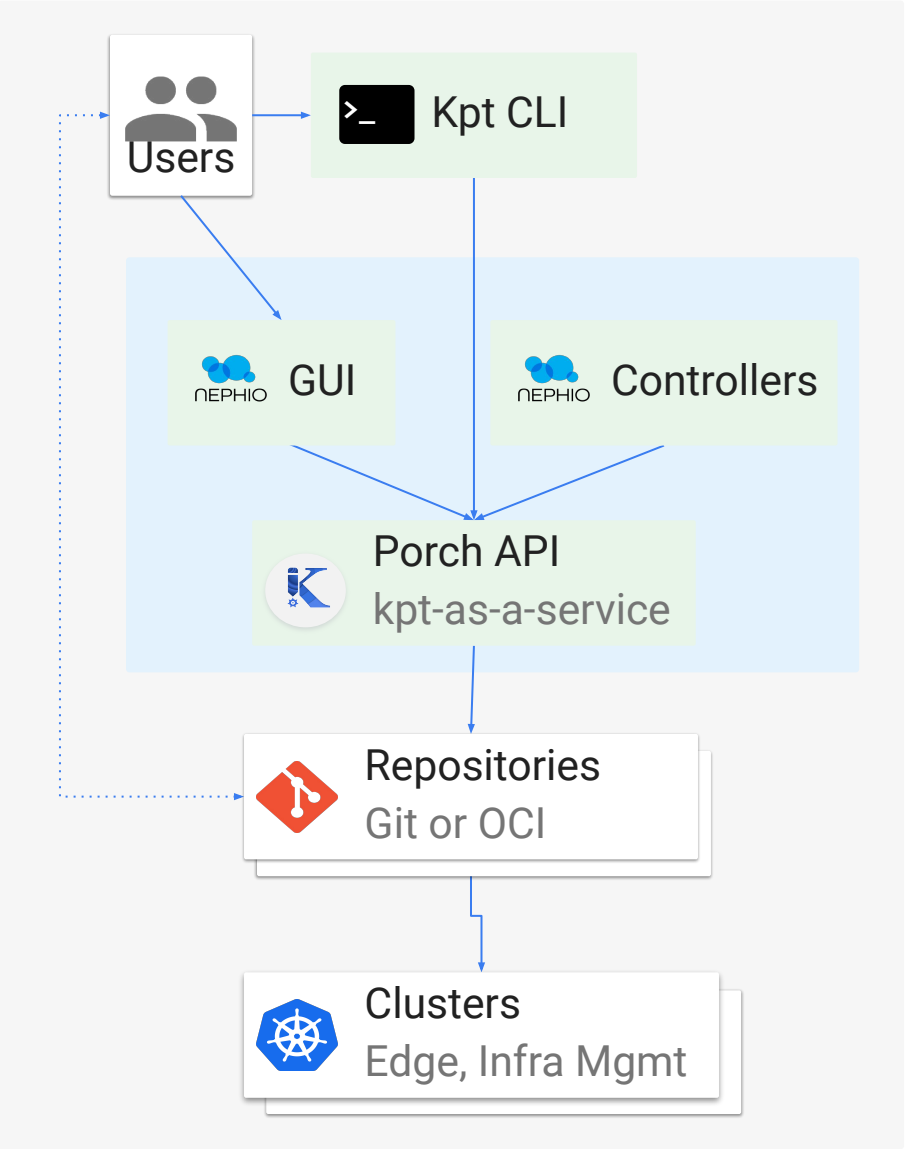
Platform enabling users and automation agents to cooperatively interact with and deploy configuration

Central Nephio K8s cluster houses GUI service, Porch APIs, and Nephio Controllers

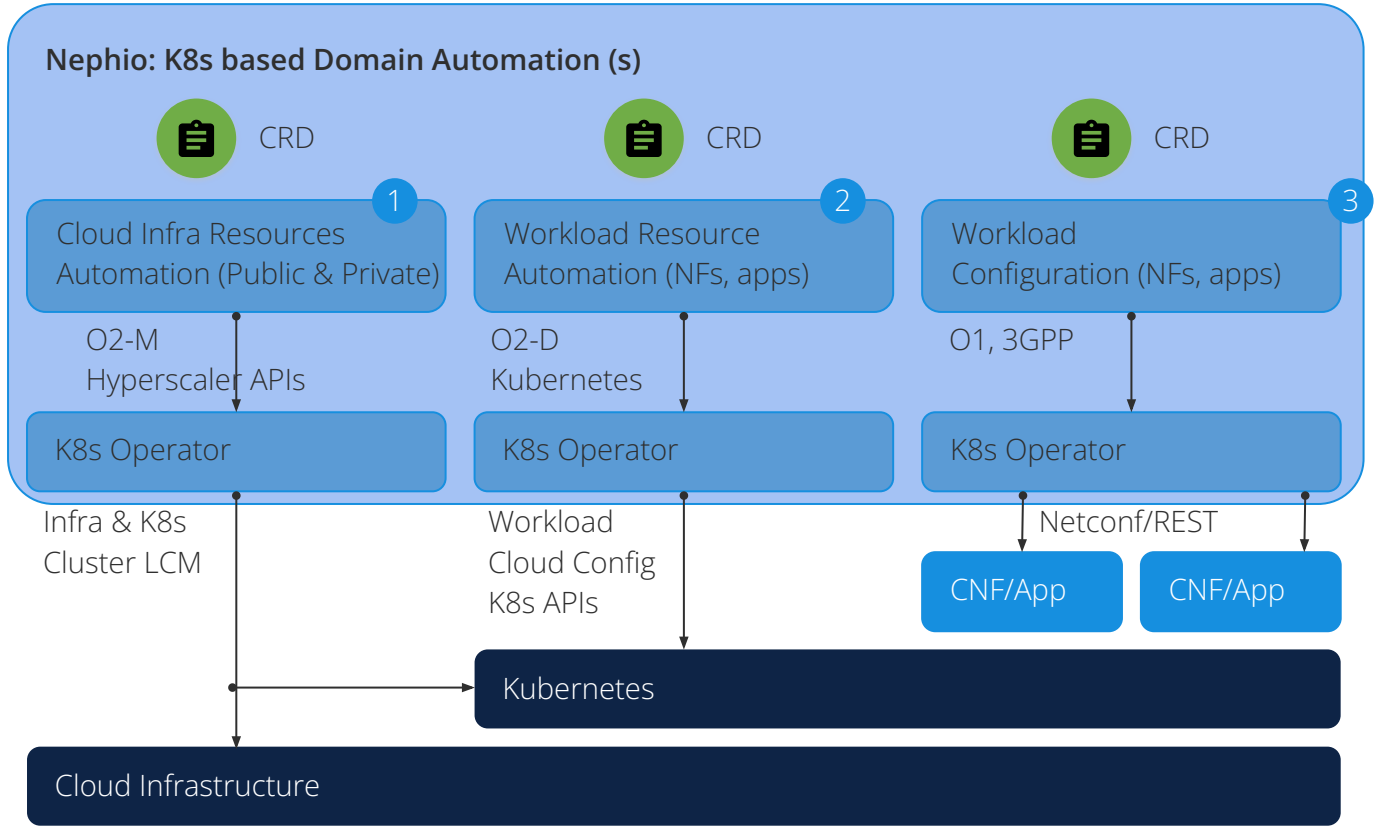
Users manipulate config packages which Porch pushes to Git or OCI repositories

Downstream clusters consume those via Config Sync, which applies them to the K8s API server

Same process for infrastructure - using for example KCC to create clusters, as for workloads



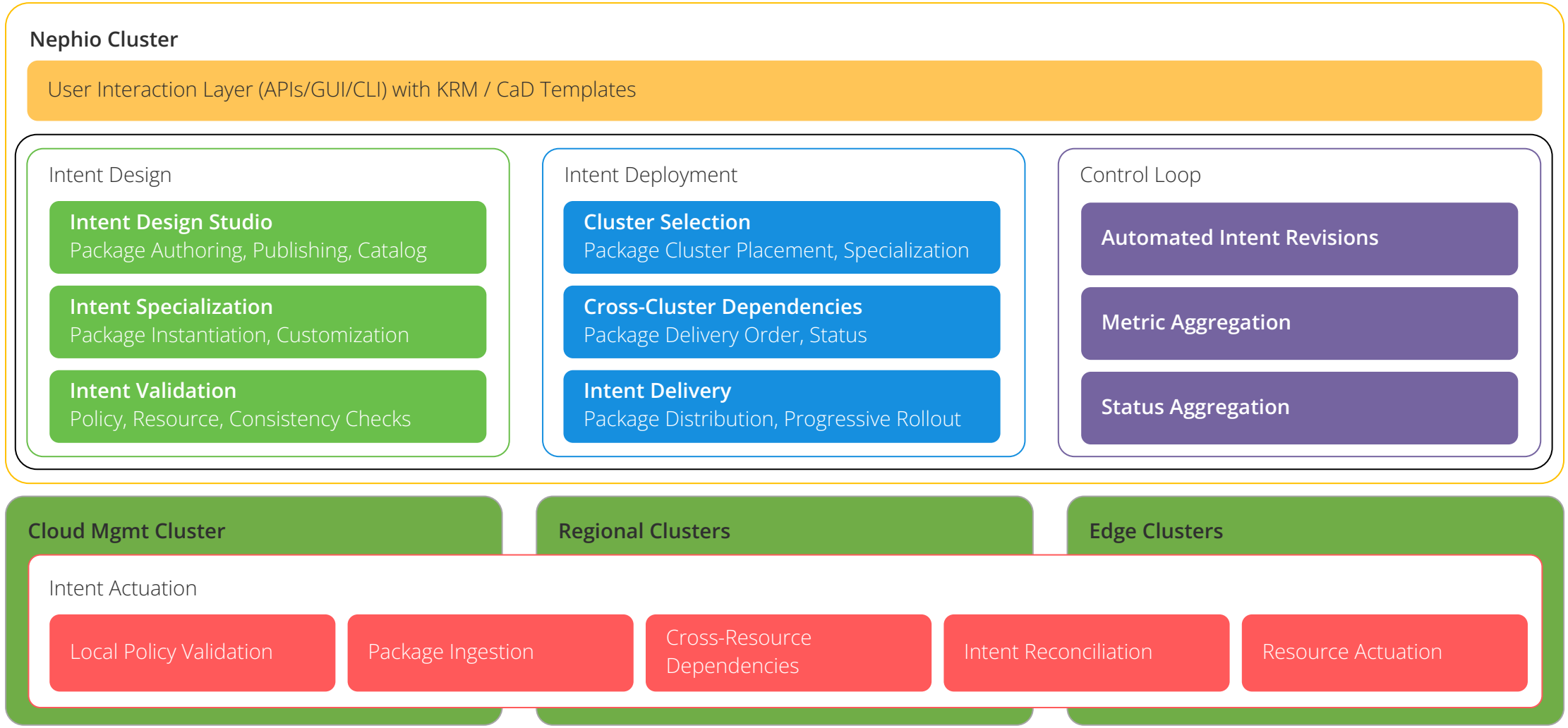
Layers of Configuration



Scope of Nephio community

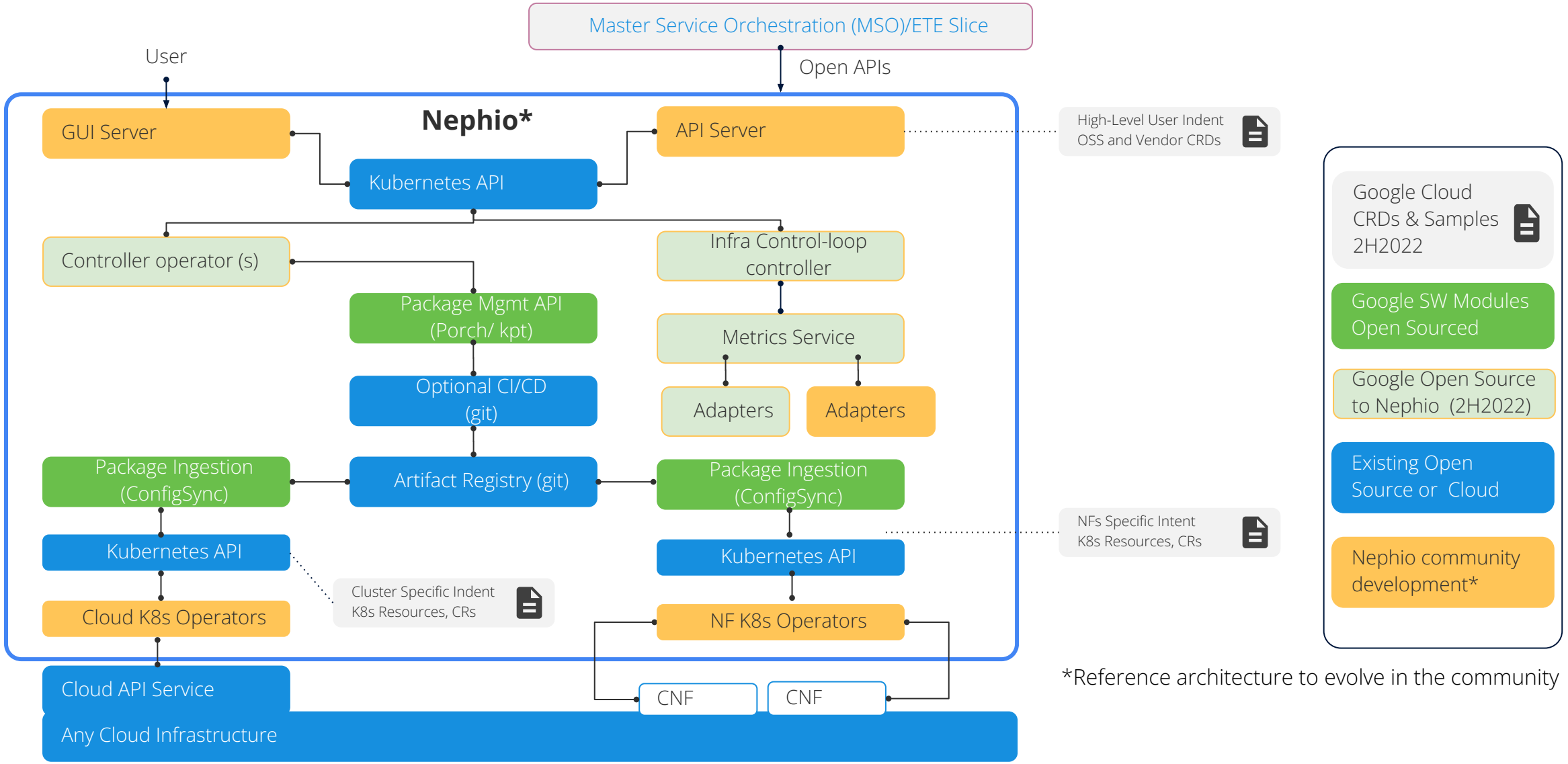
- K8s-based automation for each public and private cloud infrastructure automation
- The workload cloud resource automation (i.e CRUD operations of K8s Cluster, network functions/app deployment on top of the cloud, and NF/app infrastructure configuration such as SR-IOV)
- Workload configuration (ie. NF/app level configuration)

Nephio Functional Building blocks



KRM - Kubernetes Resource model
CaD - Configuration as Data

Nephio High-Level Architecture



*Reference architecture to evolve in the community

Nephio Configuration Layers

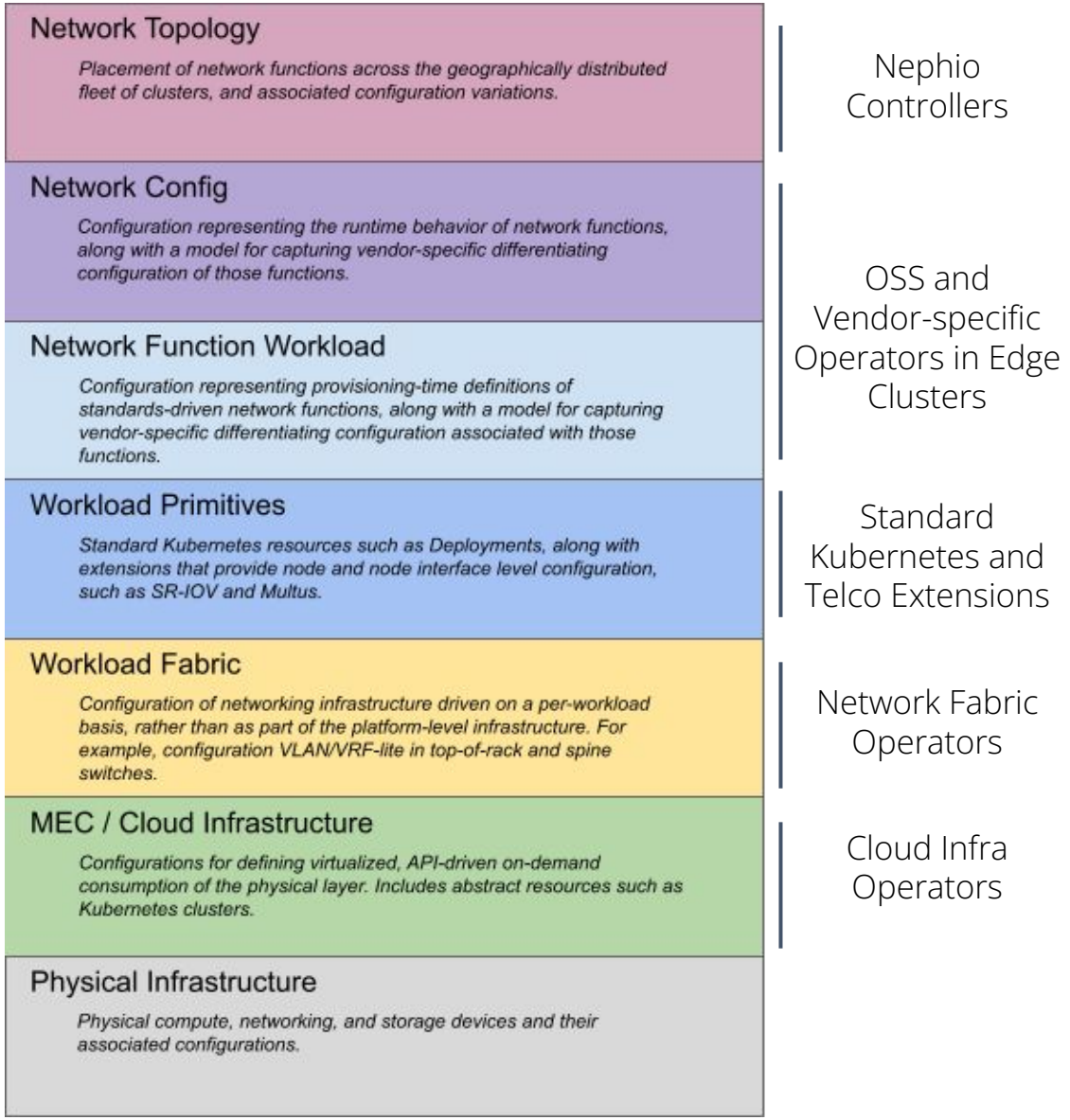
Capture the infrastructure - sites, cluster, nodes, cloud infra

Capture the workloads, their interconnections, and the relationships to the infrastructure

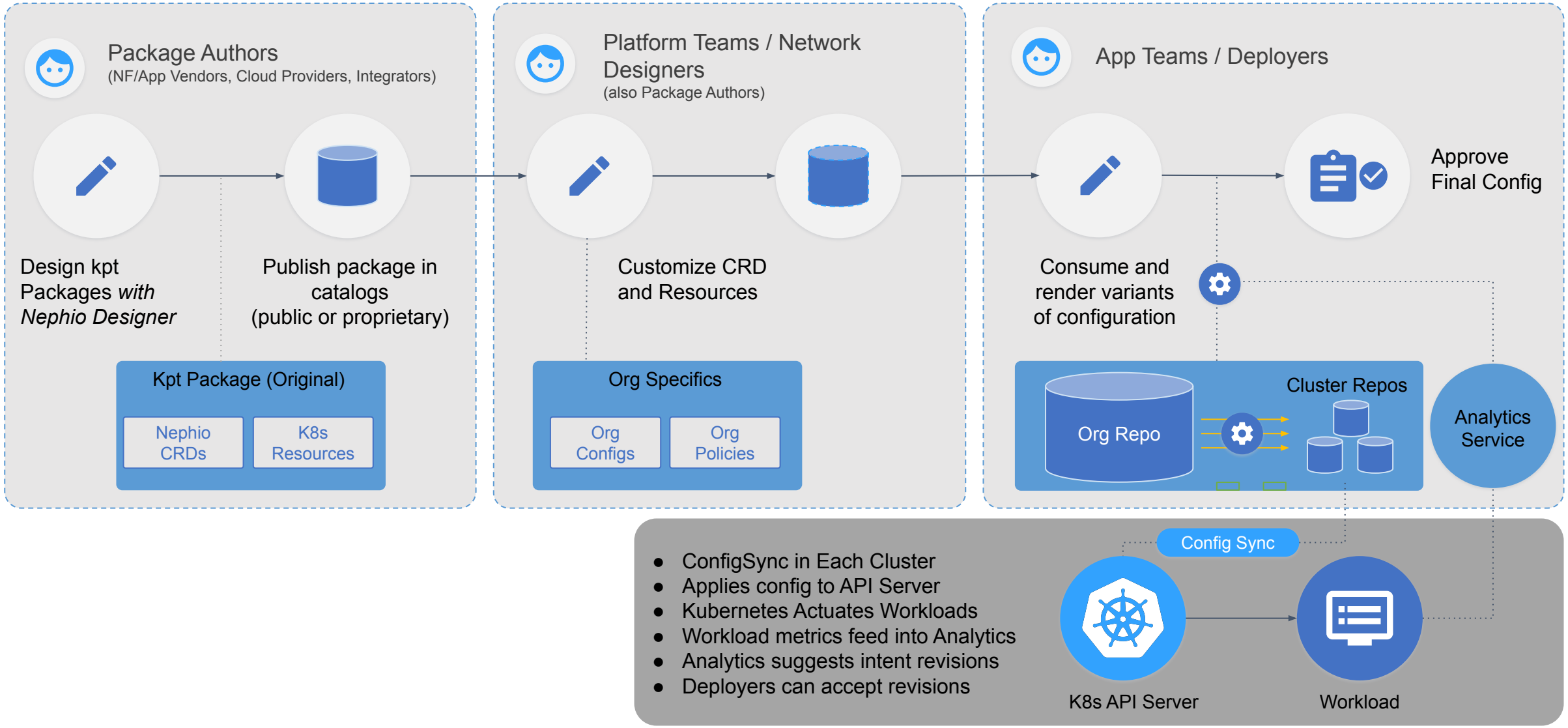
Capture the common concepts across types of workloads

Enable vendor differentiation for those workloads

Different layers may be actuated by different operators running at different levels in the stack



Nephio End-to-End Journey



- ConfigSync in Each Cluster
- Applies config to API Server
- Kubernetes Actuates Workloads
- Workload metrics feed into Analytics
- Analytics suggests intent revisions
- Deployers can accept revisions

Join Us!

KubeCon CloudNativeCon North America 2022

Upcoming Talks/Events

- Nephio co-located event at ONE Summit, Seattle Nov 15-16

Project Resources

- Website - https://nephio.org/ , https://nephio.org/about/
- Wiki - https://wiki.nephio.org/
- Blog Postings - https://nephio.org/blog/
- Project Github - https://github.com/nephio-project (Please note “nephio-project is right one”)
- Project email distro - https://lists.nephio.org
  - nephio-tsc (for TSC members and interested parties)
  - nephio-dev (for all)
  - SIG lists: sig-netarch, sig-automation, sig-release

Kpt, Porch, and Config Sync

- Main Site: https://kpt.dev
- Kpt and Porch Repository: https://github.com/GoogleContainerTools/kpt
- Config Sync Repository: https://github.com/GoogleContainerTools/kpt-config-sync
- Contribute: https://github.com/GoogleContainerTools/kpt/blob/main/CONTRIBUTING.md

KubeCon | CloudNativeCon
North America 2022

BUILDING FOR THE ROAD AHEAD

DETROIT 2022

Please scan the QR Code above to
leave feedback on this session

Backup Slides

Two presentations from the June 2022 Nephio Summit.

How Config as Data Enables Automation, *Brian Grant*

Nephio Reference Implementation, *John Belamaric*

Recording links are also available on the title pages.

How Config as Data Enables Automation

Brian Grant, Distinguished Engineer, Google Cloud

[Recording from Nephio Summit, June 22, 2022]

Who am I?

- Overall tech lead of declarative configuration at Google Cloud
- Former tech lead of the control-plane of Google's internal container platform
- Original lead architect and API design lead of Kubernetes
- Coined "Kubernetes Resource Model" (KRM)
- Creator of kubectl apply and kustomize
- Author of the CNCF definition of Cloud Native

Agenda

- The Kubernetes Resource Model (KRM)
- Helm Configuration Example
- Kustomize
- GitOps
- Kpt: Configuration as Data
- Demo

Three aspects for optimizing automation

Intent-based automation

Simplified configuration to user
e.g. Deploy 5G UPF with X capacity at Y location and do Z when this event occurs

Declarative configuration

To address day 0, 1 and 2 configurations, rainy day scenarios, intelligent auto scaling control-loops, and full life-cycle support

Uniform control plane

Simplified, unified cloud native management (Kubernetes) in every tier

Extend base Kubernetes with Infrastructure CRDs and Operators

- Declarative expression of ALL infrastructure requirements for NFs
- Beyond the Pod, to Node
- Beyond the Node, to ToR

Deploy a function anywhere

- No out-of-band infrastructure configuration

Kubernetes is Declarative

Express intent...

I'm thirsty...
I want soda in my blue cup.

Sure! Thanks for trusting me to figure out how to get you a soda.

Declarative: User Friendly

...without the toil of "how"

I'm thirsty...
Go to the kitchen
Open the fridge
Reach in the back to get the soda
Close the fridge
Grab a blue cup from the left cupboard
Pour out the soda in the cup
Bring the cup back to me

Ummm... okay thanks for telling me exactly what to do...

Imperative : Hardship to user

What is cloud native?

Cloud-native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust **automation**, they allow engineers to make high-impact changes frequently and predictably with **minimal toil.**

The Kubernetes Resource Model (KRM)

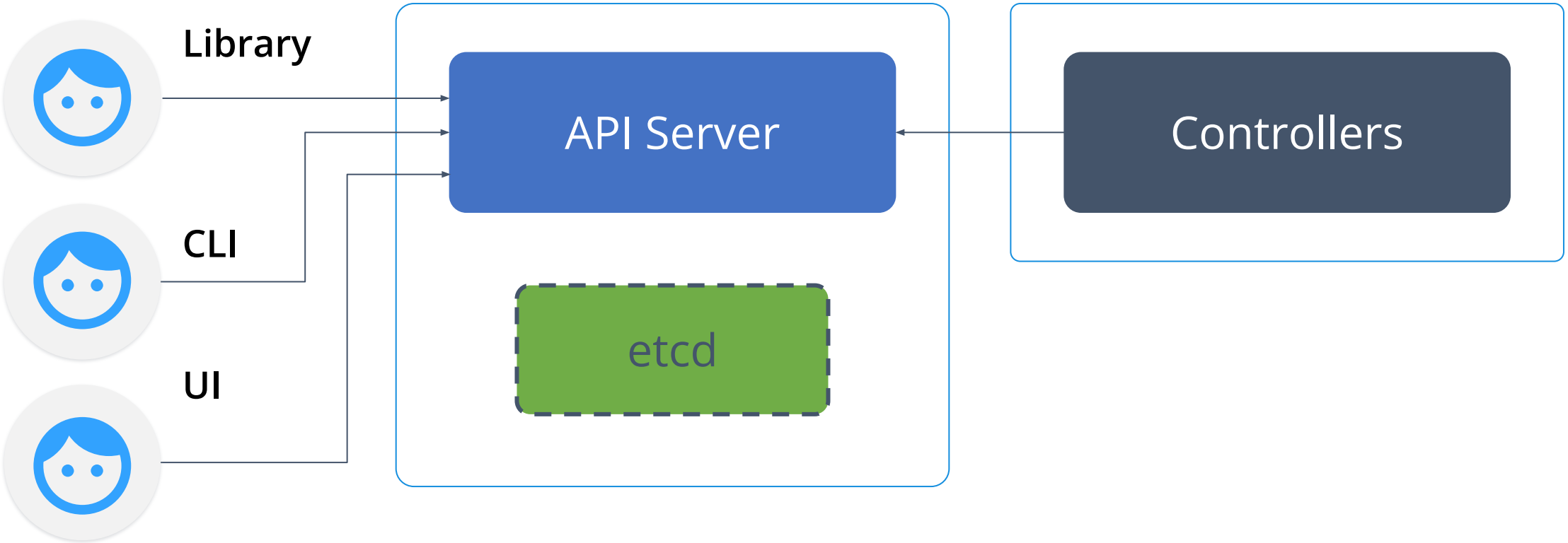
KRM: Desired State

spec:

- Specifies **desired state**
- Includes user intent
- Also includes automatically set attributes
  - default values
  - allocated values
  - policy-imposed values
  - dynamic values
  - …
- Clearly separated from observed state (**status**)

```
apiVersion: v1
kind: Pod
metadata:
 namespace: default
 name: explorer
 labels:
   category: demo
 annotations:
   commit:
483ac937f496b2f36a8ff34c3b3ba84f70ac5782
spec:
 containers:
   - name: explorer
     image:
gcr.io/google_containers/explorer:1.1.3
     args: ["-port=8080"]
     ports:
       - containerPort: 8080
         protocol: TCP
```

Kubernetes Desired State is Managed Through its API



KRM: Uniform Structure Data

Consistent & self-describing:

- API group and version
- (e.g., apps, network)
- Resource type
- (e.g., Pod, Service, ReplicaSet)
- Namespace (scope)
- Name (idempotent handle)
- Key-value Labels
- (user categorization and grouping)
- Key-value Annotation data
- (system and tool extensions and checkpoints)

```
apiVersion: v1
kind: Pod
metadata:
 namespace: default
 name: explorer
 labels:
   category: demo
 annotations:
   commit:
483ac937f496b2f36a8ff34c3b3ba84f70ac5782
spec:
 containers:
   - name: explorer
     image:
gcr.io/google_containers/explorer:1.1.3
     args: ["-port=8080"]
     ports:
       - containerPort: 8080
         protocol: TCP
```

Kubernetes Configuration is KRM

Configuration in storage uses the identical representation as the Kubernetes API

```
$ kubectl apply -f krm/
```

For arbitrary resource types, regardless how the configuration was produced

```
{{- range .roleBindings }}
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: {{ .name }}
  namespace: {{ .namespace }}
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: {{ .roleKind }}
  name: {{ .role }}
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: {{ .namespace }}.admin@myco.com
{{- end }}
```

Configuration is often generated from templates

Templates

- Facilitates generation of similar instances
- Due to the demand for flexibility, templates often become fully parameterized struct constructors

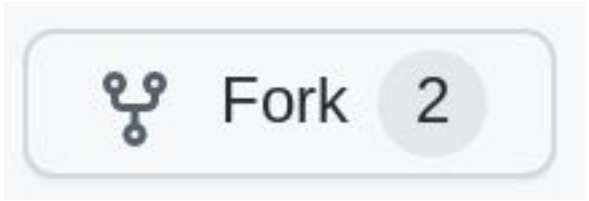
Helm Configuration Example

Simple Scenario

- Create 3 Namespaces with ServiceAccounts and RoleBindings

- Using an existing Helm chart

Fork the Repo



- And then set up repo permissions, branch protections, PR status checks, etc.
- Changes to the configuration will affect your clusters

Clone Repo

```
$ git clone https://github.com/bgrant0607/kube-common-setup
Cloning into 'kube-common-setup'...
remote: Enumerating objects: 30, done.
remote: Counting objects: 100% (30/30), done.
remote: Compressing objects: 100% (23/23), done.
remote: Total 30 (delta 12), reused 24 (delta 7), pack-reused 0
Receiving objects: 100% (30/30), 8.51 KiB | 2.13 MiB/s, done.
Resolving deltas: 100% (12/12), done.
```

Initial Setup and Create a Branch for Patches

```
$ cd kube-common-setup/

$ git remote add upstream \
  https://github.com/nghnam/kube-common-setup

$ git remote set-url --push upstream no-push

$ git checkout -b patched
```

Add Missing Attributes

```
$ cd templates/
$ vi namespaces.yaml
# add annotations
```



```
{{- range .Values.namespaces }}
---
apiVersion: v1
kind: Namespace
metadata:
  name: {{ .name }}
  {{- if .labels }}
  labels:
    {{- range $key, $value := .labels }}
    {{ $key }}: {{ $value | quote }}
    {{- end }}
  {{- end }}
  {{- if .annotations }}
  annotations:
    {{- range $key, $value := .annotations }}
    {{ $key }}: {{ $value | quote }}
    {{- end }}
  {{- end }}
{{- end }}
```

Update Out of Date Content, and Change an Opinion

```
$ vi role-bindings.yaml
# update API version
# change SA namespace
```

```
{- range .Values.serviceAccounts }}
{{- $saname := .name -}}
{{- range .roleBindings }}
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: RoleBinding
metadata:
  name: {{ $saname }}-{{ .name }}
  namespace: {{ .namespace }}
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: {{ .kind }}
  name: {{ .name }}
subjects:
- kind: ServiceAccount
  name: {{ $saname }}
  namespace: {{ .namespace }}iam
{{- end }}
{{- end }}
```

Check that the rest is Ok

```
$ cat service-accounts.yaml
```

```
{{- range .Values.serviceAccounts }}
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: {{ .name }}
  namespace: {{ .namespace }}
{{- end }}
```

Commit and Tag

```
$ git commit -a -m “add annotations, fix RoleBinding”
[patched 19d82bb] add annotations, fix RoleBinding
 2 files changed, 9 insertions(+), 3 deletions(-)
$ git tag my-kube-setup/v0.1
```

Push

```
$ git push origin patched
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 8 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 485 bytes | 485.00 KiB/s, done.
Total 5 (delta 4), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (4/4), completed with 4 local objects.
remote:
remote: Create a pull request for 'patched' on GitHub by visiting:
remote:      https://github.com/bgrant0607/kube-common-setup/pull/new/patched
remote:
To https://github.com/bgrant0607/kube-common-setup
 * [new branch]      patched -> patched

$ git push --tags
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/bgrant0607/kube-common-setup
 * [new tag]         my-kube-setup/v0.1 -> my-kube-setup/v0.1
```

Create a Repo for Values

- And then set up repo permissions, branch protections, PR status checks, etc.
- Changes to the configuration will affect your clusters

Clone Repo

```
$ git clone \
  https://github.com/bgrant0607/ns-values
Cloning into 'ns-values'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (4/4), 4.49 KiB | 4.49 MiB/s, done.
```

Create Values File

```
$ cd ns-values/
$ vi values.yaml
```

```
# My parameter values

# What should go here?
```

Look at the Example Chart Values File

```
namespaces:
  - name: iam # used to store all service accounts
  - name: istio-system # used to install istio
  - name: project-a
    isolated: true
    labels:
      istio-injection: enabled
    quota:
      requests:
        cpu: 1
        memory: 1Gi
        storage: 15Gi
      limits:
        cpu: 2
        memory: 2Gi
      persistentVolumeClaims: 5
  - name: project-b
    isolated:
    quota:
      requests:
        cpu:
        memory:
        storage:
      limits:
        cpu:
        memory:
      persistentVolumeClaims:
    dockerconfigjson: |
      {
        "auths": {
          "custom-registry.local": {
            "username": "admin",
            "password": "custom-registry-password"
          }
        }
      }
  - name: project-c
```

```
serviceAccounts:
  - name: namnh
    namespace: iam
    roleBindings:
      - namespace: project-a
        kind: ClusterRole
        name: admin
      - namespace: project-b
        kind: ClusterRole
        name: edit
    clusterRoleBindings:
      - kind: ClusterRole
        name: namespace-reader
  - name: gitlab-admin
    namespace: kube-system
    roleBindings:
    clusterRoleBindings:
      - kind: ClusterRole
        name: cluster-admin

privateRegCred: &privateRegCred |
  {
    "auths": {
      "private-registry.local": {
        "username": "admin",
        "password": "private-registry-password"
      }
    }
  }
```

To figure out what this means, I more or less had to read all the templates

Check the rest of the templates

```
{{- range .Values.serviceAccounts }}
{{- $saname := .name -}}
{{- range .clusterRoleBindings }}
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: {{ $saname }}-{{ .name }}
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: {{ .kind }}
  name: {{ .name }}
subjects:
- kind: ServiceAccount
  name: {{ $saname }}
  namespace: iam
{{- end }}
{{- end }}
---
{{- /* Created cluster roles */ -}}
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: namespace-reader
rules:
- apiGroups: [""]
  resources: ["namespaces"]
  verbs: ["get", "watch", "list"]
```

```
{{- range .Values.namespaces }}
{{- if .isolated }}
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  namespace: {{ .name }}
  name: isolate-namespace
spec:
  podSelector:
    matchLabels:
  ingress:
  - from:
    - podSelector: {}
{{- end }}
{{- end }}
---
{{- range .Values.namespaces }}
{{- $ns := .name -}}
{{- if .dockerconfigjson }}
---
apiVersion: v1
kind: Secret
metadata:
  name: regcred
  namespace: {{ $ns }}
data:
  .dockerconfigjson: {{ .dockerconfigjson | b64enc }}
type: kubernetes.io/dockerconfigjson
{{- end }}
{{- end }}
```

```
{{- range .Values.namespaces }}
{{- if .quota }}
---
apiVersion: v1
kind: ResourceQuota
metadata:
  name: resource-quota
  namespace: {{ .name }}
spec:
  hard:
    requests.cpu: {{ .quota.requests.cpu | default 1 | quote }}
    requests.memory: {{ .quota.requests.memory | default "1Gi" }}
    limits.cpu: {{ .quota.limits.cpu | default 2 | quote }}
    limits.memory: {{ .quota.limits.memory | default "4Gi" }}
    persistentvolumeclaims: {{ .quota.persistentVolumeClaims |
default 5 | quote }}
    requests.storage: {{ .quota.storage | default "10Gi" }}
{{- end }}
{{- end }}
---
```

Configure Values to Create 3 Namespaces, ServiceAccounts, & RoleBindings

```
namespaces:
  - name: backend1
    labels:
      istio-injection: enabled
    annotations:
      network-type: mesh
  - name: backend2
    labels:
      istio-injection: enabled
    annotations:
      network-type: mesh
  - name: backend3
    labels:
      istio-injection: enabled
    annotations:
      network-type: mesh
```

```
serviceAccounts:
  - name: app-admin
    namespace: backend1
    roleBindings:
      - namespace: backend1
        kind: ClusterRole
        name: app-admin
  - name: app-admin
    namespace: backend2
    roleBindings:
      - namespace: backend2
        kind: ClusterRole
        name: app-admin
  - name: app-admin
    namespace: backend3
    roleBindings:
      - namespace: backend3
        kind: ClusterRole
        name: app-admin
```

A bit of repetition, but I didn't want to drastically change a forked chart.

Render the Chart with my Values

```
$ helm template test kube-common-setup -f ns-values/values.yaml \
  --output-dir test

wrote test/k8s-common/templates/namespaces.yaml
wrote test/k8s-common/templates/namespaces.yaml
wrote test/k8s-common/templates/namespaces.yaml
wrote test/k8s-common/templates/service-accounts.yaml
wrote test/k8s-common/templates/service-accounts.yaml
wrote test/k8s-common/templates/service-accounts.yaml
wrote test/k8s-common/templates/cluster-roles.yaml
wrote test/k8s-common/templates/role-bindings.yaml
wrote test/k8s-common/templates/role-bindings.yaml
wrote test/k8s-common/templates/role-bindings.yaml
$ cd test/k8s-common/templates
```

Look to See if it did the Right Thing

```
---
# Source: k8s-common/templates/namespaces.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: backend1
  labels:
    istio-injection: "enabled"
  annotations:
    network-type: "mesh"
---
# Source: k8s-common/templates/namespaces.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: backend2
  labels:
    istio-injection: "enabled"
  annotations:
    network-type: "mesh"
---
# Source: k8s-common/templates/namespaces.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: backend3
  labels:
    istio-injection: "enabled"
  annotations:
    network-type: "mesh"
```

```
---
# Source: k8s-common/templates/service-accounts.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: app-admin
  namespace: backend1
---
# Source: k8s-common/templates/service-accounts.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: app-admin
  namespace: backend2
---
# Source: k8s-common/templates/service-accounts.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: app-admin
     namespace: backend3
```

```
---
# Source: k8s-common/templates/role-bindings.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: app-admin-app-admin
  namespace: backend1
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: app-admin
subjects:
- kind: ServiceAccount
  name: app-admin
  namespace: backend1
---
# Source: k8s-common/templates/role-bindings.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: app-admin-app-admin
  namespace: backend2
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: app-admin
subjects:
- kind: ServiceAccount
  name: app-admin
  namespace: backend2
---
# Source: k8s-common/templates/role-bindings.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: app-admin-app-admin
  namespace: backend3
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: app-admin
subjects:
- kind: ServiceAccount
  name: app-admin
  namespace: backend3
```

Commit and Tag

```
$ git add values.yaml
$ git commit -m "create 3 backend namespaces"
[main e990171] create 3 backend namespaces
 1 file changed, 36 insertions(+)
 create mode 100644 values.yaml
$ git tag v1
```

Push

```
$ git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 496 bytes | 496.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/bgrant0607/ns-values
   0100b4c..e990171  main -> main

$ git push --tags
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/bgrant0607/ns-values
 * [new tag]         v1 -> v1
```

Deploy

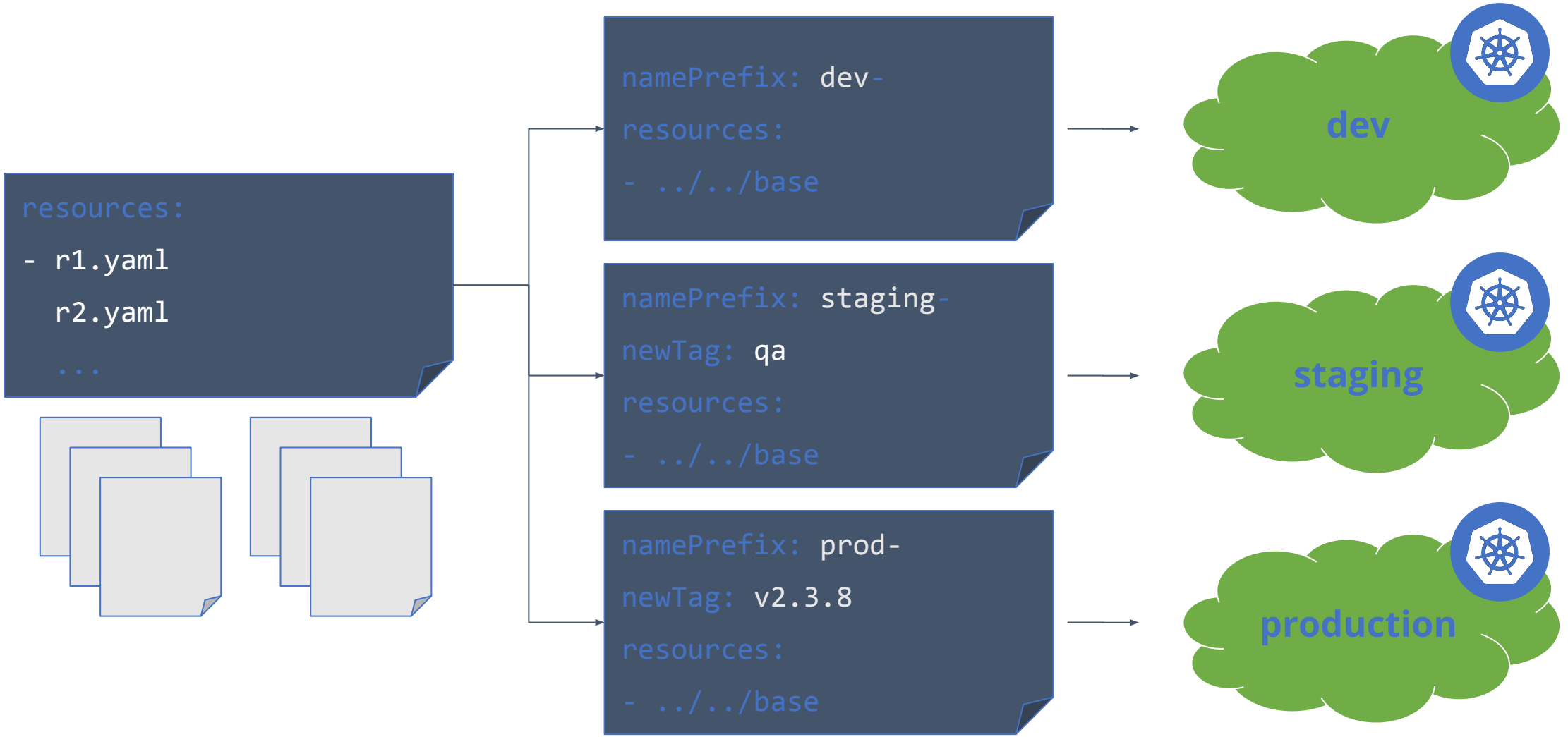
```
$ helm install mysetup kube-common-setup -f
ns-values/values.yaml


# To make a change to the chart and/or values:

# helm upgrade mysetup kube-common-setup -f
ns-values/values.yaml
```

Kustomize

Variants are a Common Case



Kustomize Enables Splitting the Common and Custom

kustomization.yaml

```
namePrefix: prod-
commonLabels:
  variant: prod
commonAnnotations:
  note: Hello, I am
production!
bases:
- ../../base
patches:
- replica_count.yaml
- cpu_count.yaml
```

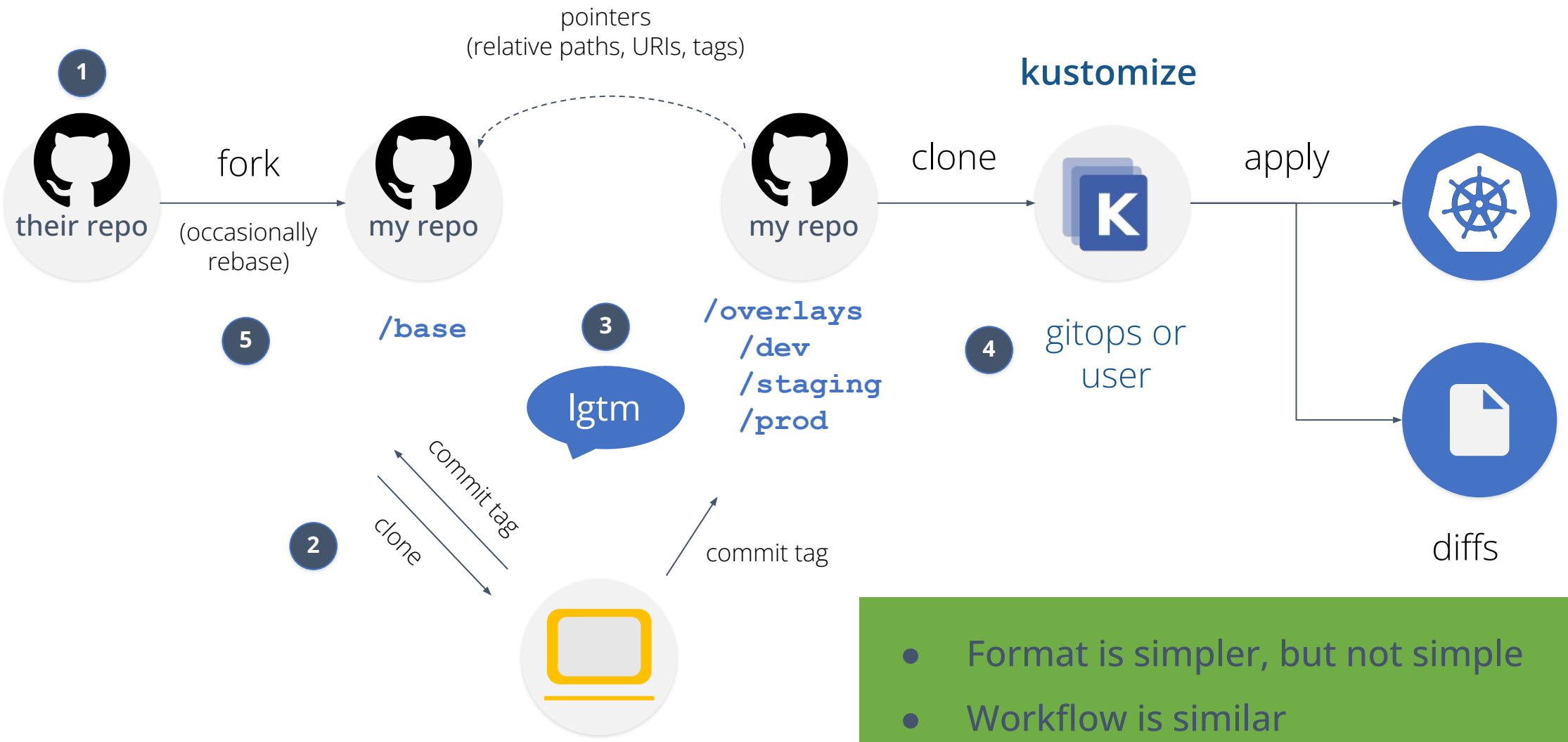
replica_count.yaml

```
apiVersion: v1
kind: Deployment
metadata:
  name: wordpress
spec:
  replicas: 80
```

cpu_count.yaml

```
apiVersion: v1
kind: Deployment
metadata:
  name: wordpress
spec:
  template:
    spec:
      containers:
      - name: my-container
        resources:
          limits:
            cpu:  7000m
```

Off-the-Shelf Config Workflow



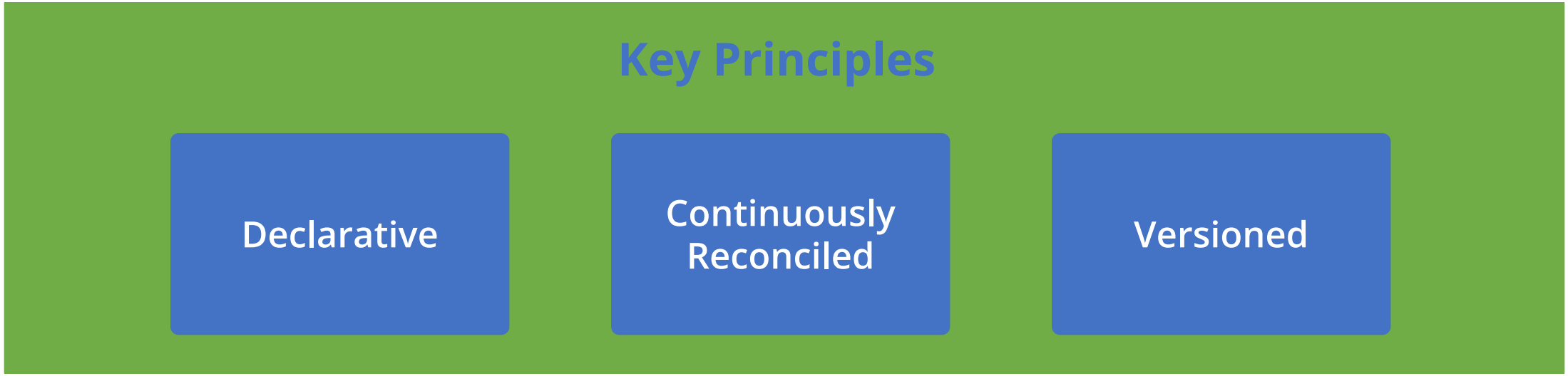
- Format is simpler, but not simple
- Workflow is similar

GitOps

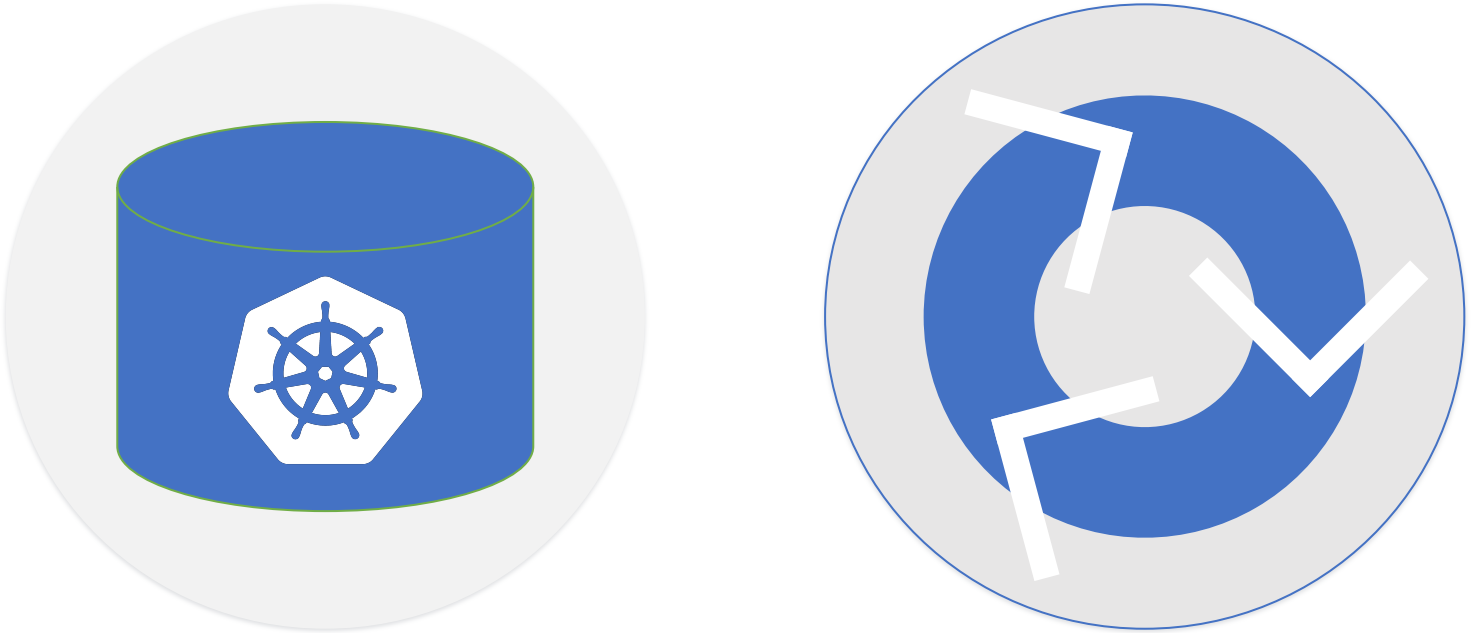
What is GitOps?

Software agents continuously observe actual system state and attempt to apply the desired state.

- Automates that last deploy/update step
- More importantly, binds live state to the configuration in storage



The Kubernetes control model is continuous reconciliation

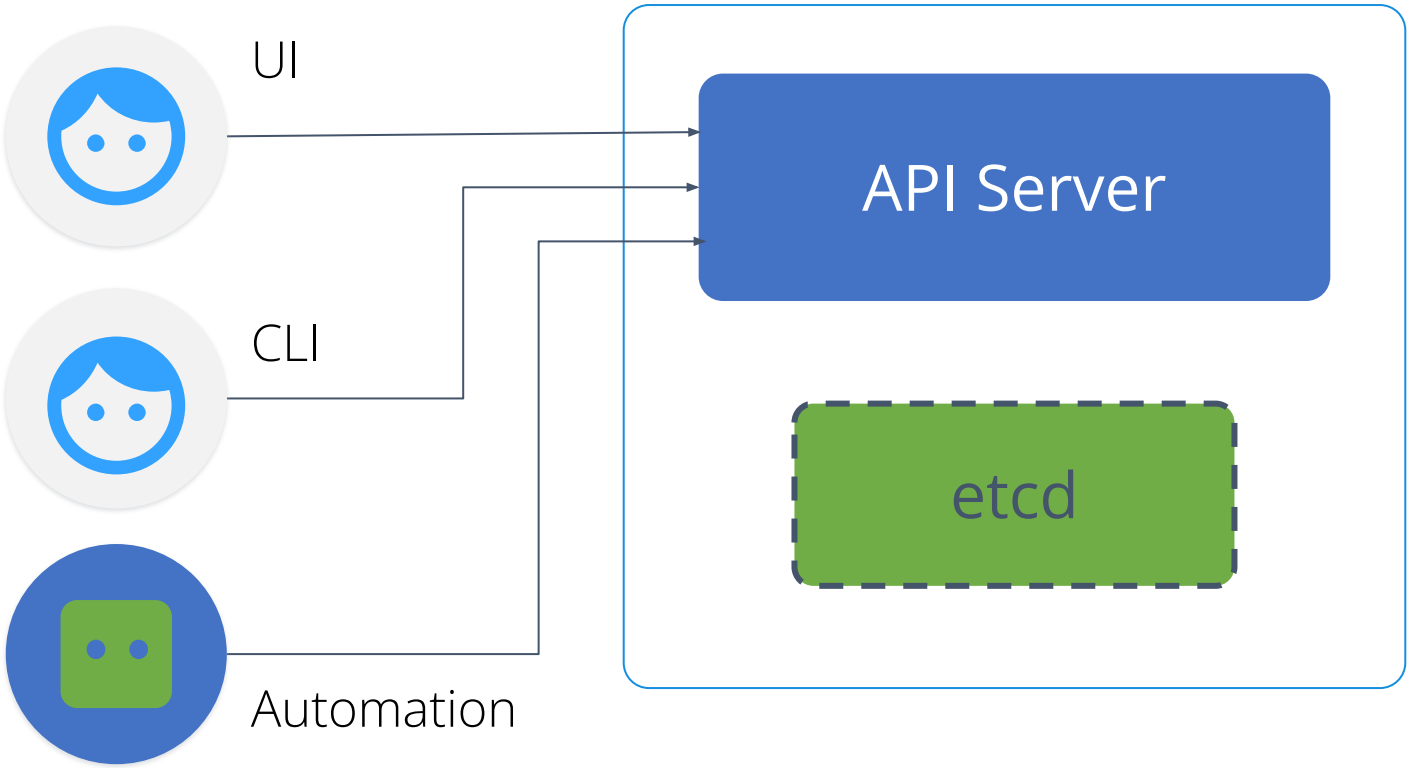


The source of truth is etcd. It’s declarative, but not versioned

Kpt:
Configuration as Data

THE LINUX FOUNDATION

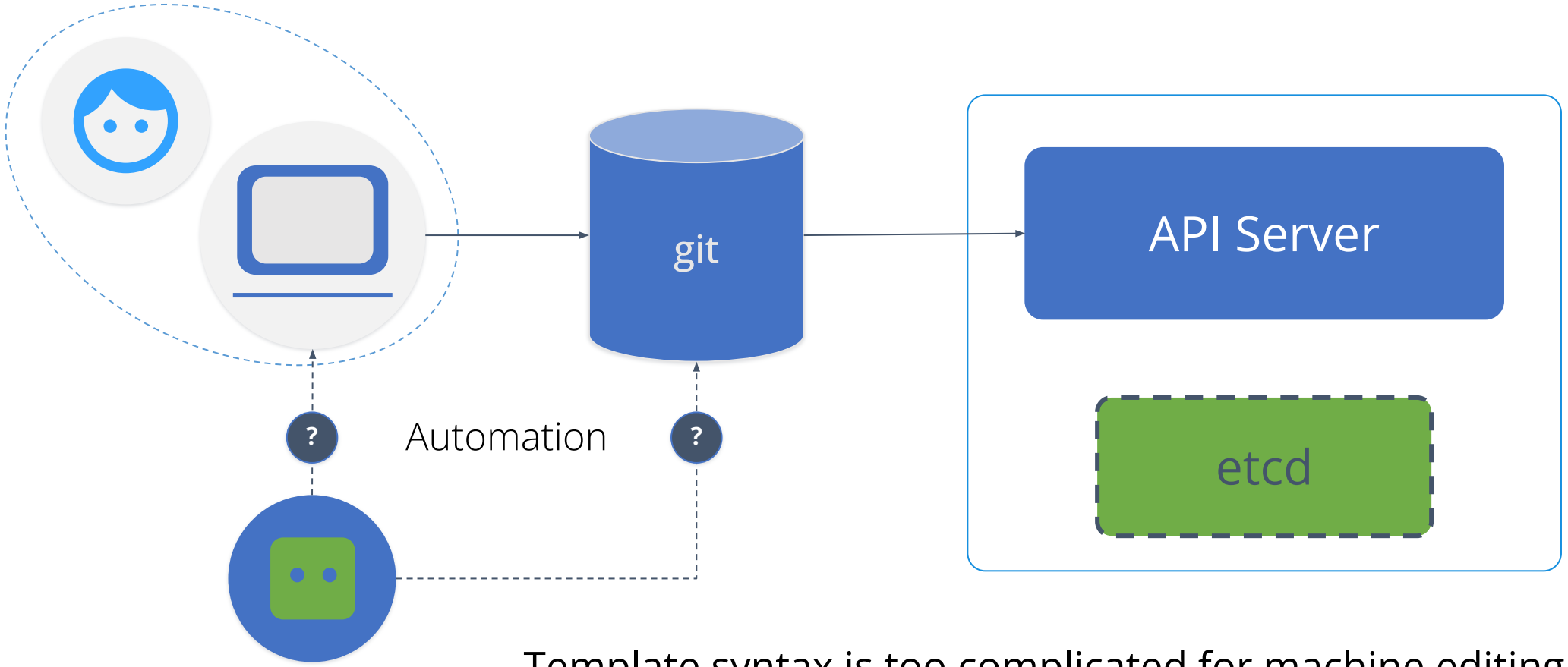
Kubernetes is Automated Through its API



Existing Methods of Mutated Changes via the Kubernetes API

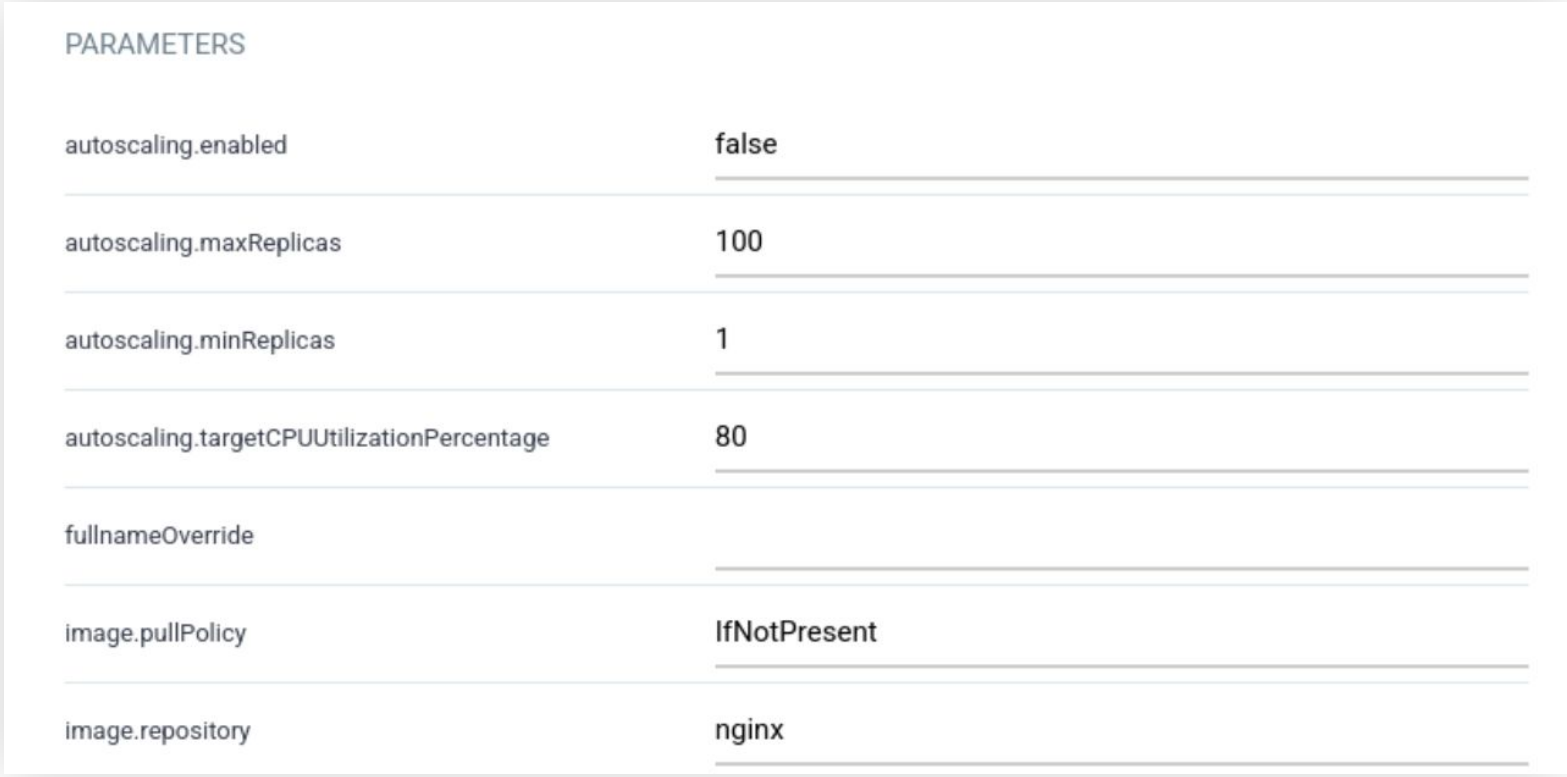
- Synchronous mutation: mutating admission controllers
- Asynchronous mutation: controllers

Templates Need to be Edited by Hand & Block Automation



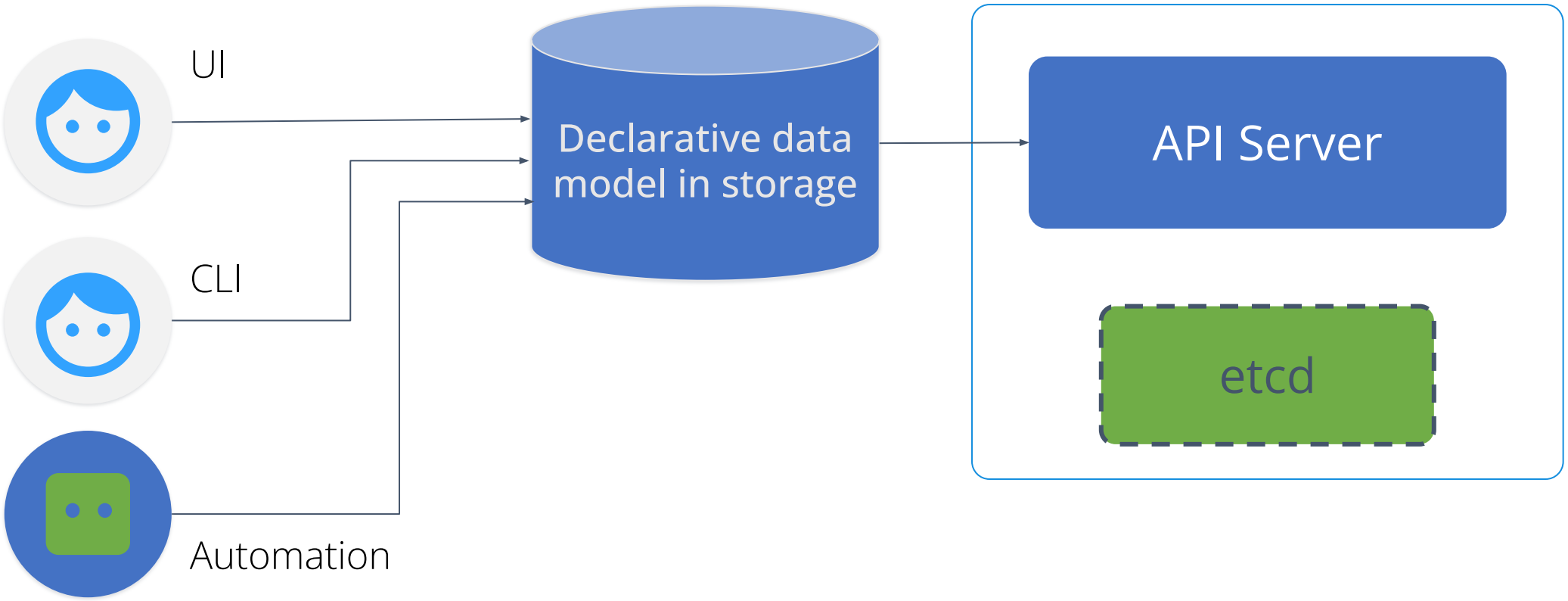
Template syntax is too complicated for machine editing, but direct API writes are considered undesirable “drift”

Templates Parameters are a Human Interface



Not an API. Different for every package.

What if we could Operate on Configuration in Storage?

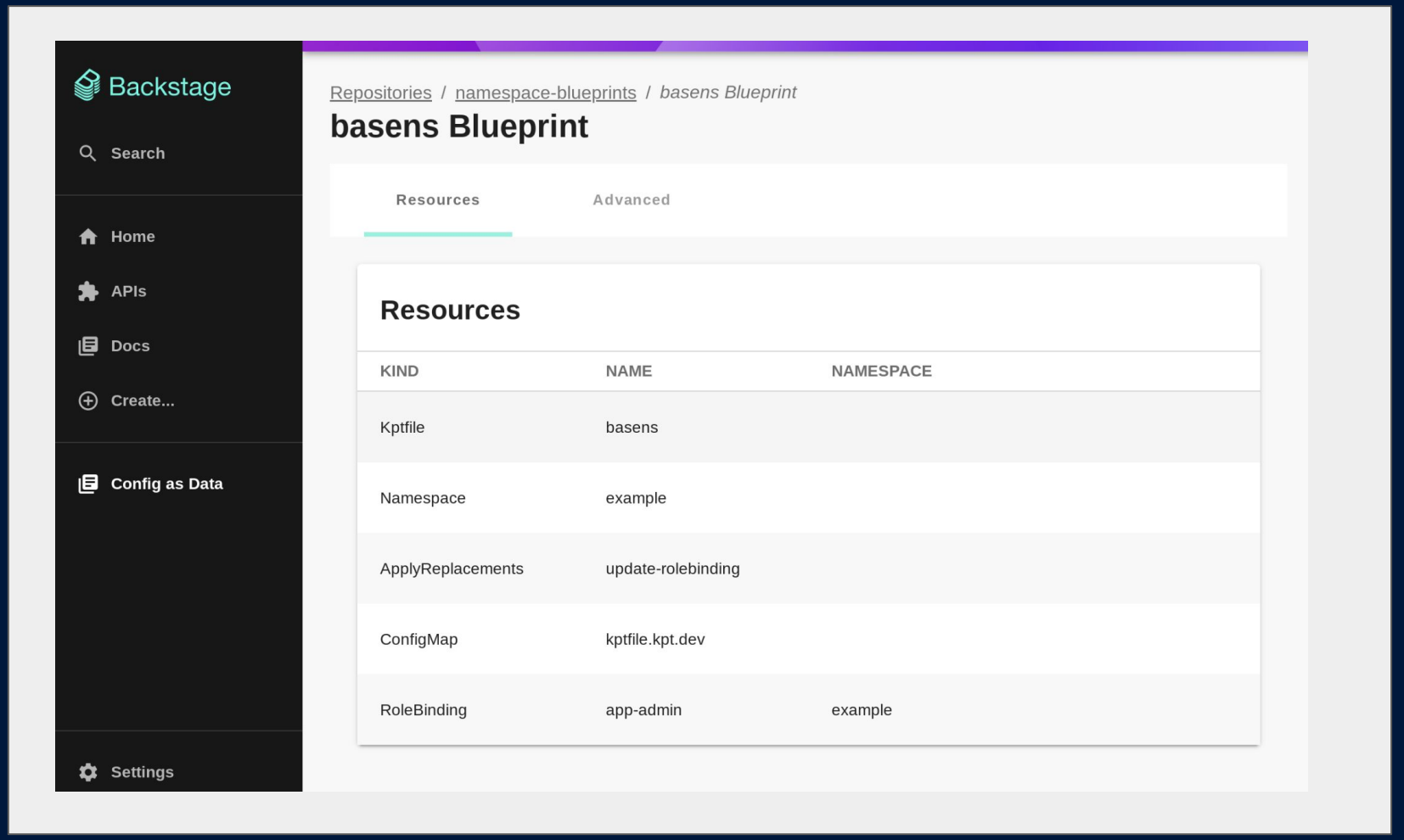


What is Configuration as Data?

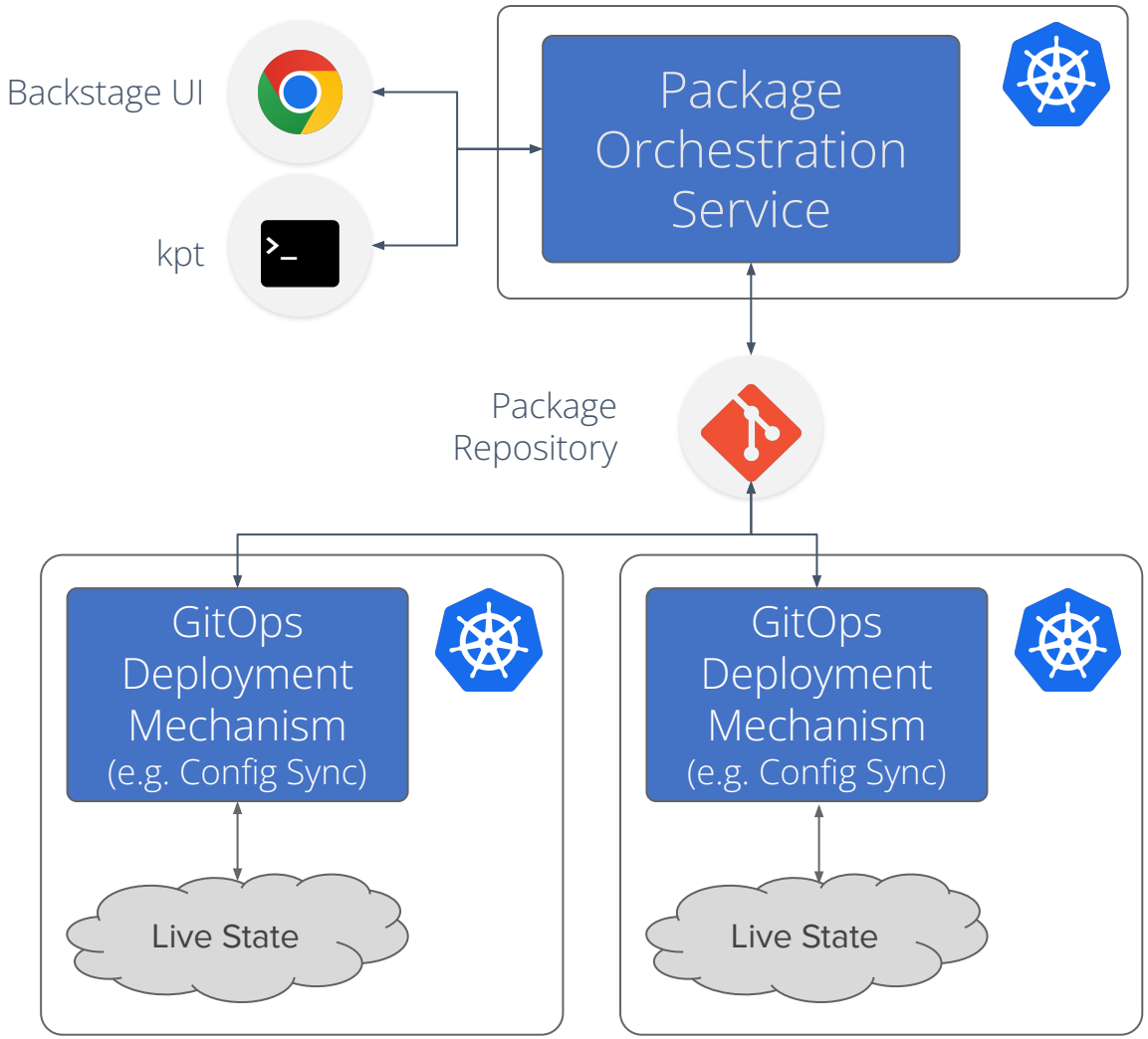
Simple core principles:

1. Makes configuration data in versioned storage (git) the source of truth
2. Uses a uniform, serializable data model (KRM) to represent configuration
3. Separates code that acts on the configuration from the data
4. Clients manipulating configuration data don’t need to directly interact with storage, they operate on data via APIs

Demo!


Backstage
Search
Home
APIs
Docs
Create...
Config as Data
Settings
Repositories / namespace-blueprints / basens Blueprint
basens Blueprint
Resources
Advanced
Resources
KIND NAME NAMESPACE
Kptfile basens
Namespace example
ApplyReplacements update-rolebinding
ConfigMap kptfile.kpt.dev
RoleBinding app-admin example

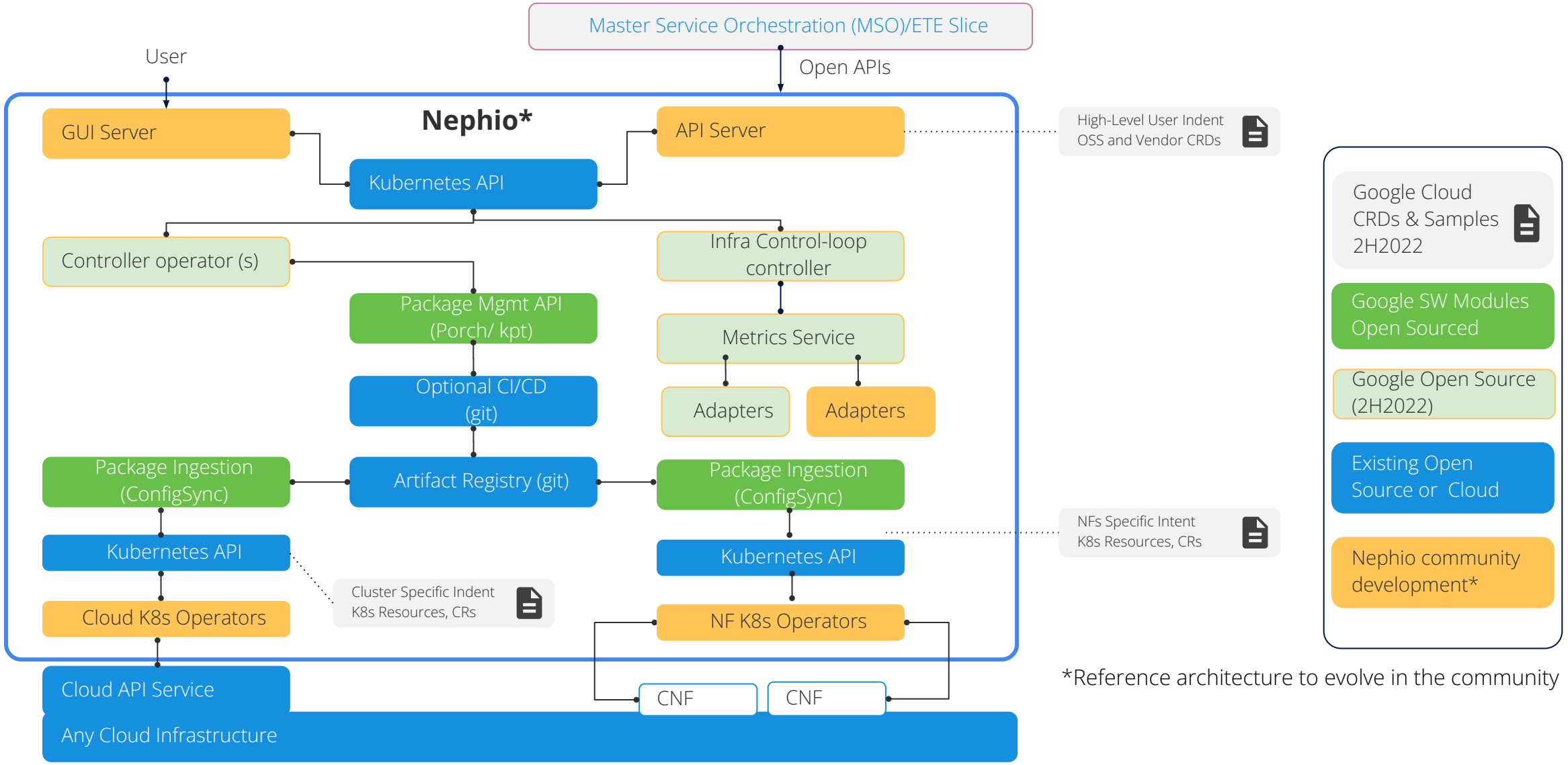
kpt's Current Architecture



Package orchestrator functionality:

- Create, clone, upgrade, rollback, delete, read, write packages
- Execute functions and function pipelines on packages
- Change control
- Package and function discovery

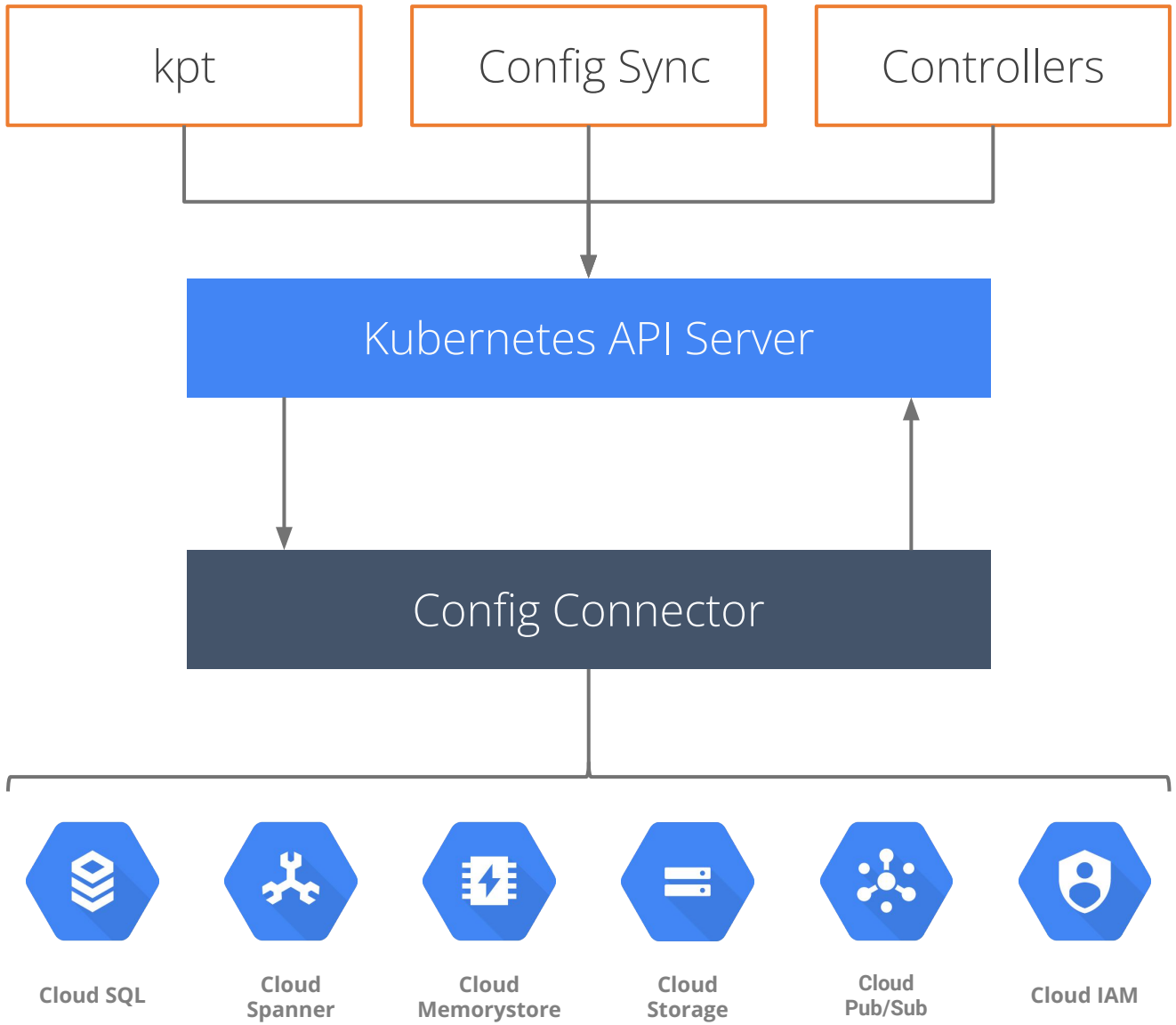
Nephio Reference Architecture



*Reference architecture to evolve in the community

CaD can be extended to infrastructure through Operators

```
apiVersion:
 pubsub.cnrm.cloud.google.com/v1beta1
kind: PubSubTopic
metadata:
 labels:
   label-one: "value-one"
 name: pubsubtopic-sample
```



Wrap up

- kpt enables programmatic manipulation of configuration data via APIs
- This provides a foundation for Nephio's configuration automation
- These components are open source
- Contributors are welcome!

Check out kpt.dev

Nephio Reference Implementation

Deep Dive into SIG Platform

John Belamaric, Senior Staff Software Engineer, Google Cloud

[Recording from Nephio Summit, June 22, 2022]

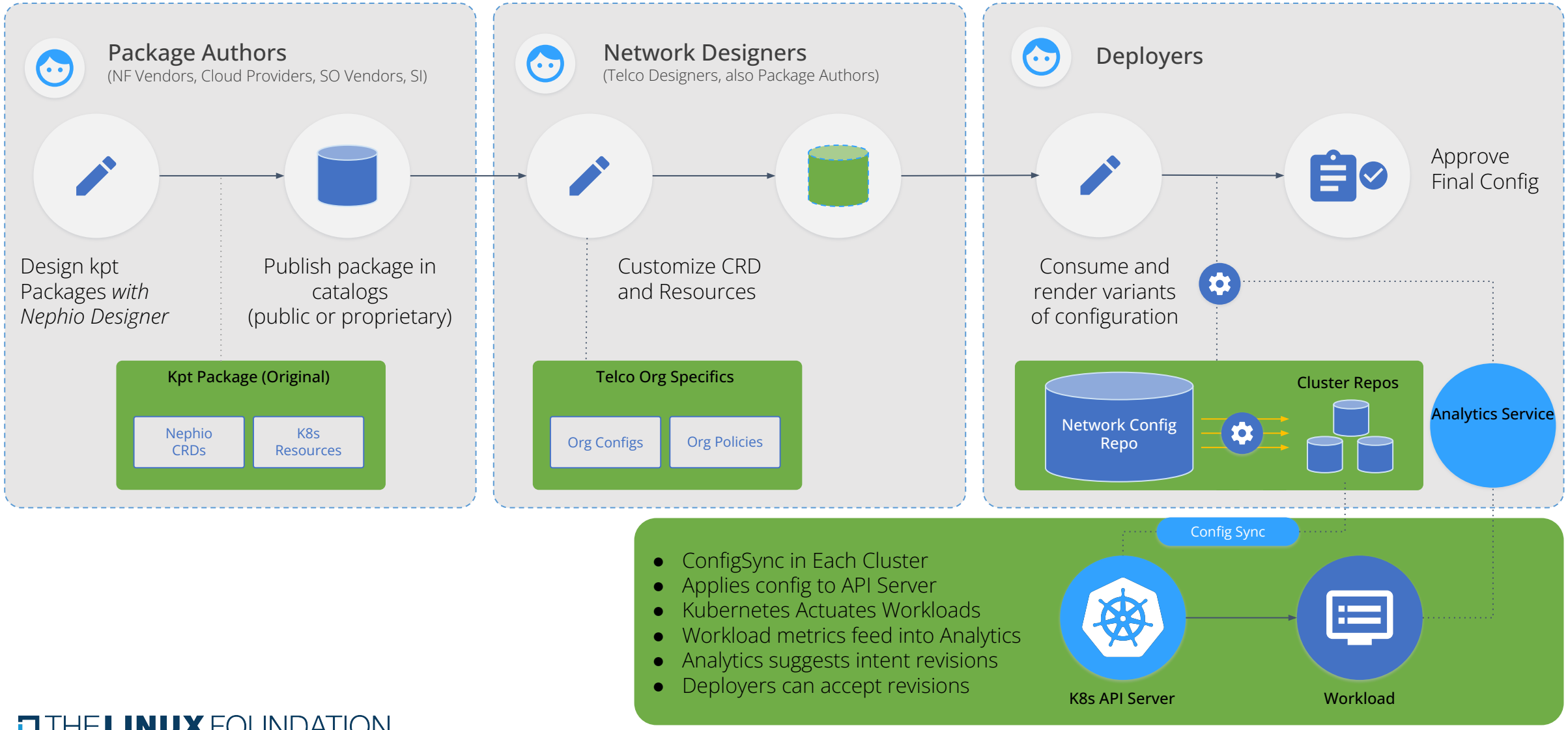
Who am I?

- Senior tech leader in Anthos and GKE at Google Cloud
- Co-chair of Kubernetes SIG Architecture
- Emeritus approver of Kubernetes SIG Network
- Led development of CoreDNS/Kubernetes integration
- Led development of pluggable IPAM in OpenStack Neutron
- Former tech lead / architect of a successful commercial enterprise network automation product
  - Automates configuration of tens of thousands of switches, routers, and other devices

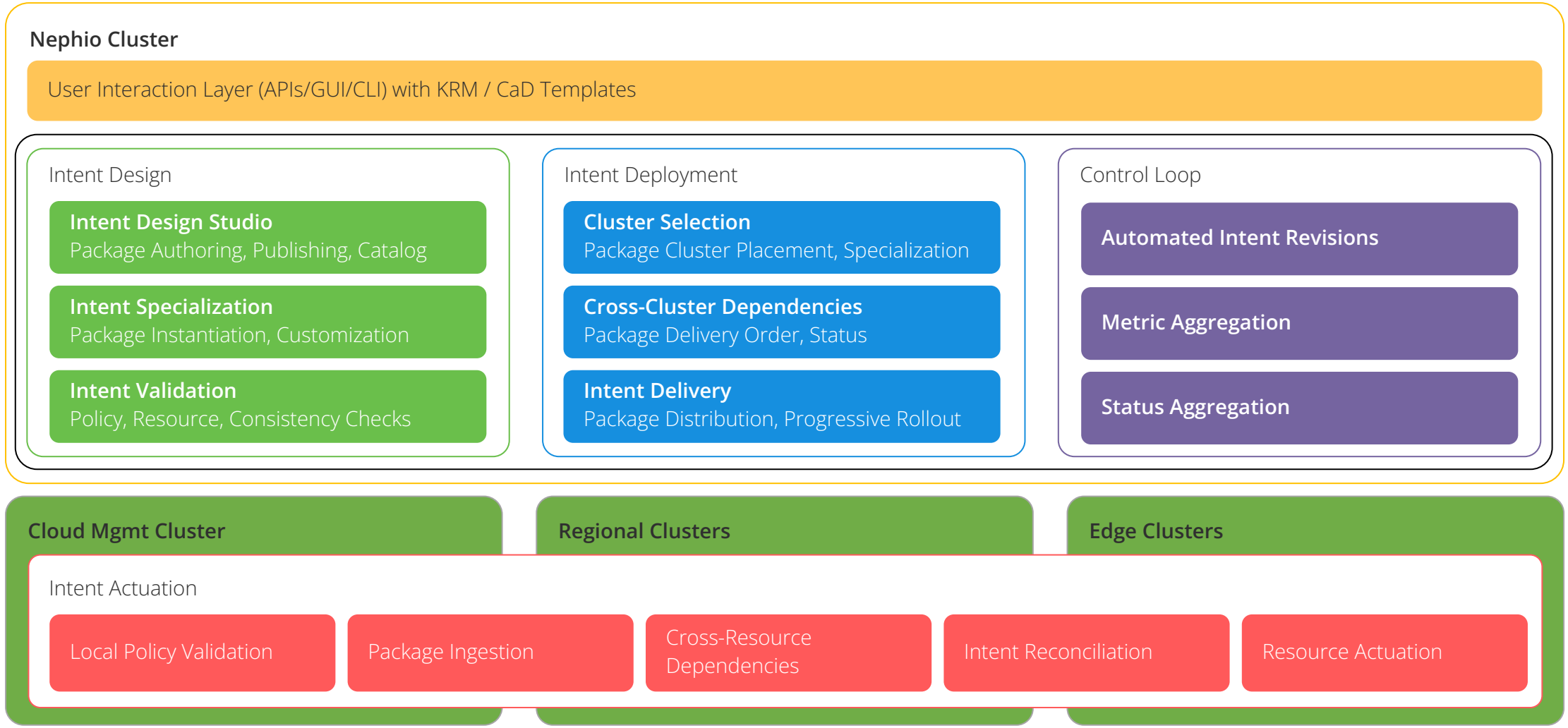
What we're going to talk about

- Revisit the end-to-end user journey
- The Nephio architecture
  - Components, and why do we say "reference" implementation?
  - How they support the end-to-end user journey
  - Which come from Google and which the community needs to build
- What will we actually release?
- Let's get started!

Revisiting the Nephio End-to-End Journey

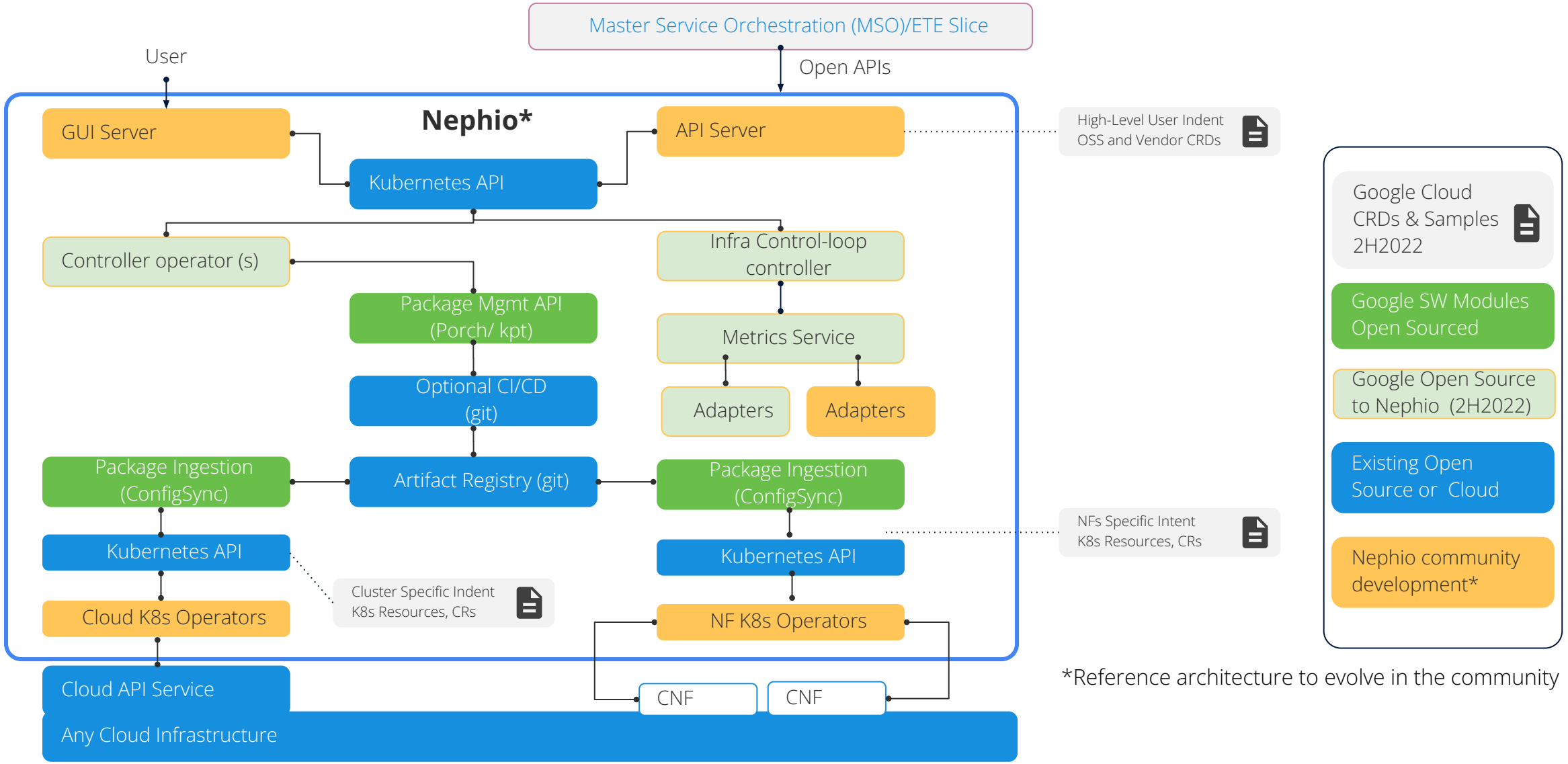


Nephio Functional Building blocks



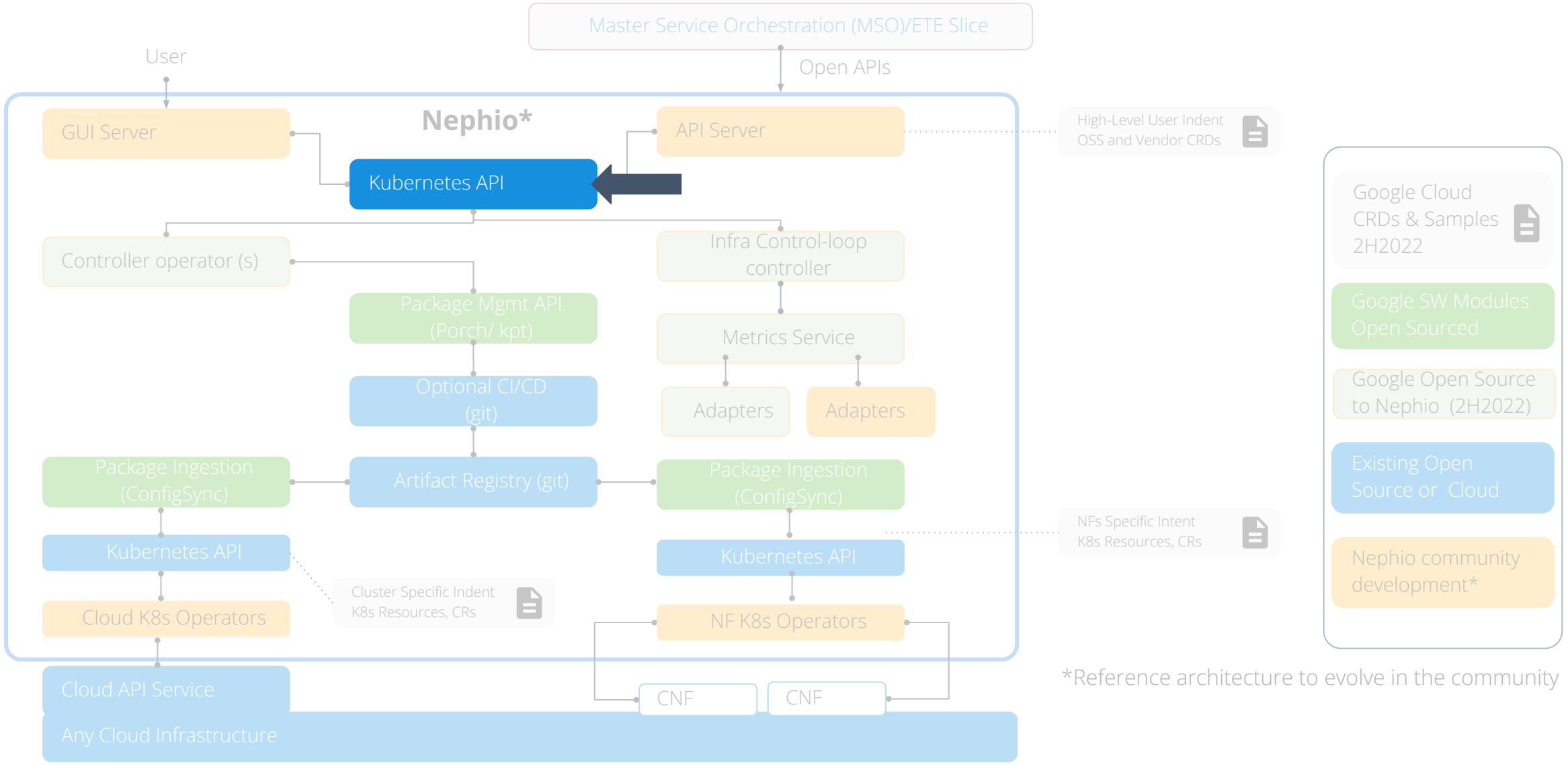
KRM - Kubernetes Resource model
CaD - Configuration as Data

Nephio High-Level Architecture



*Reference architecture to evolve in the community

Nephio High-Level Architecture



*Reference architecture to evolve in the community

Kubernetes API Server

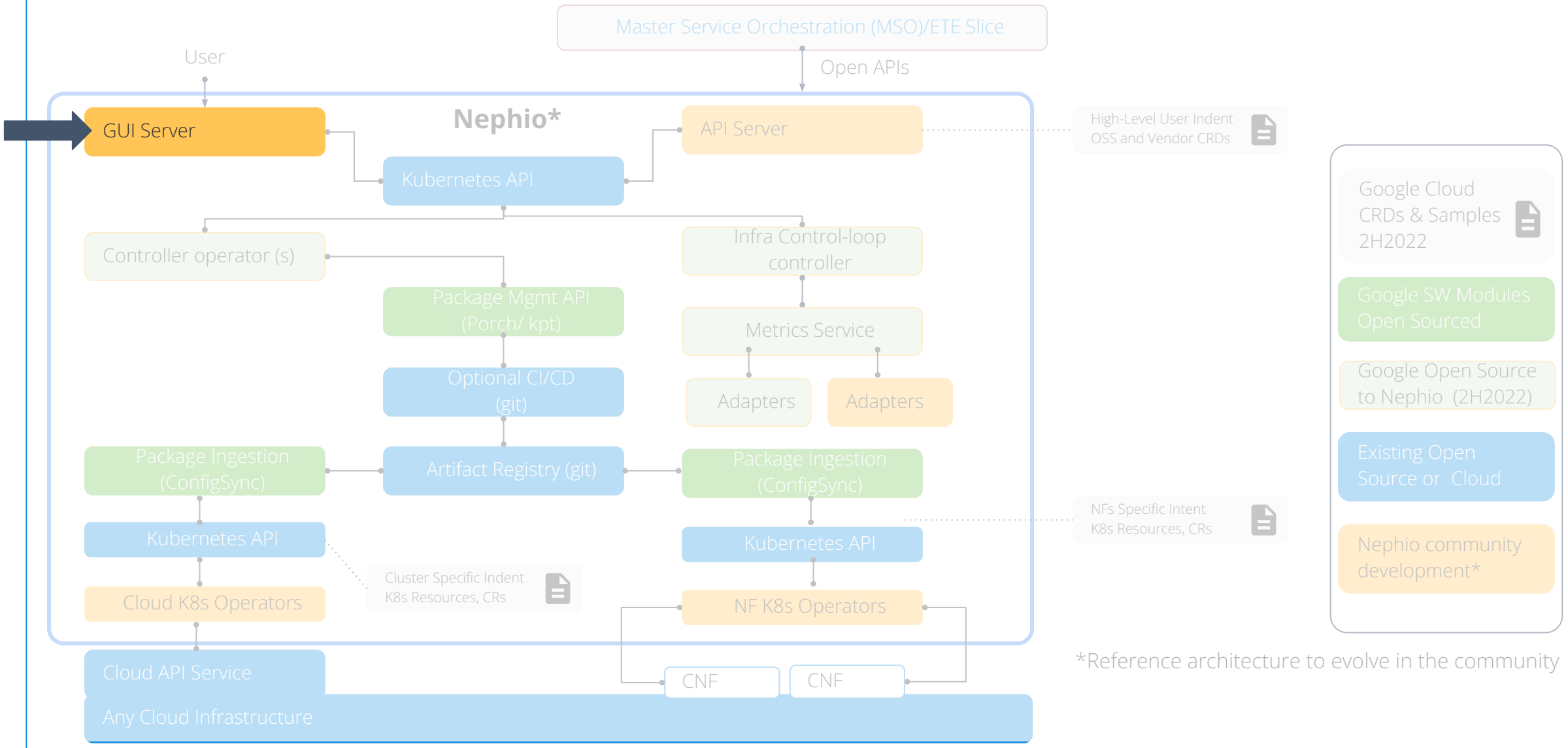
Functional Diagram: **Nephio Cluster**

Architecture Diagram: Kubernetes API

OSS Status: Existing Open Source or Cloud

- Kubernetes API server is the basis for all of Nephio
- Runs in all clusters
  - Central Nephio cluster, and cloud mgmt cluster, edge clusters
- Nephio runs as workloads on Kubernetes
- Nephio APIs are extensions of the Kubernetes API - CRDs
- Nephio relies on all basic Kubernetes API server functionality
  - Authentication, Authorization / Role-based Access Control
  - Schema definition and serving of APIs (CRDs, Aggregated API server)
  - Workload management, monitoring
  - And more

Nephio High-Level Architecture



*Reference architecture to evolve in the community

Graphical User Interface & Nephio CLI

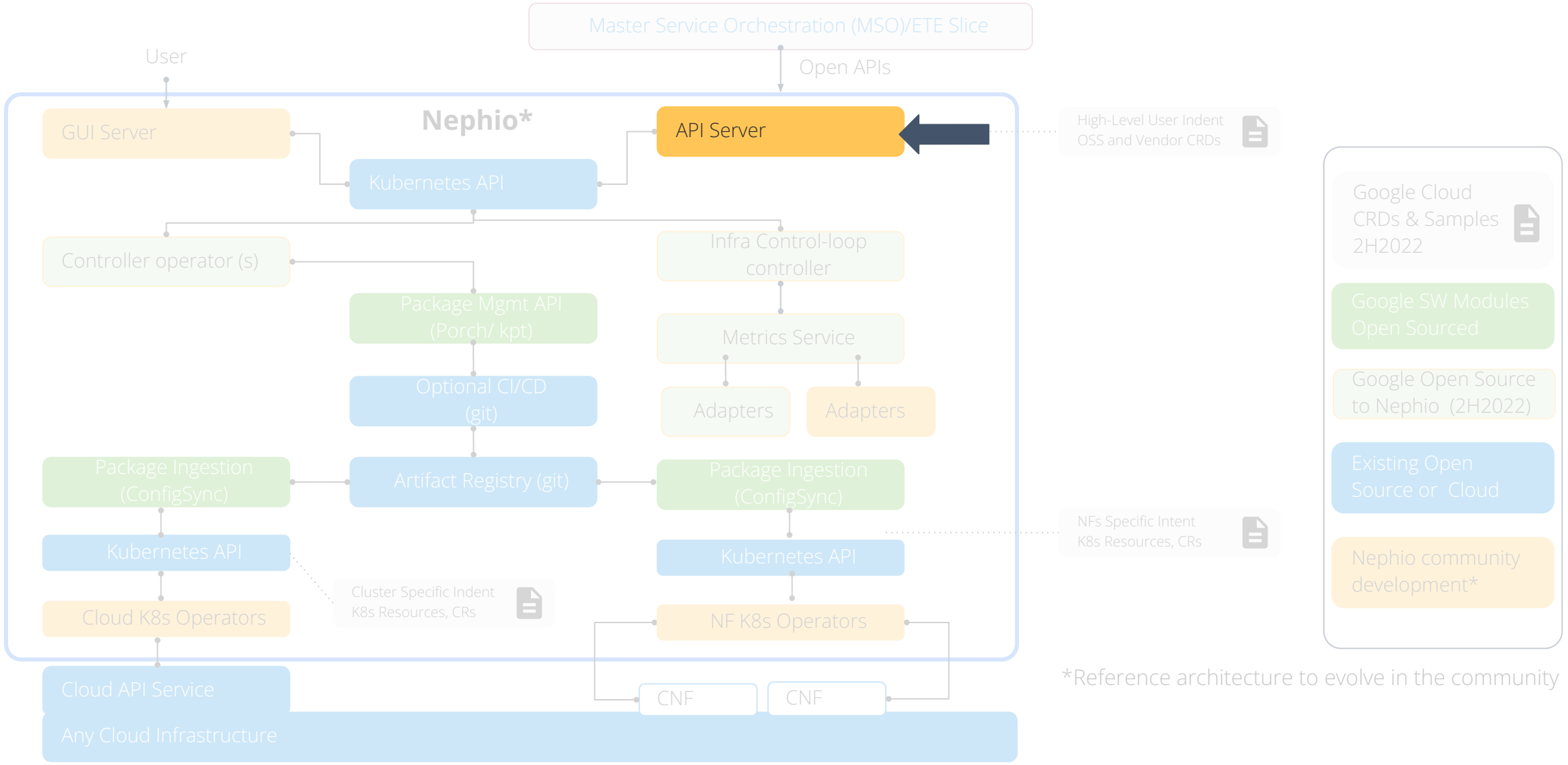
Functional Diagram: User Interaction Layer

Architecture Diagram: GUI Server

OSS Status: Nephio community development

- Community-driven development of a GUI / CLI client
- Overall design needed from the community
- May be simply a web server to download client-side app
- Or combination of client-side app and additional APIs
- Designs and decisions needed on:
  - Separate `nephioctl` or rely on `kpt` / `kubectl` ?
  - Client or server side opinions on top of Kubernetes / Porch APIs ?
  - Extend or rely on OSS Porch GUI ?

Nephio High-Level Architecture



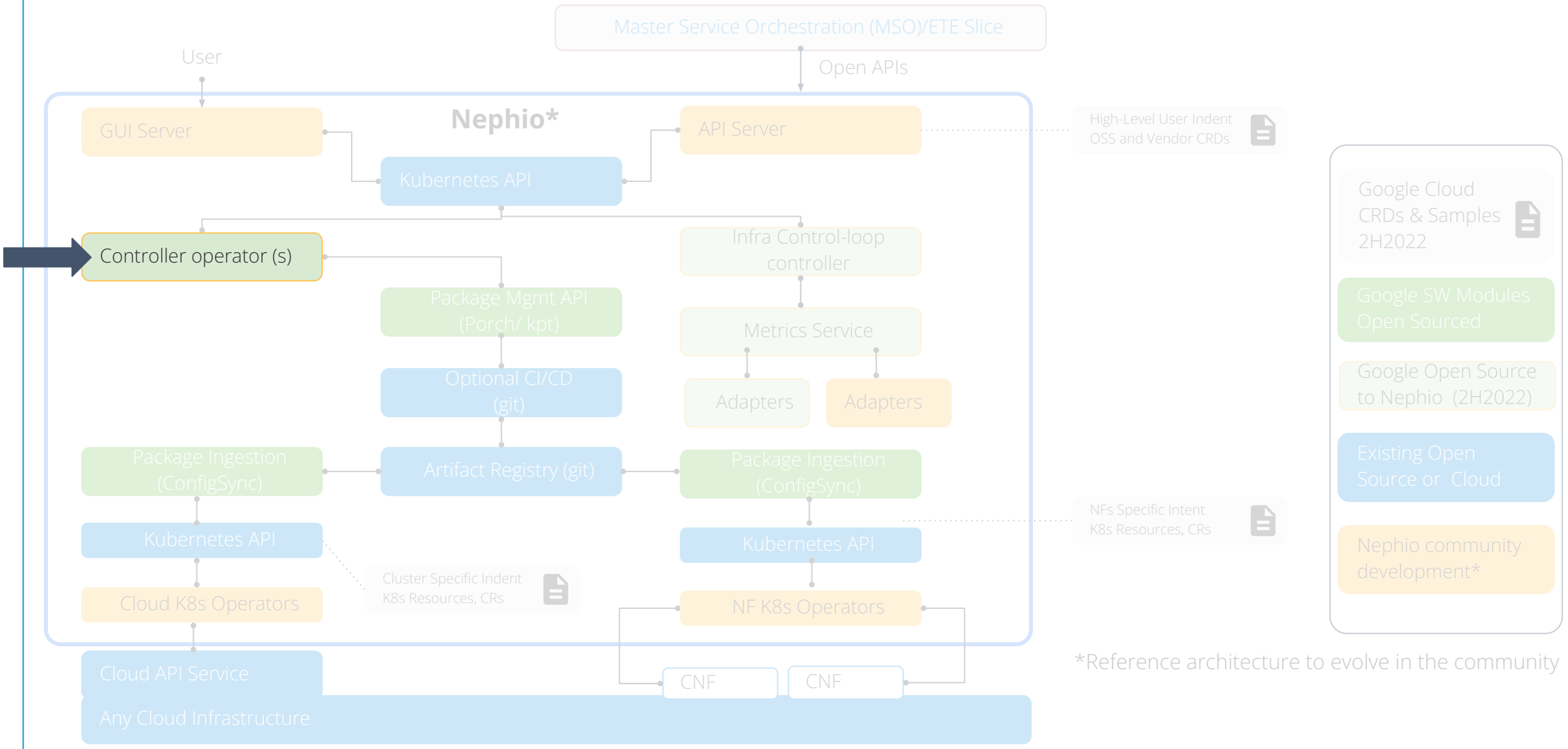
*Reference architecture to evolve in the community

Northbound API Service

Functional Diagram:	User Interaction Layer
Architecture Diagram:	API Server
OSS Status:	Nephio community development

- Provides standards-based APIs for MSO/E2E use
- Overall design needed from the community
- Could be combined with API surface used by GUI
- Could simply not be done, and let MSO talk to K8s API Server

Nephio High-Level Architecture



*Reference architecture to evolve in the community

Nephio Controller Manager

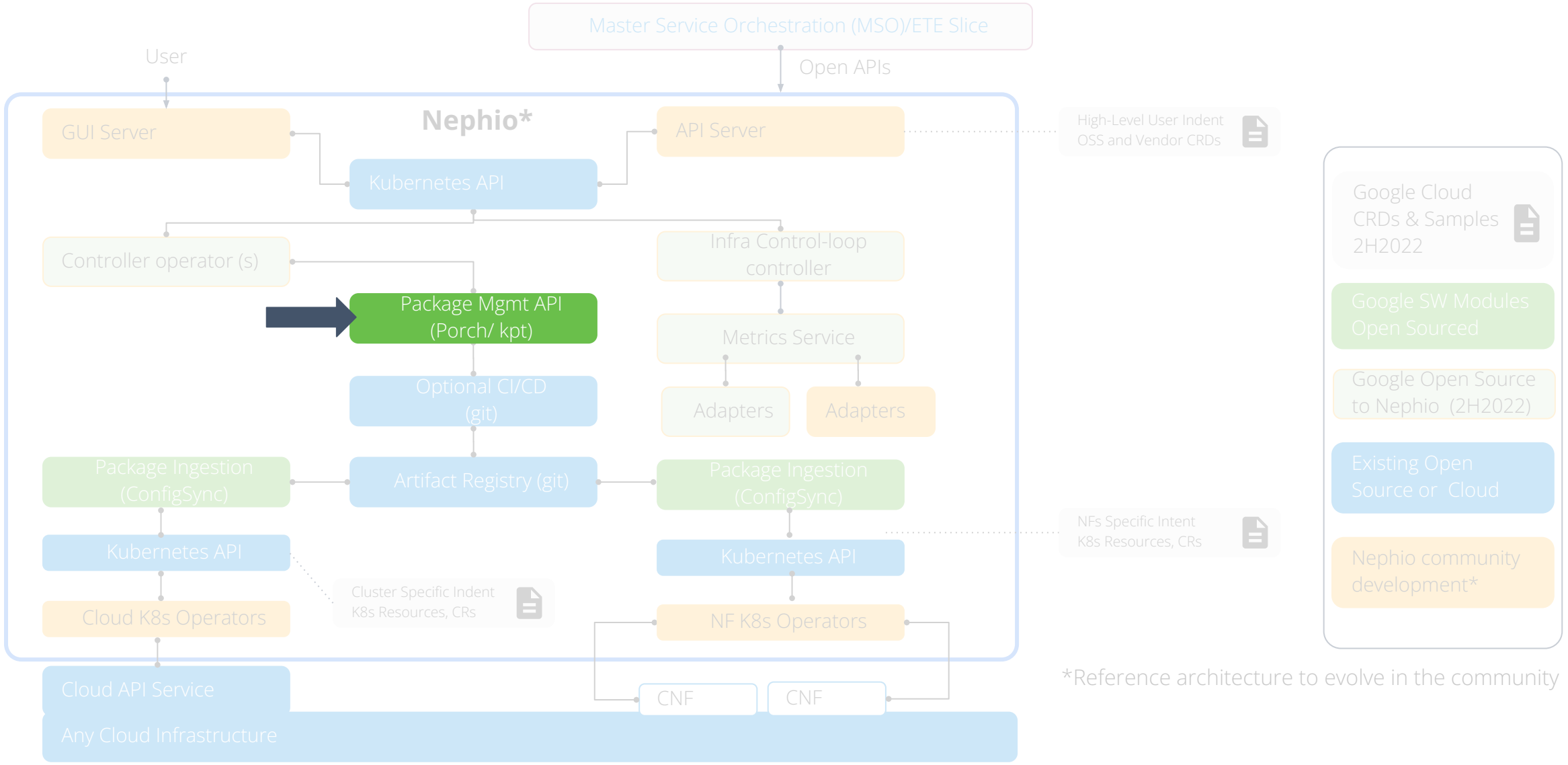
Functional Diagram: Intent Design | Intent Deployment

Architecture Diagram: Controller operator(s)

OSS Status: Google Open Source to Nephio (2H2022)

- Runs in the Nephio cluster
- Kubernetes controllers for Nephio-specific resources
- No cloud provider specific or function vendor specific code
- May be broken into several binaries if it makes sense
- Example functionality:
  - Edge site resource controllers
  - Edge cluster resource controllers
  - Logic mapping clusters to repositories
  - Vendor-neutral portions of function config and interconnection

Nephio High-Level Architecture



*Reference architecture to evolve in the community

Porch

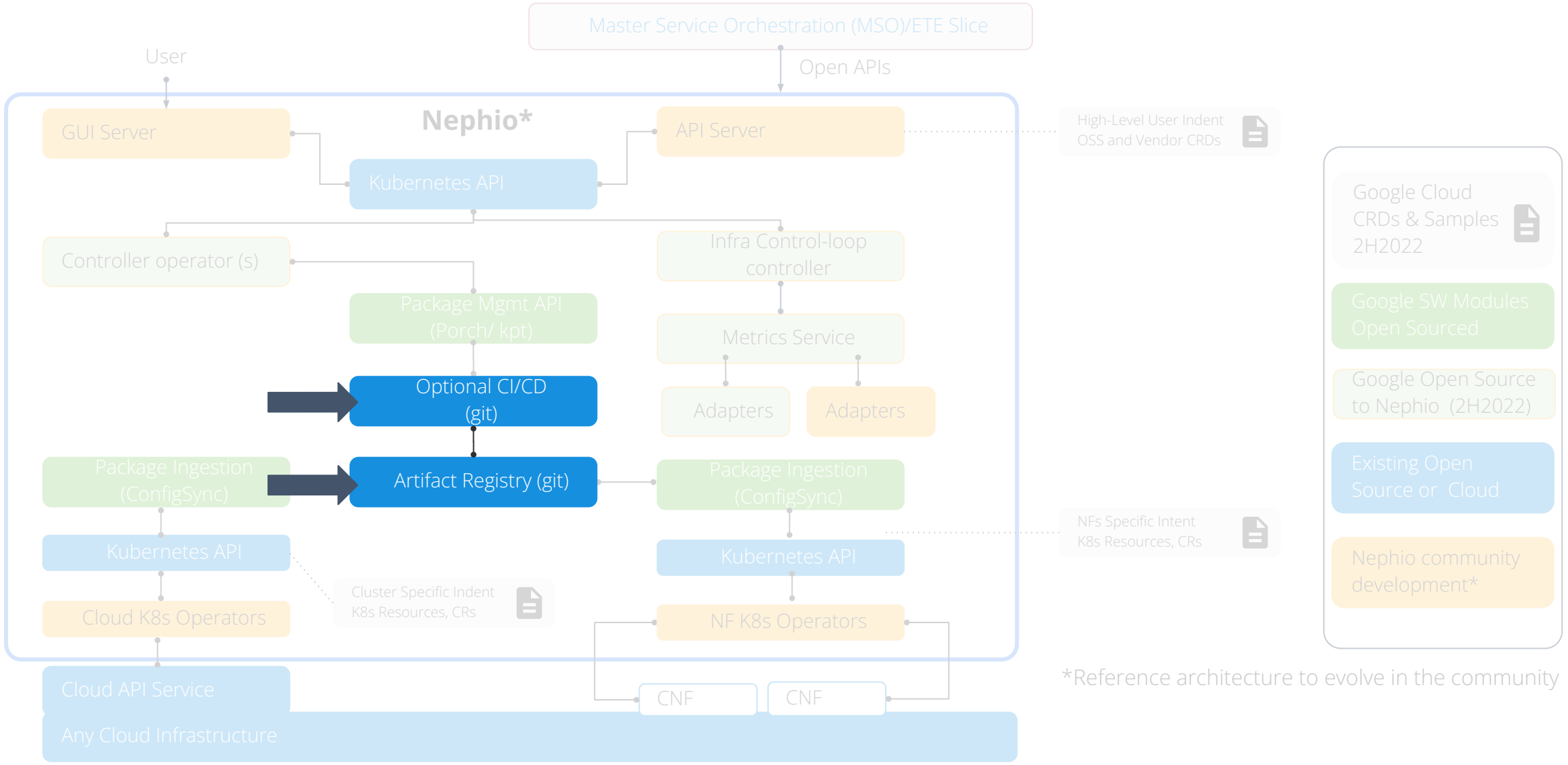
Functional Diagram: Intent Design | Intent Deployment

Architecture Diagram: Package Mgmt API (Porch / kpt)

OSS Status: Google OSS

- Runs in the Nephio cluster
- Aggregated Kubernetes API Server for managing repositories and packages
- Hydrates and validates packages
- Enables programmatic manipulation of packages and contents
- Nephio Controller Manager uses this as a southbound API on top of repositories

Nephio High-Level Architecture



*Reference architecture to evolve in the community

Config Repositories

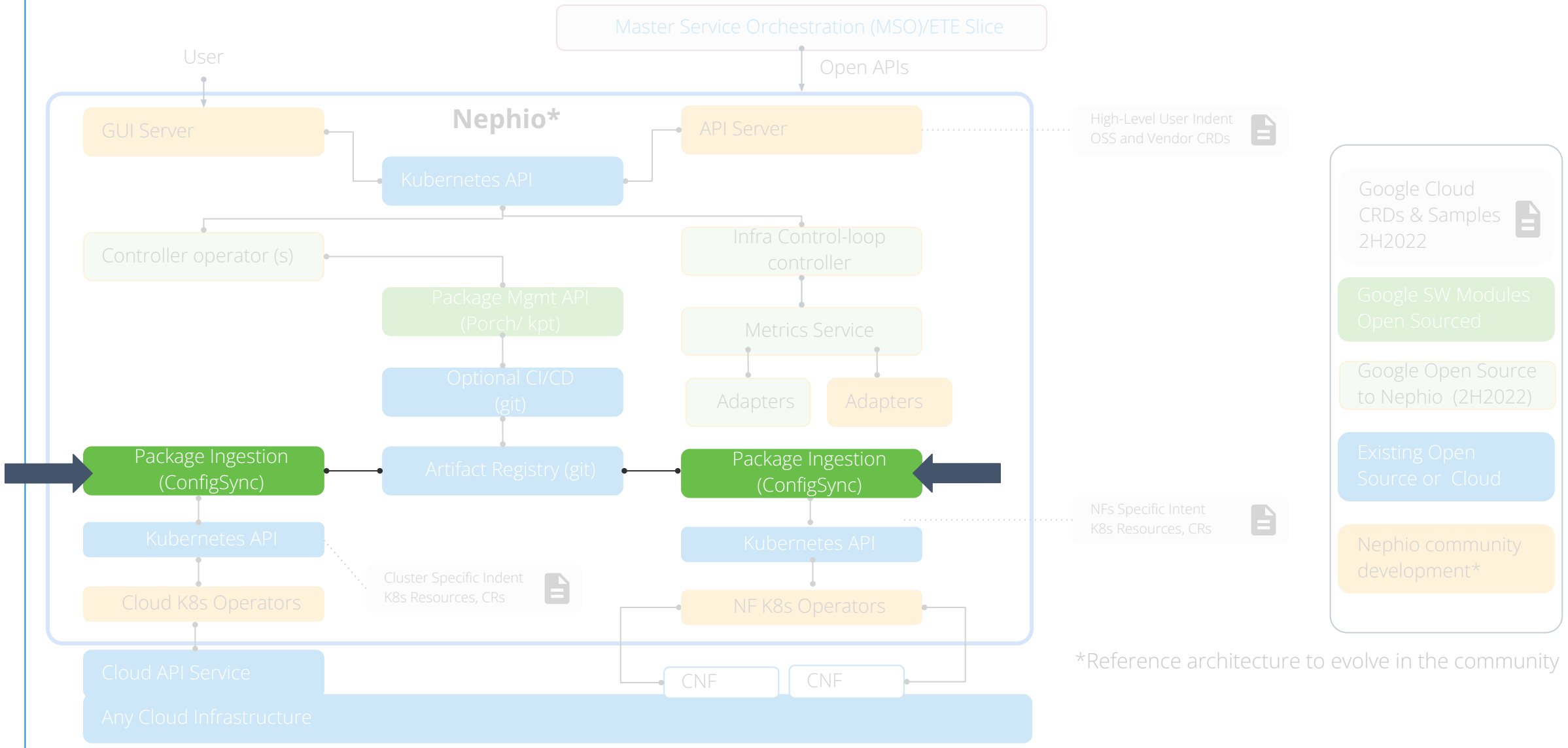
Functional Diagram: Intent Deployment

Architecture Diagram: Optional CI/CD (git) | Artifact Registry (git)

OSS Status: Existing Open Source or Cloud

- Storage for kpt packages / configuration
- Support for various Git-providers and OCI repositories
- Provide a hook-point for existing CI / CD services
- “Catalog” repositories contain abstract packages available for cloning and customization
- Packages may be cloned and modified into other repositories
- Final, deployable (fully specified) packages are stored in per-cluster repositories
- Packages refer to upstream predecessors in other repos to enable Day 2 upgrade / update scenarios

Nephio High-Level Architecture



*Reference architecture to evolve in the community

Config Sync

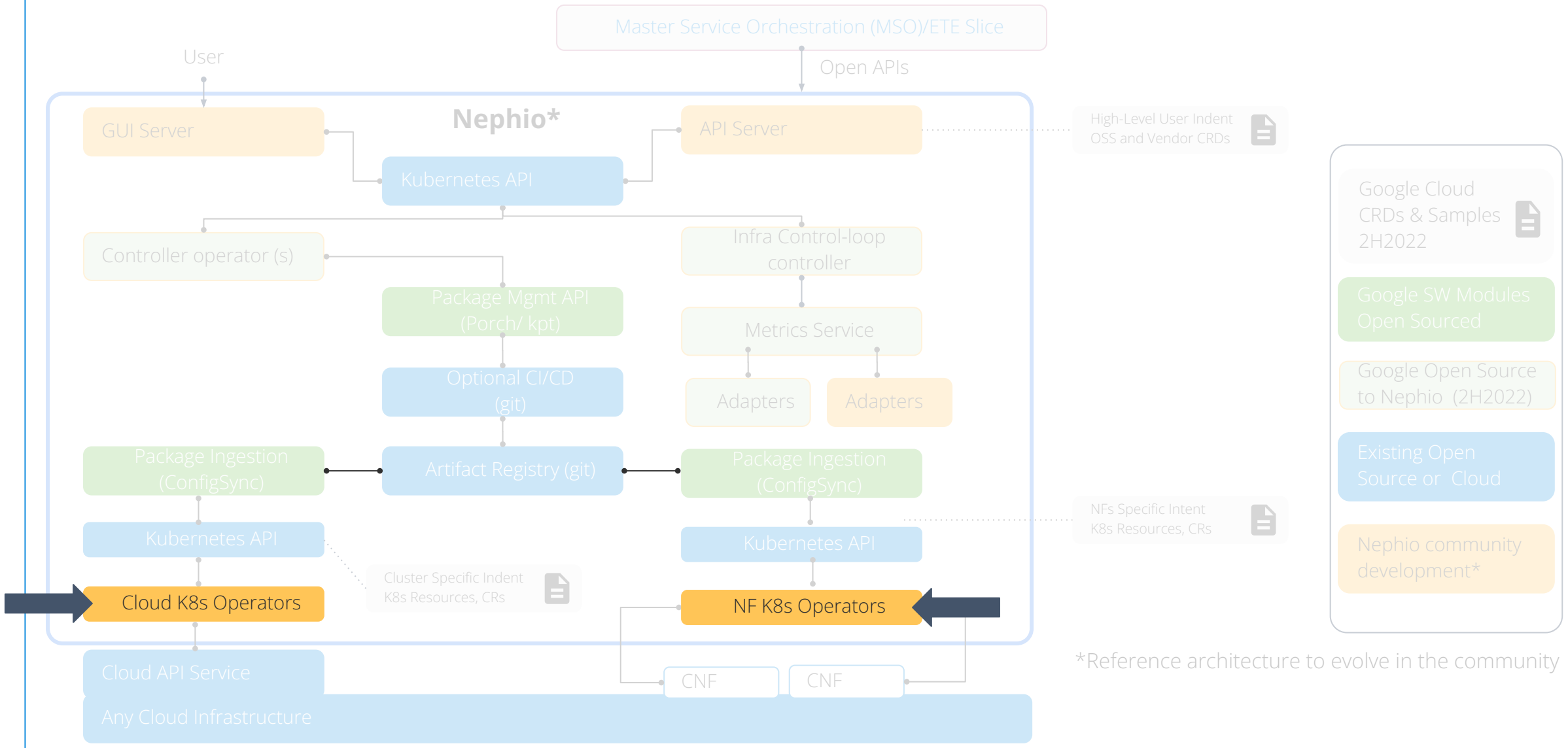
Functional Diagram: Intent Actuation

Architecture Diagram: Package Ingestion (ConfigSync)

OSS Status: Google OSS

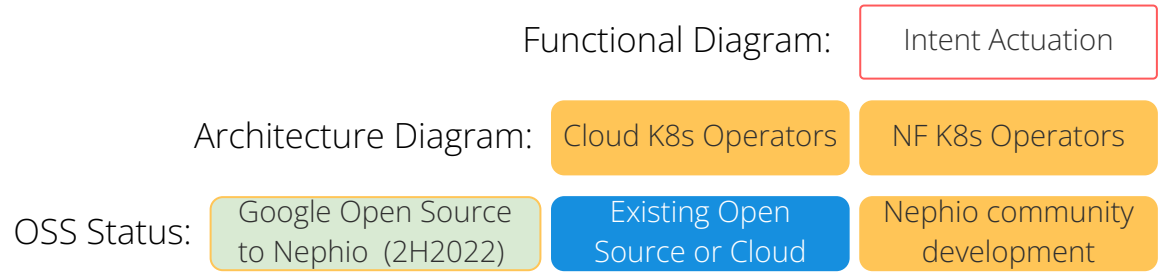
- Each cluster has a corresponding repository for its workload configs
- ConfigSync synchronizes resources from storage (git or OCI) to Kubernetes API Server
- ConfigSync runs in each cluster
  - Edge clusters
  - Cloud mgmt cluster - contains KRM representations of Cloud Infra
  - Nephio cluster - for automation-consumed resources

Nephio High-Level Architecture



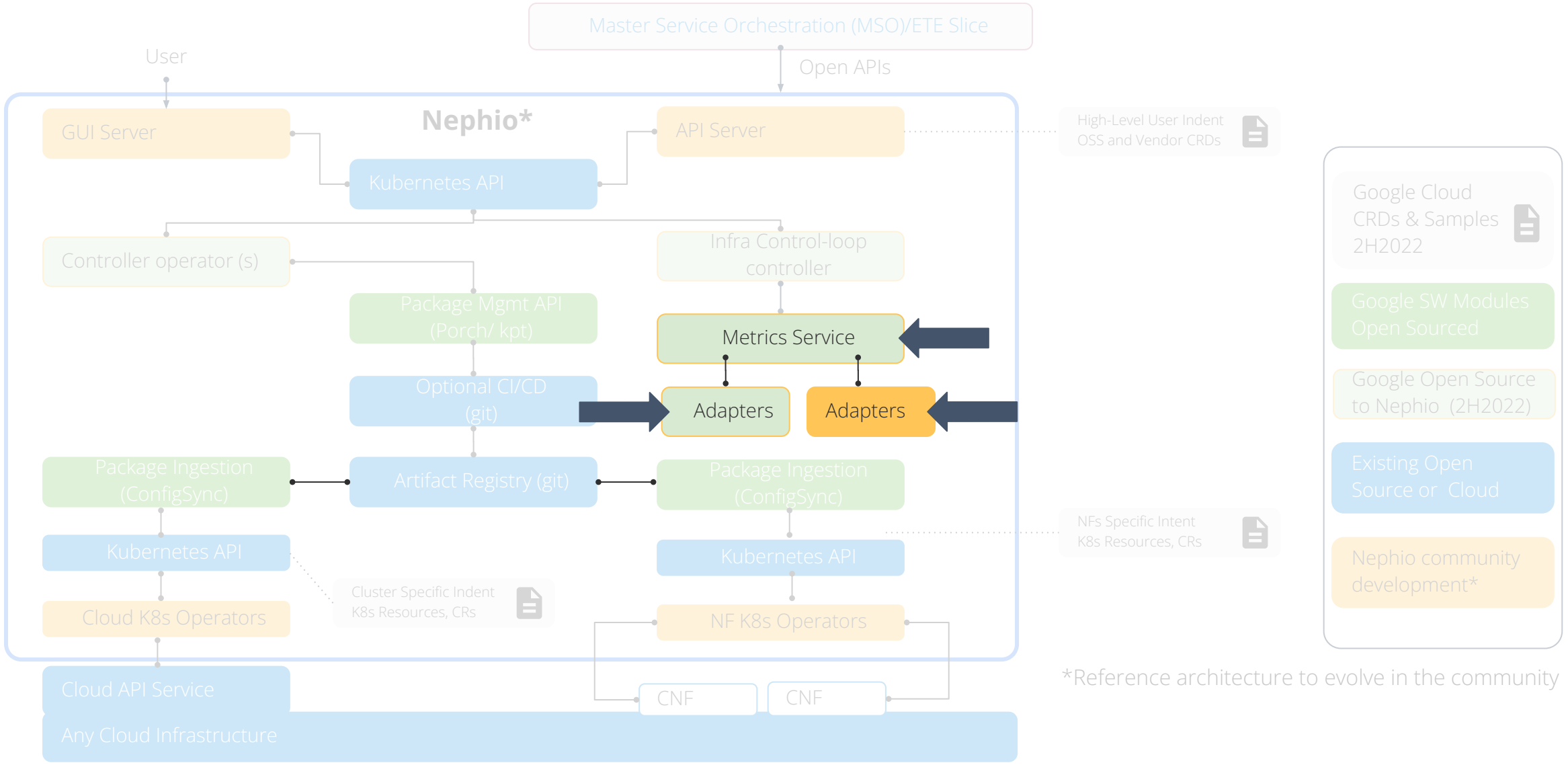
*Reference architecture to evolve in the community

Cloud and Network Function Operators



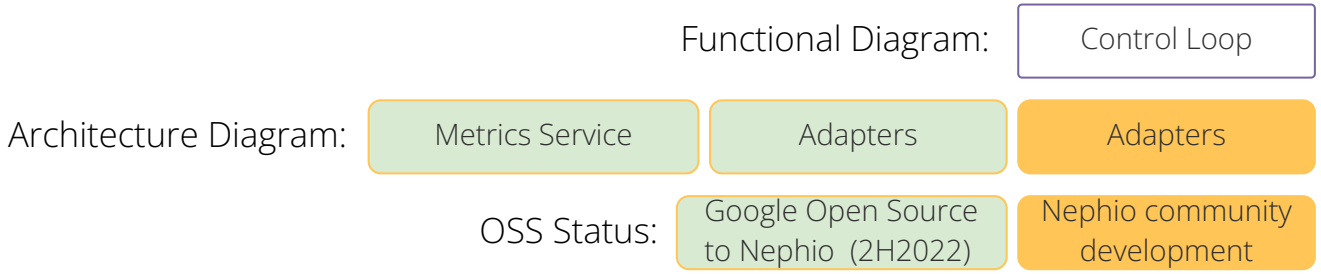
- Cloud K8s Operators bridge Nephio with cloud providers
  - Rely on existing cloud provider Kubernetes operators
  - Connect Nephio CRDs to Cloud Provider CRDs
  - Google with open source Nephio / GCP CRDs and controllers
- Network Function Operators manage specific NFs
  - Nephio builds CRDs, operators, toolkits for common portions
  - Vendors provide CRDs, operators for their NFs
- Nephio `kpt` functions for manipulating and validating Nephio resources

Nephio High-Level Architecture



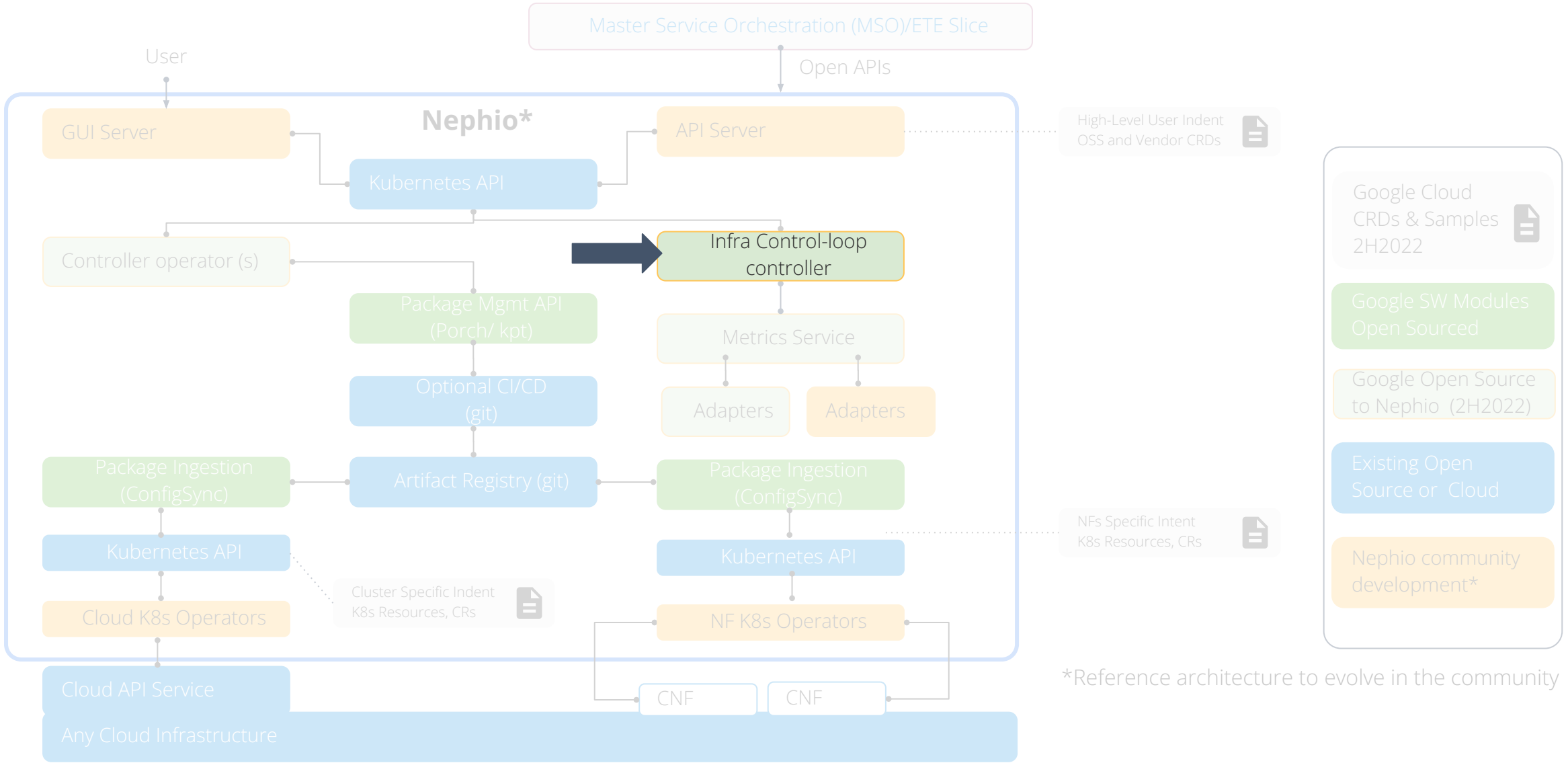
*Reference architecture to evolve in the community

Metrics Service and Adapters



- Report metrics in a metrics backend neutral way
- Adapters can normalize common metrics from cloud providers and network functions
- Status and metrics are aggregated across edge clusters to provide a holistic view

Nephio High-Level Architecture



*Reference architecture to evolve in the community

Nephio Control Loop

Functional Diagram: Control Loop

Architecture Diagram: Infra Control-loop controller

OSS Status: Google Open Source to Nephio (2H2022)

- Runs in the Nephio cluster
- Interprets metrics to determine potential config improvements
- Suggests config changes which may then be approved by deployers
- Examples:
  - Identify over and under provisioning
  - Identify workload placement optimizations
  - Suggest cluster-specific or upstream intent revisions

Nephio Release Artifacts

What will we actually ship?

- Binaries and container images for Nephio-developed components
  - Nephio Controller Manager
  - Nephio kpt functions, Google Infra and Function Operators
  - Nephio Metrics Proxy and Status Aggregation Service
  - Nephio GUI service, CLI
  - Nephio Control Loop and Analytics Service
- Kpt Packages ands docs for Installing and Configuring Nephio
  - Core Nephio Components
  - Cloud-provider Specific packages for infrastructure automation

How will I run it?

- Bring-your-own Kubernetes cluster for Nephio
- Apply kpt packages for Nephio and cloud provider infra automation
- Nephio can then be used to provision edge clusters and functions on them

Let’s Get Started!

- Much of this work falls under **SIG Platform**
- Reference Implementation, Packaging, Installation
- Ready for design and planning contributions now
- Execution work will flow from that
- Birds-of-a-Feature session tomorrow for those interested in contributing