Google Cloud

Reference architecture:

GKE Enterprise hybrid environment (part 2) - implementation details

Part 1 - Architecture, reference deployments, design prerequisites and considerations

December 2024

Table of contents

Overview

Organizations that embrace cloud-first technologies like containers, container orchestration, and service meshes, often reach a point where they need more than a single Kubernetes cluster. Many organizations that use Google Cloud also want to run workloads in their own data centers, factory floors, retail stores, or even in other public clouds.

However, operating multiple Kubernetes clusters has its own difficulty and overhead in terms of consistent configuration, security, and management. For example, manually configuring one Kubernetes cluster at a time creates risks, and it can be challenging to see exactly where errors are happening.

GKE Enterprise is Google's cloud-centric container platform for running modern apps anywhere consistently at scale. GKE Enterprise can help organizations by providing a consistent platform that lets them:

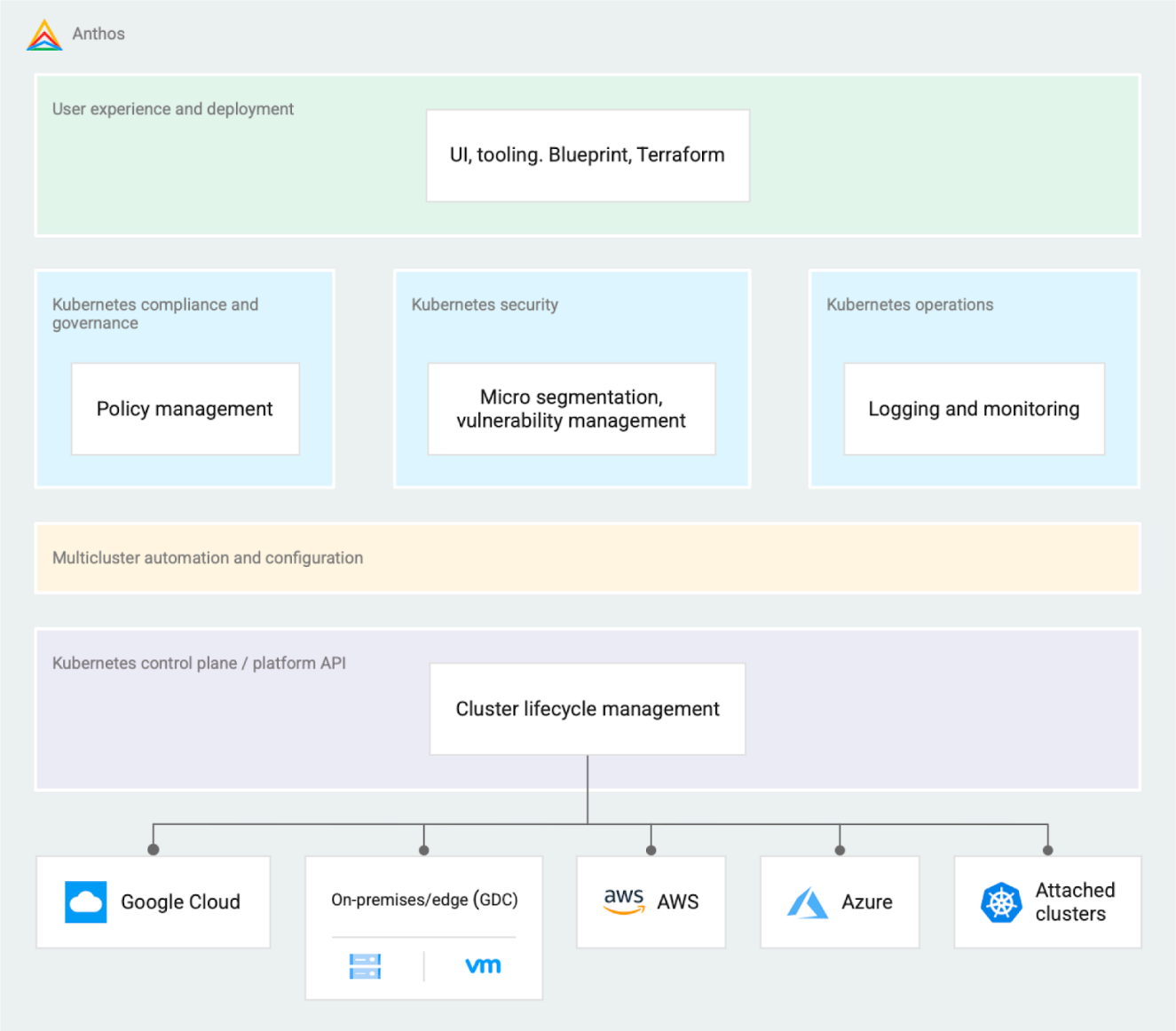
- Modernize applications and infrastructure in-place.
- Create a unified cloud operating model (single pane of glass) to create, update, and optimize container clusters wherever they are.
- Scale large multi-cluster applications as *fleets* - logical groupings of similar environments - with consistent security, configuration, and service management.
- Enforce consistent governance and security from a unified control plane.

GKE Enterprise helps you increase operational consistency in governance and security and developer velocity while reducing cost, deployment risk, and operational complexity. Specifically, GKE Enterprise helps with the following areas:

- Customers who want cloud-like experience on-premises or are looking for a unified solution while migrating their applications to cloud (***GKE Enterprise hybrid environment***).
- Google Cloud customers who want to better manage their containerized applications (***GKE Enterprise on Google Cloud***).
- Customers who want to solve multicloud complexity with a consistent governance, operations, and security posture (***GKE Multi-Cloud***).

The following diagram shows a high-level overview of GKE Enterprise in a hybrid environment. GKE Enterprise can help with Kubernetes compliance and governance, security, and operations, along with

multi-cluster automation and configuration. Kubernetes clusters managed by GKE Enterprise can then run on-premises, in Google Cloud, or in another cloud provider:



This reference architecture provides opinionated guidance to deploy GKE Enterprise in a hybrid environment to address some common challenges that might face.

In this reference architecture, the term *cluster* means a Kubernetes cluster managed by GKE unless stated otherwise. For example, some sections discuss *VMware vSphere clusters* composed of ESXi servers that pool compute resources.

This reference architecture is separated into two parts. Make sure you read both parts carefully to learn how to plan, design, and implement your own GKE Enterprise hybrid environment:

- Part 1 - Architecture, GKE Enterprise components, reference deployments, design prerequisites, and design considerations.
- Part 2 (this document) - Implementation details.

Implementation details

To help you successfully deploy a GKE Enterprise hybrid environment that follows this reference architecture, this section contains some important implementation details. These details include the suggested Config Sync repository structure, recommended project and cluster permission and role assignments, and application namespace examples.

Per-site preparation

Services

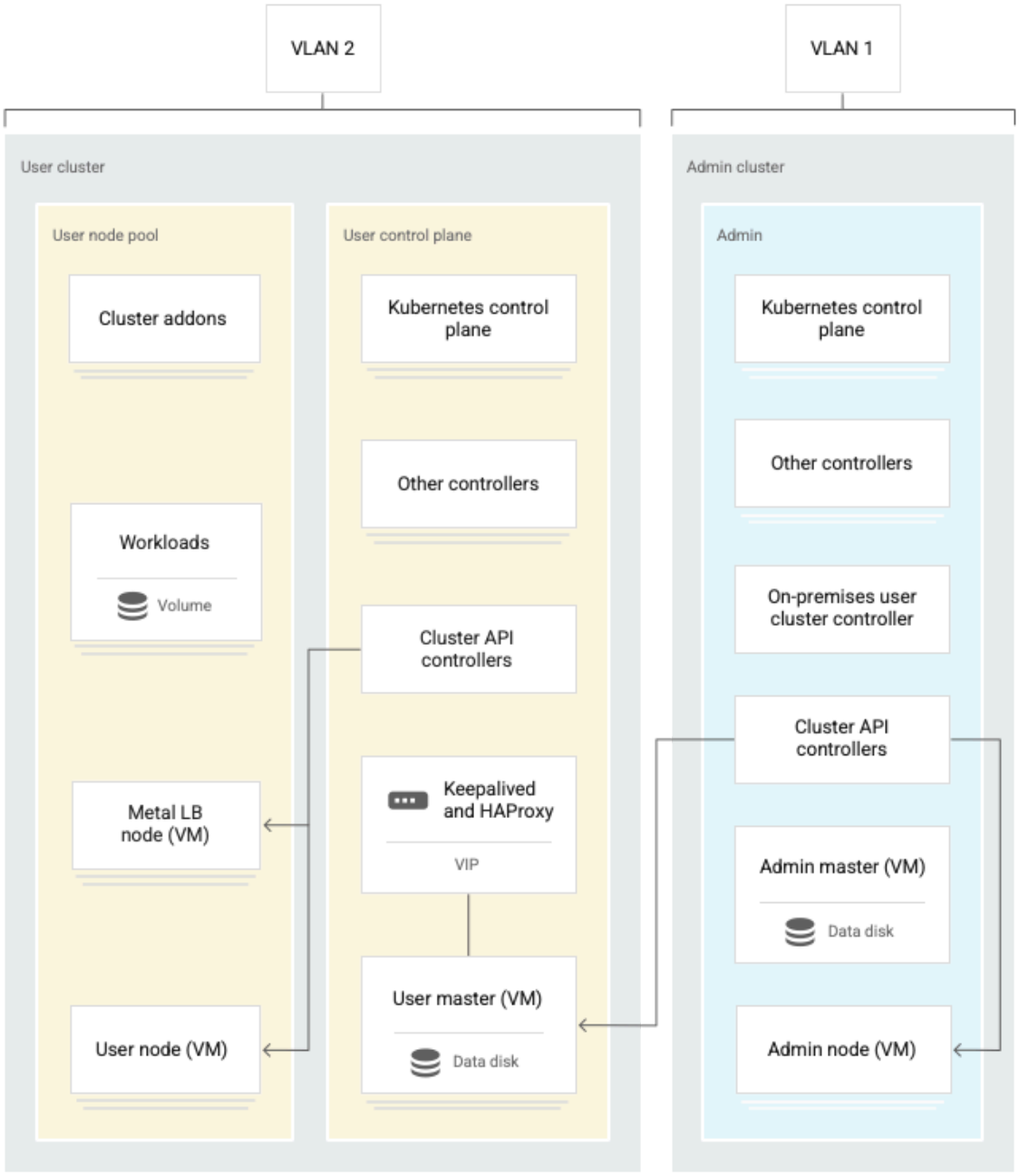
Create an image repository in each site, such as using JFrog Artifactory or Harbor. This site-level repository isn't part of the GKE deployment or supported by Google. If you use Artifact Registry, you could set up the site to replicate from the cloud. Or, you could replicate between local repositories across sites. This approach stores artifacts close to where they're needed for deployments, and lets you replicate a single source of truth across all sites with similar local repositories.

Networking

Place each admin cluster on its own VLAN. Place each user cluster on its own VLAN. The following guidance also applies:

- For GKE on VMware, all nodes of a user cluster should be on the same broadcast domain (one subnet and VLAN).
- For GKE on bare metal, it's simpler if all the nodes of a user cluster are on the same broadcast domain. Multiple domains may be used if necessary, with the following additional guidance:
    - All the load balancer nodes must be on the same broadcast domain.
    - Routing traffic between the L2 domains is your responsibility. GKE doesn't configure this routing.
- Avoid using VLAN-per-node-pool as a security mechanism for separating individual applications in a cluster. This configuration can be complex to manage, and can limit the cost-efficiency benefits otherwise expected from moving to containers. Instead, use Kubernetes Network Policies and Service Mesh L7 Authorization to meet isolation requirements.
- All virtual IP addresses (VIPs) must be in the load balancer machine subnet and routable to the gateway of the subnet.

The following diagram shows an example of how the admin cluster and user clusters should be in their own VLANs. The user control plane and user node pool components share the same VLAN. The user cluster components communicate to the admin cluster and VLAN through the admin cluster API controllers:



Dataplane v2

Use Dataplane V2 in your deployments. GKE Dataplane V2 is a data plane that's optimized for Kubernetes networking, and is based on eBPF on Linux and Open vSwitch on Windows nodes. Dataplane V2 lets you flexibly process network packets in-kernel using Kubernetes-specific metadata.

Handling high traffic

If a site-level load balancer like F5 is available and a cluster is expected to handle a lot of traffic, balance traffic across several VIPs of the same cluster. Use MetalLB to balance within the cluster.

Avoid using only a single ingress to handle large numbers of backend services, such as 2,000 services. Instead, create several ingresses.

Configure vSphere

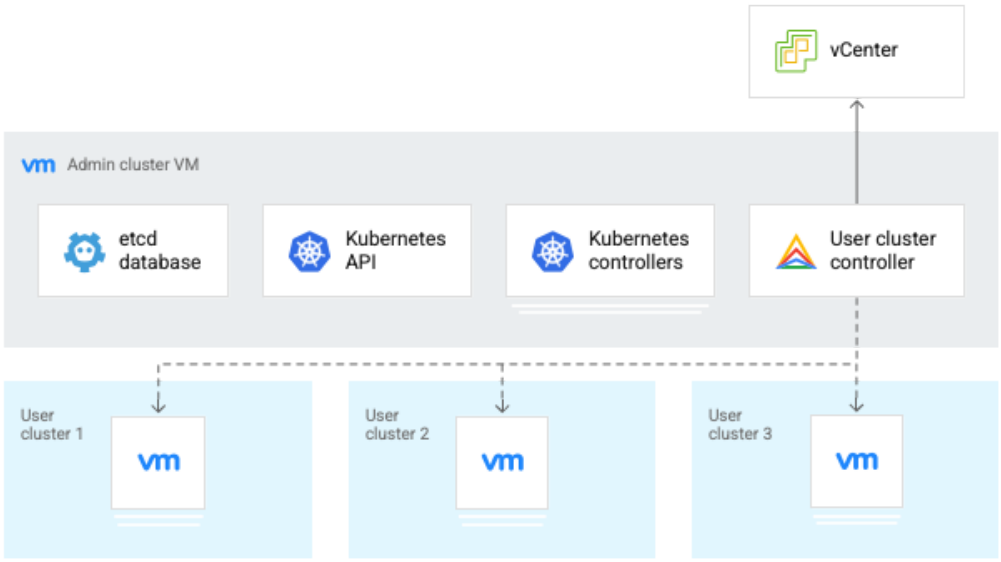
When using vSphere with GKE on VMware, configure as follows:

vCenter user account privileges	vCenter Server Administrator role is *not* required for GKE deployment after the vSphere environment is set up. For GKE deployments, we recommend creating several roles with varying degrees of privilege to limit access to your vCenter environment.
vCenter settings	• Enable vCenter High Availability (HA) • Enable vMotion • Enable vSphere HA Host Monitoring[1] with `Host Failure Response` set to `Restart VMs` • Disable vSphere Storage DRS[2]

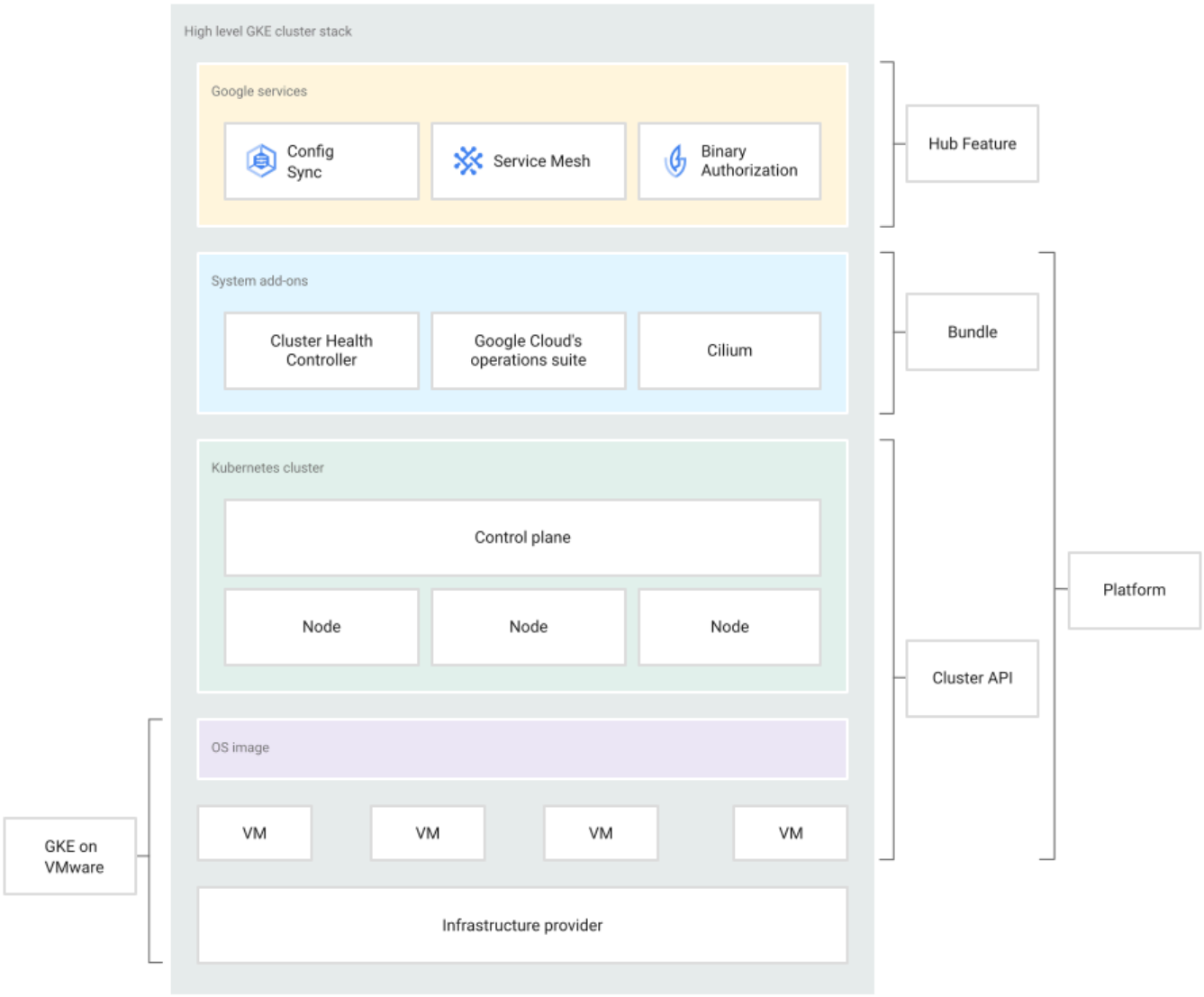
Use the following options when creating GKE clusters on VMware:

- `enableControlplaneV2: true`
- `enableDataplaneV2: true`
- `antiAffinityGroups.enabled: true`

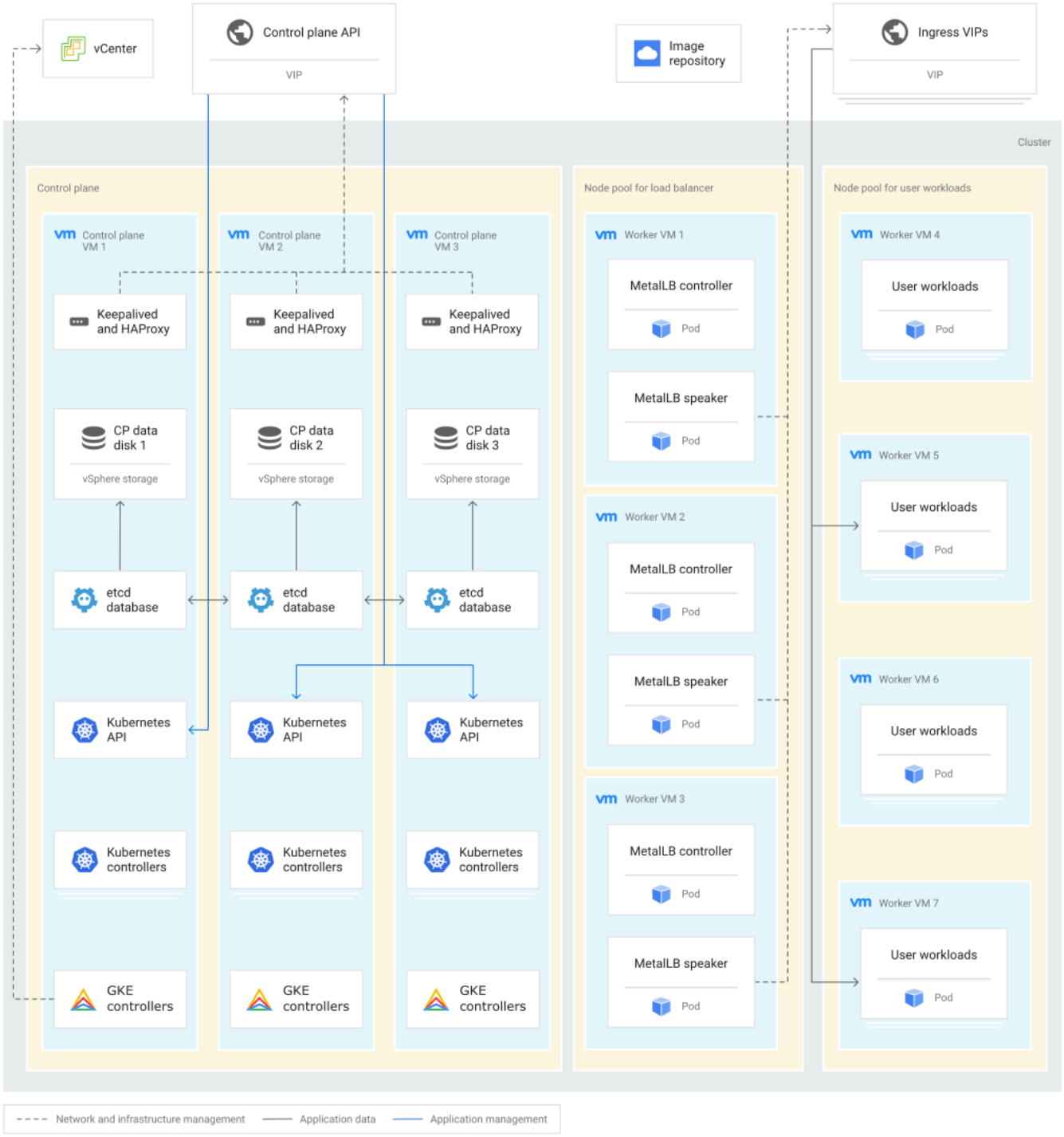
The following diagram provides an overview of how GKE on VMware looks in a deployed state. The user cluster controller in the admin cluster communicates with vCenter. This connection lets the controller create the user cluster VMs:



The following diagram shows a more complete example of GKE on VMware deployed in a hybrid environment. Additional services like Config Sync, Binary Authorization, and Operations Suite supplement the on-premises GKE services that run in VMware:

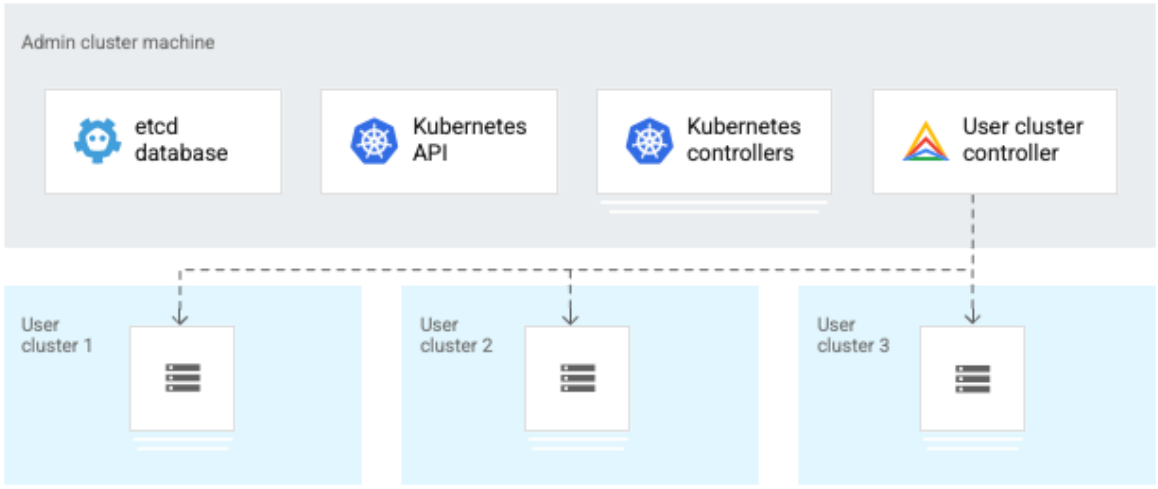


The following diagram shows that the control plane VIP sends traffic to the Kubernetes API. Keepalived / HAproxy keep the control plane VIP pointed at a working control plane VM. Node pools are created by a controller in the user cluster that talks to vCenter. The control plane VMs are created by the admin cluster. There's a dedicated node pool for MetalLB pods, but using the same VLAN as the other node pool. The MetalLB pods communicate using ingress VIPs, which also provide inbound traffic for the user workloads:

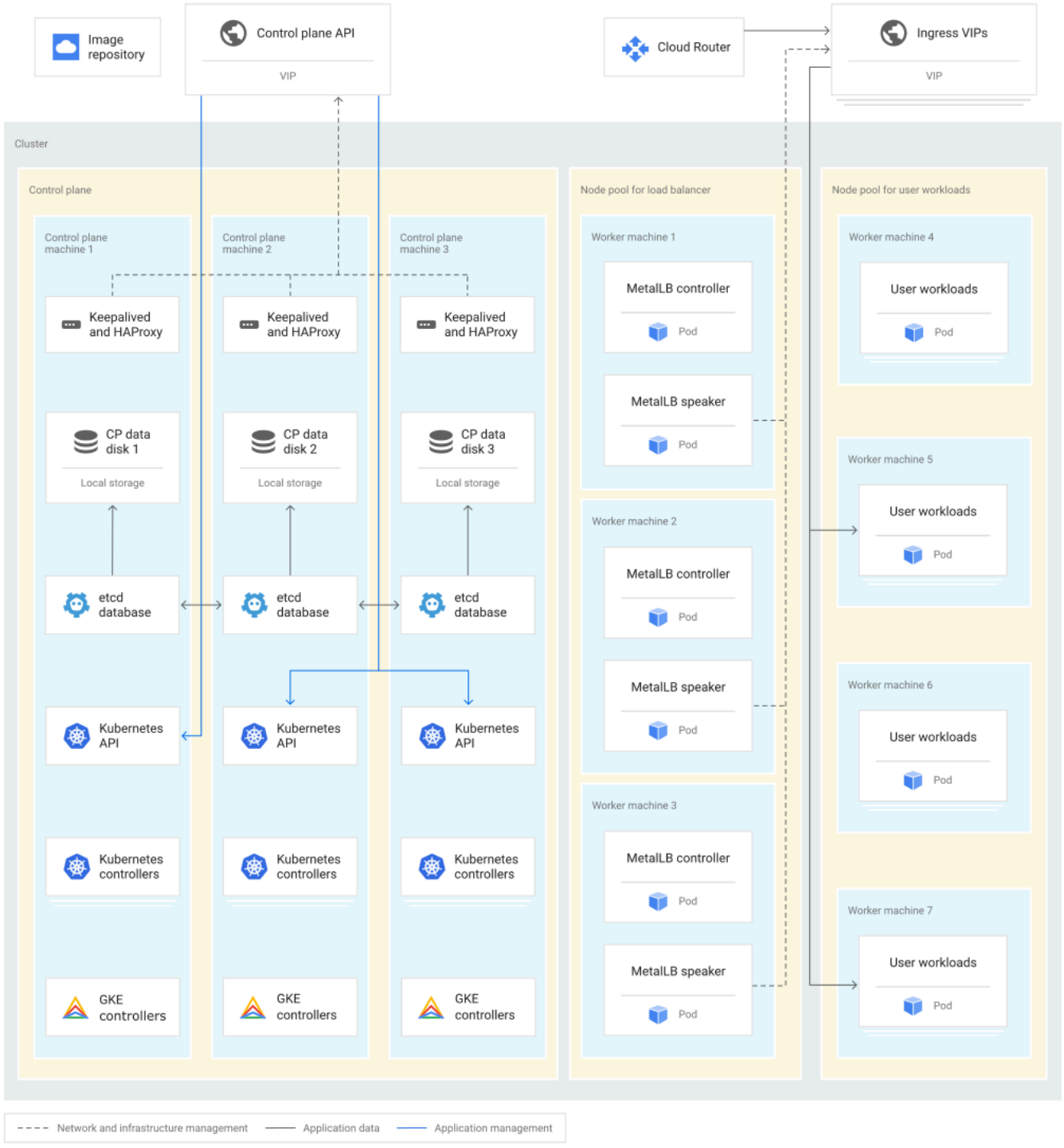


Bare metal deployments

The following diagram provides an overview of how GKE on bare metal looks in a deployed state. The user cluster controller in the admin cluster communicates with the physical machines in your environment. This connection lets the controller create and manage the physical user clusters:



The following diagram shows that the control plane VIP sends traffic to the Kubernetes API. Keepalived / HAproxy keep the control plane VIP pointed at a working control plane machine. Node pools are created by a controller in the user cluster. The control plane machines are created by the admin cluster. There's a dedicated node pool for MetalLB pods, but using the same VLAN as the other node pool. The MetalLB pods communicate using ingress VIPs, which also provide inbound traffic for the user workloads:



Fleet management

- Place the prod clusters and their admin clusters in the production fleet, `project-2-fleet-prod`.

- Place the staging cluster and its admin cluster in the staging fleet, `project-3-fleet-staging`.

Hardware

GKE can run on a wide range of customer-provided hardware. When you run on virtual machines, they can be easily sized to their roles.

If you deploy GKE on physical machines, it can be more efficient to include at least some medium machines instead of all large-sized machines. Large machines are those that include more RAM and number of vCPUs than medium machines. Medium-sized physical machines have better utilization when they run as user cluster control plane nodes and admin cluster nodes.

Operating system

For GKE on bare metal, the base operating system on the nodes is customer-managed. Only install those OS packages that are prerequisites for GKE, are needed to monitor the hardware and operating system, or to debug issues.

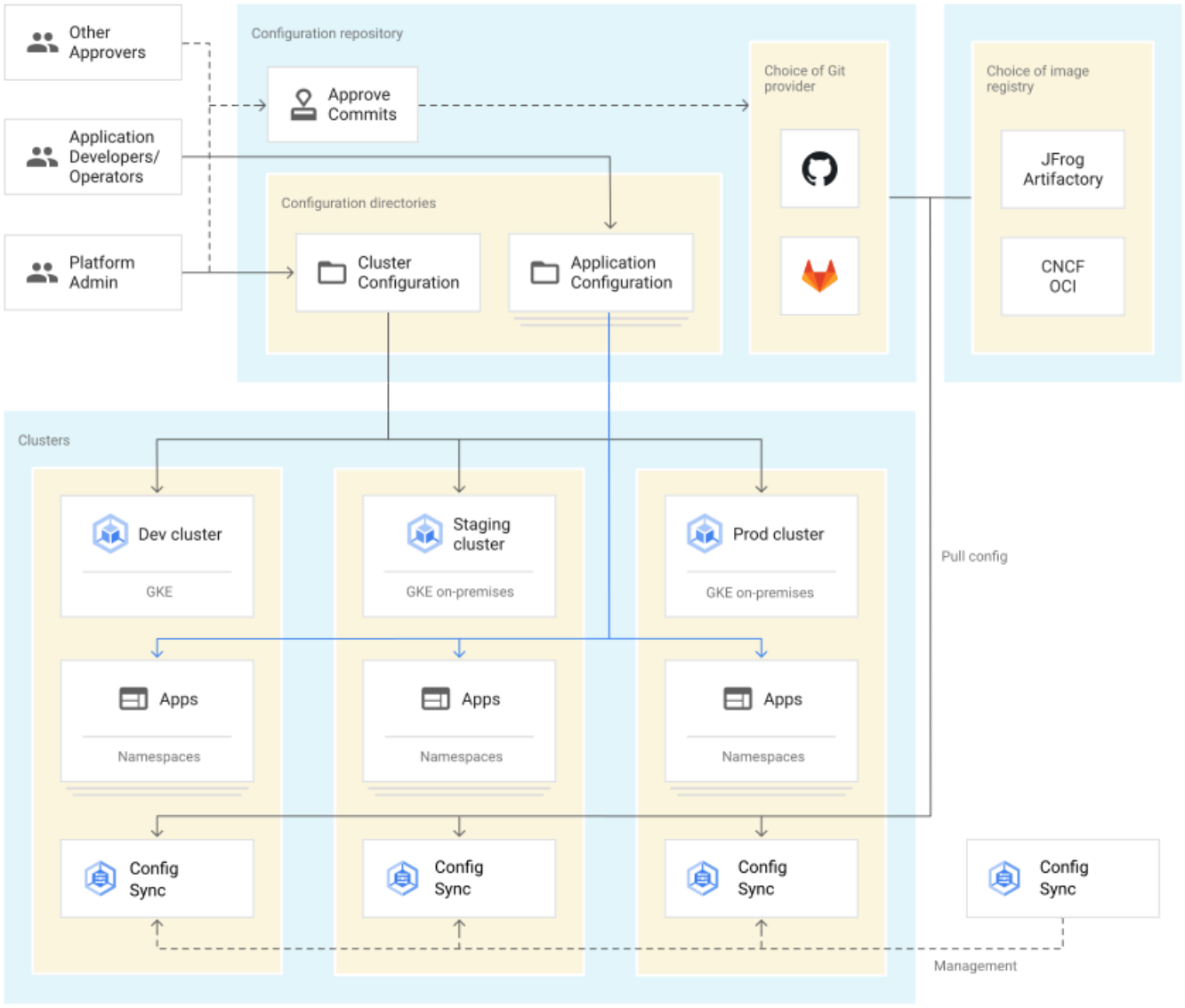
Because applications are containerized, they largely don't depend on libraries or services that run on the base operating system. Instead, application dependencies such as libraries are managed at the container image level.

Config Sync

Config Sync is used to manage Kubernetes objects in all the clusters. Config Sync is a GitOps-style tool. GitOps is a process where a Git source control repository is the source of truth for configuration. Git provider workflows allow multiple stakeholders to participate in review of changes.

Config Sync can pull configurations directly from a Git repository by using an agent in every cluster. It can also pull bundled files from an image registry.

As shown in the following diagram, the recommended Config Sync deployment uses one folder containing configuration for all clusters. Other individual folders each hold configuration data for one application:



A Config Sync component per cluster pulls configuration directly from a source control repository or an image registry. A cloud management plane lets you centrally configure data sources and what tag or version of an application to pull. This cloud management plane means that you don't have to manually interact with each GKE cluster using the Kubernetes API to make individual config or application changes.

Config Sync manages in-cluster resources that you define and deploy. Resources that affect the whole cluster (cluster-wide or cluster-scoped) are managed differently from resources that only affect one namespace (namespace-scoped). The latter are used to define applications. For GKE on bare metal, cluster configuration drift is enabled that detects configuration drift of the core component manifest files. This configuration drift feature is not designed for managing the state of your own components, Services, and Deployments. You still use Config Sync to manage the state of your own resources.

Different categories of resources each have a set of tooling:

- Google Cloud resources are managed with the Google Cloud console, `gcloud` CLI, or Terraform.
- In-cluster resources are managed with Config Sync.
    - Config Sync doesn't manage admin clusters and their resources.

This reference architecture uses a specific subset of Config Sync features. Additional features exist and can be used where appropriate. In particular, this reference architecture uses the following choices in the use of Config Sync:

- Uses unstructured mode.
- Doesn't use cluster selectors or namespace selectors. Therefore, cluster labels also aren't required.
- Configuration can be specified and delivered to each cluster as a template with parameters substituted with per-cluster values. Or, use a fully rendered, or *hydrated*, configuration to each cluster.
    - A fully hydrated configuration allows for users to understand what the desired state is, without having to mentally run a transformation process.
- Uses Git as the source of truth. Config Sync also supports using an OCI image repository[3] or a Helm repository[4] to pull configurations from.

Cluster configuration

Platform owners define what policies are deployed to all clusters. Policies include the following:

- **PodSecurity admission controller:** Includes preventing Pods from using the root Linux user.
- **NetworkPolicies:** Control the network traffic inside your clusters.
- **ClusterRoles and ClusterRoleBindings:** Control permissions within a cluster.
- **Service Mesh:** Includes policies such as for authorization, transport security, or security policy constraints.
- **Policy Controller:** Install the Policy Controller security bundle for Service Mesh[5].
- User permissions (RBAC) granted cluster-wide.

The configuration for each cluster is stored in a Root Repository, also known as a RootSync. The Root Repository configuration includes both literal configuration resources, and pointers to other repositories holding application resources.

Application configuration

Platform owners can control the specifics of each application, or delegate control to application owners.

Applications repositories, also referred to as *Namespace Repos* or *RepoSyncs*, configure resources within a single namespace, including the following:

- Workloads such as Deployments and StatefulSets.
- VolumeClaims
- RBAC permissions scoped to namespace, such as Role and RoleBinding.
- Services
- Service Mesh configuration for services in this namespace.

This reference architecture offers a choice of two patterns for deploying application configuration. Each approach has advantages and disadvantages:

- Helm templates:
  - Simple to set up.
  - Uses familiar templating patterns.
  - Templates can become complex to reason about over time.
  - Parameters substituted by Config Sync in each cluster.

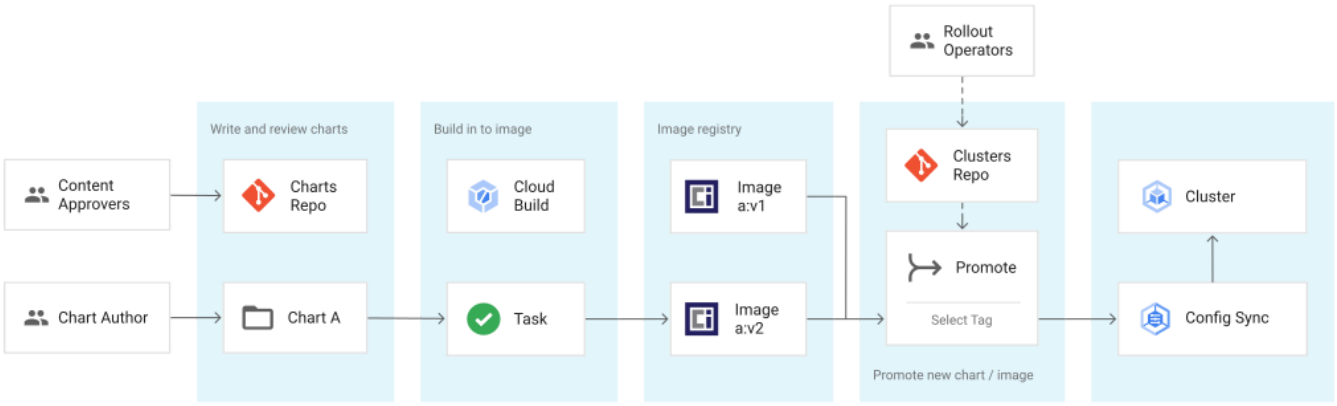
- Fully hydrated configuration:
  - More steps to set up.
  - Can use templating, or other config customization tools like `kustomize` or `kpt`.
  - Validation tools can run directly on the configuration during the review process.
  - Requires setting up a rendering pipeline that reads and writes to Git.
  - Both *dry* and *hydrated*, or wet, configurations are stored in Git.
    - Dry means configuration that uses templates and doesn't have per-cluster or per-app customizations applied.
    - Hydrated, or wet, mean configuration with all template parameters substituted and per-cluster and per-app customizations applied.
  - Storing both types of config in Git has the following advantages:
    - The Git repository is a source of truth, and you don't need to mentally run a transformation process to be sure of the desired state.
    - More flexibility in what processes can be used.
    - Greater scope to validate configuration during the review process.

Helm templates

The Helm templates approach is appropriate for teams that already use the popular open source Helm tool to define applications. In the Helm approach, the following benefits and management approaches can be used:

- Application owners specify their applications as Helm Charts, with a limited number of parameters (Values).
- Platform admins and operators control which application instances are deployed in which clusters and namespaces using the Root Repo.
- Use Helm templating language to parameterize charts per cluster.
- Platform admins and operators control parameter values by cluster.
- Platform admins and operators control promotion of new chart versions (rollout across clusters).
- Anthos Config Management pulls hydrated config from Git and charts from an image registry. Helm configuration and container images can be stored in the same registry.
- Platform policy is validated before deployment, and again when applied to the cluster.

The following diagram shows how you can write and review Helm charts that can build images to be stored in a registry. You can then deploy the Helm carts to the cluster using Config Sync:



Fully hydrated configuration

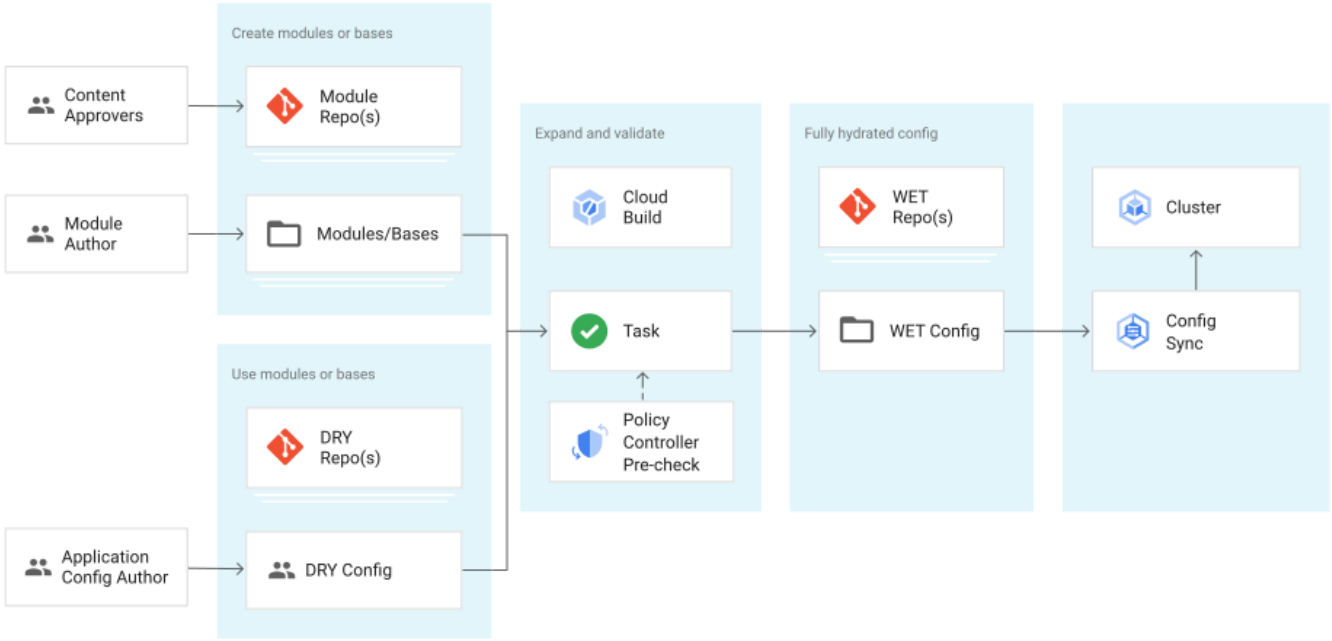
The fully hydrated pattern works well when a central platforms team wants to create modules and allow other teams to modify them, subject to code review.

In the fully hydrated approach, the following benefits and management approaches can be used:

- Platform admins and operators define a base application definition.
- Applications define instantiations with changes to the base definition using the platform owner's choice of packaging, templating, or customization tool, such as `kustomize` or `kpt`.

- Platform owners review modifications to the base application definition, but don't necessarily explicitly have to manage parameters.
- A pipeline renders the packages, templates, and customizations to a fully hydrated version of the configuration.
- Config Sync takes the configuration verbatim from the repository. This approach is a key advantage of the fully hydrated approach as there's no interpretation of your desired configuration.
- Platform policy is validated both before deployment in the expansion pipeline, and again when applied to the cluster.

In the following diagram, the modules define tasks to be performed that generates a final configuration that is applied to clusters using Config Sync:



Install considerations

The following steps are performed once to provide a foundation for the deployment:

- Create a Git repository for the root config, such as `$ORGNAME/config-sync-root`
- Create sub directories for each fleet, such as `fleet-prod` and `fleet-nonprod`
- Enable Config Sync on all clusters of the fleet.
    - This step can be done with the Google Cloud console or with Terraform.

The following steps are performed as each cluster is created:

- Create a directory for that cluster within the root config Git repository, such as `fleet-prod/cluster-abc01-prod`.
    - Each cluster name must be unique across the organization.
- Set up configuration sync for that cluster.
    - You can use the Google Cloud console or Terraform.
    - When using Terraform, create a `member_feature`. The membership has already been created when the cluster was, and doesn't require import. Instead, it's an output parameter of the `gkeonprem` user cluster.
- The path within the repository is different for each cluster:
    - For example, `fleet-prod/cluster-abc01-prod`. This path is referred to as adding a RootSync to the cluster.
- Other settings can be the same across sites, such as the following:
    - All clusters use the same URL, like `http://$GITPROVIDER/$ORGNAME/config-sync-root`
        - If Git is mirrored or separated across sites, `$GITPROVIDER` can be different per site.
    - All clusters use the same branch, such as HEAD.
    - Set Unstructured mode.

When these steps are done and several clusters have been created, the directory structure for `config-sync-root` looks like the following example:

```
fleet-prod/
  cluster-abc01-prod/
    asm/
    policy/
    storage/
  cluster-xyz01-prod/
    asm/
    policy/
    storage/
fleet-nonprod/

cluster-abc01-staging/
    asm/
    policy/
    storage/
```

Related policy objects can be grouped into directories by concern. In this example, categories `asm/`, `policy/`, and `storage/` are used.

Application deployment

The following steps are performed for each application that you deploy:

- Create a Git repository to hold that application's configuration.
  - For example, if the application's name is $X, make a repository `$ORGANIZATION/app-config-$X`.
  - Use multiple Git repositories to reflect administrative boundaries. Each team should work in only one repository and ideally different teams have their own repositories.
  - To make sure that changes can be reviewed by all stakeholders, use the `CODEOWNERS` feature of GitHub or GitLab.
- For each environment and site where the app should run, complete the following steps:
  - Create a namespace in the format like `$appname-$env-$site` by adding it to the Root Repo
  - Create a `reposync.yaml` file in the namespace directory, such as `cluster-xxx/namespaces/app-site-env/reposync.yaml`
  - For the Helm approach, point to the artifact registry path and image name (method). Set cluster-specific values.
  - For the fully hydrated approach, set the Git repository and directory for the namespace. A single Git repository can be used for Root and Application repositories.

After deploying two applications, the `anthos-acm-root` directory looks like the following example:

```
fleet-prod/
  cluster-abc01-prod/
    asm/
    namespaces/
      team1/
        app1/
          reposync.yaml
          rbac.yaml
        app2/
          reposync.yaml
          rbac.yaml
    policy/
    storage/
  cluster-xyz01-prod/
    asm/
    namespaces/
      team1/
        app1/
          reposync.yaml
          rbac.yaml
        app2/
          reposync.yaml
          rbac.yaml
     policy/
    storage/
```

```
fleet-nonprod/
  cluster-abc01-staging/
    asm/
    namespaces/
      team1/
        app1/
          reposync.yaml
          rbac.yaml
        app2/
          reposync.yaml
          rbac.yaml
    policy/
    storage/
```

Each `reposync.yaml` file follows one of the following two patterns:

1. A Git repository and path holding fully hydrated configuration, like the following example:

```
# File: fleet-prod/cluster-abc01-prod/namespaces/app1/reposync.yaml
apiVersion: configsync.gke.io/v1beta1
kind: RepoSync
metadata:
  name: app1
  namespace: config-management-system
spec:
  sourceFormat: unstructured
  sourceType: git
```

```
  git:
   repo: https://gitmirror.abc01.company.com/team1/app1-config/
   branch: main
   auth: token
   secretRef:
      name: git-repo-secret
```

2. An OCI image that holds a Helm chart and has a tag, and Helm parameters like the following example:

```
# File: fleet-prod/cluster-abc01-prod/namespaces/app1/reposync.yaml
apiVersion: configsync.gke.io/v1beta1
kind: RepoSync
metadata:
  name: app1
  namespace: config-management-system
spec:
  sourceFormat: unstructured
  sourceType: helm
  helm:
    repo:  oci:/imgreg.abc01.company.com/team1/manifests/app1:v1.1
    chart: app1
    version: v2
    releaseName: instance1
    namespace: app1
    auth: token
    secretRef:
      name: helm-repo-secret
    values:
      param1: 1234
      param2: asdf
```

Roll out of cluster configuration

Determine an order of updates for clusters. Update staging before production, and one site at a time. Apply edits to each cluster subdirectory at a time, such as in the following order:

1. `fleet-nonprod/cluster-abc01-staging`
2. `fleet-prod/cluster-abc01-prod`
3. `fleet-prod/cluster-xyz01-pro`

Roll out of an application configuration

Use a development cluster to test changes. There can be one cluster for each application team. These clusters should use the same base configuration or Helm chart, but be parameterized to use test data and dependencies. Refactoring applications so that they can work in different environments with similar configuration is an important part of achieving frequent successful deployments.

Complete the following steps to deploy an application at a new image tag:

- Helm approach:
  - The app is built into an image with a new tag.
  - Update the Root repository to use the tag, progressively in a series of clusters, like in `cluster-acb01-staging` and then in production clusters like `cluster-abc01-prod` and `cluster-xyz01-prod`.

- Fully hydrated approach:
  - Modify the parameters that control the image tag in that cluster, such as a `Kustomize.yaml` or Kptfile.
  - Review and commit.
  - Pipeline expands configuration and commits to Namespace repository.
  - Config Sync pulls the fully expanded config from the Namespace repository.
  - Repeat this process progressively in a series of clusters.

Recommended policies

The following policy bundles are recommended:

- `K8sPSPHostFilesystem`[6]
  - This policy ensures that applications don't take unexpected dependencies on the host operating system version, which allows independent application and OS updates. This policy also helps to increase security.
- Consider requiring mesh authorization for each namespace[7].
  - Start with Namespace level, rather than at the workload selector or mesh level.
  - Use EnvoyFilters when finer control of authorization is needed.
- You can also port existing policies from OpenShift[8].

Service Mesh

Service Mesh provides security, observability, and traffic management features both within and across clusters[9]. In this reference architecture, only the within-cluster capabilities of Service Mesh are used.

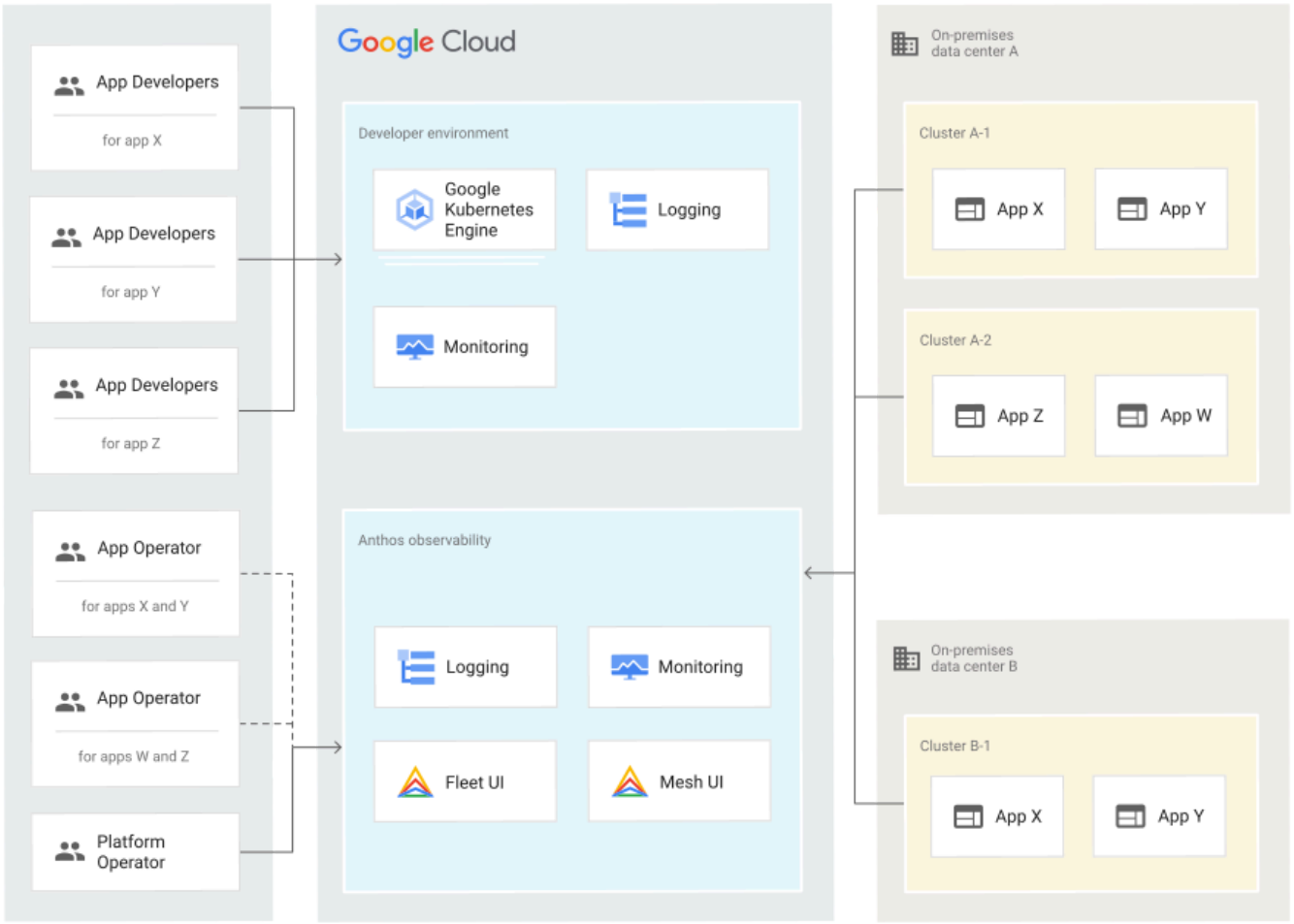
Clusters don't have to connect with each other. However, metrics for equivalent services, those with the same service name and namespace name, can be readily viewed in aggregate.

Configure Service Mesh as follows:

- Service Mesh for GKE clusters is unmanaged. Follow the instructions to set up a multi-cluster mesh. However, don't set up cross-cluster trust[10].
- Enable strict mTLS.
    - As a security best practice, enable strict mTLS at the mesh level[11].
    - To enforce this setting, use the policy bundle with strict mode[12].
- mTLS certificates for in-mesh communication:
    - Protect the CA signing key by selecting either Mesh CA or Certificate Authority Service (CA Service).
        - Use CA service if you have special CA requirements[13]. Otherwise, use Mesh CA.
    - Install Mesh CA or CA Service.
- Certificates for serving ingress traffic:
    - Use external certificate management on ingress[14] and egress[15].
    - Applications can continue to use existing application certificates, such as Let's Encrypt or Google-managed certificates.
- Enable Cloud Monitoring (HTTP in-proxy metrics), which is part of the platform and included as system metrics.
- Implementation of the following examples items is optional and can be enabled when convenient after initial deployment:
    - User authentication with Identity-Aware Proxy.
    - User authentication with your existing Identity Provider.
    - Service Mesh user authentication.

Observability

The following diagram shows how application developers and application or platform operators can view logging and monitoring data in Google Cloud. The on-premises clusters and applications send this logging and monitoring data back to Google Cloud for analysis and review:



The following observability considerations apply to this reference architecture:

- All observability data for production clusters, such as system logs, system metrics, application logs, and application metrics, is directed into `project-2-fleet-prod`.
- All observability data for staging clusters is directed into `project-3-fleet-staging`.
- The platform team may create a dashboard for each application, with application operators given access to that application's dashboard.
    - The dashboard may contain container-level metrics, service metrics (Service Mesh), application-specific metrics if applicable, log-based metrics, and logs panels (if appropriate).
    - Grant members of the team access to their appropriate dashboard. This approach gives access to their metrics without exposing other team's metrics.
    - Use Identity and Access Management (IAM) conditions based on the resource name attached to the project resource, such as `project-x-ops`, if the teams need log access.
- If compatible with company policies, give application operators and application developers `view` access to `project-x-ops`.

    - Application developers and application operators can benefit from debugging other services that they depend on or are dependencies of.
    - Placing all metrics and logs in a single project allows most effective use of Cloud Operations.
    - Placing all metrics and logs in a single project ensures proper tagging of logs and metrics with cluster identifiers.
- Platform teams should create dashboards and grant permissions with IAM conditions on them for one or more appropriate teams to see the dashboard.

Application monitoring

- All application metrics go to the operations project.
- Application metrics are automatically labeled and the labels can be used for aggregation of metrics. Labels include the cluster name, namespace name, project, pod name, deployment name, and user-provided Kubernetes labels. The following Applications section provides more details on recommended labels.
- Applications which provide Service Mesh services also produce service logs. These service logs also go to the operations project.
- You have two options for giving application developers access to monitoring data:
    - Give application developer teams `view` permissions on the namespace and on the cloud resources in this team / function project.
    - Delegate dashboard access with a third-party tool.

System monitoring

We recommended that you set service level indicators (SLIs) to track the health of GKE clusters, and define playbooks for responding to these conditions. The following suggested SLIs cover the ability and performance of the cluster to schedule workloads. These SLIs use metrics that are available in Cloud Monitoring:

- **Control plane responsiveness:**
    - Suggested SLI: 90th percentile control plane request latency
    - PromQL expression:

```
histogram_quantile(0.9,sum(
rate(kubernetes_io:anthos_apiserver_request_duration_seconds
_bucket[5m])) by (cluster_name, le))
```

    - Alert: Establish a baseline value and then alert if 2x above baseline.
    - Response to alert: Control plane latency may increase if an excessive number of objects are created, or if automated processes are sending an excessive number of requests.

- **Node availability:**
  - Suggested SLI**:** Number of nodes not ready for more than 10 minutes.
  - PromQL expression:

```
sum(kubernetes_io:anthos_kube_node_status_condition{condition="Ready",status="true"})/sum(kubernetes_io:anthos_kube_node_status_condition{condition="Ready"})
```

  - Alert: Start by alerting if any nodes aren't ready for more than 10 minutes. Adjust with experience with your workloads.
  - Response to alert:
    - Check last known values of secondary node status conditions, such as with the following PromQL expressions:
      - Memory pressure:

```
sum (kubernetes_io:anthos_kube_node_status_condition{condition="MemoryPressure",status="true"})
```

      - Running out of disk space:

```
sum (kubernetes_io:anthos_kube_node_status_condition{condition="DiskPressure",status="true"})
```

    - Confirm network reachability between unready node and control plane.

- **Scheduler latency:**
  - Suggested SLI: Median end-to-end scheduling latency (the length of time that it takes from Pod creation to when the node name is set on the Pod object).
  - PromQL expression:

```
histogram_quantile(0.5,sum(kubernetes_io:anthos_scheduler_pod_scheduling_duration_seconds_bucket) by (le))
```

  - Alert: Start by alerting if the median scheduling time exceeds one second. Adjust with experience with your workloads.
  - Response to alert:

    - Investigate number of pending pods in scheduler queue (`kubernetes_io:anthos_scheduler_pod_scheduling_duration_seconds`)
    - Investigate pending pod resource status.

- Monitor your cluster networking (DataplaneV2) metrics to avoid exceeding kernel resource limits[16].

Logging

Choose one of the following options for providing logs access to application operator teams:

- Grant application operator teams `view` access to the logs project. This approach works well if a single application operator team already manages most applications and has application logs access.
- Don't grant `view` access to application operator teams. This approach works well if existing policies don't allow application operator teams to access logs.
- Create logging views[17] for each application operator team. Create a filter to control which logs each application operator team can use. This approach works when there are several, but less than 25, different application operator teams that each manage different sets of applications.

Roles and permissions

In this section, permissions are suggested for two different types of organizations.

In the first type of organization, a platform team hides the complexity of operations and infrastructure for developers. Application developers don't directly participate in operating applications in staging and production environments. They might be unaware of facts like their containerized applications are deployed onto Kubernetes, what clusters exist, or how individual applications are replicated. For this type of organization, the *minimum permissions* are recommended.

The second type of organization promotes the development of a *DevOps culture* - a shared sense of responsibility between development and operations teams for the health of services. This approach requires additional permissions for application teams. The *expanded permissions* are recommended for the second type of organization.

GKE Enterprise supports both types of organization, and organizations at various points in between.

The following roles are assumed:

- Application developer team:
    - In the first type of organization, there may be no need for this role to have permissions on any of the resources covered in this document.
    - In the second type, application developer teams are assumed to take a role in observing whether their applications meet service level objectives. To make these observations, teams need some level of access to monitoring, and to see the configuration and status of deployed applications. Teams don't necessarily need to see the logging data itself, which might contain more sensitive information.

    - In larger organizations of the second type, there might be multiple distinct application developer roles, like `app-dev-team-1`, `app-dev-team-2`, or `app-dev-team-3`. Each team might specialize in developing and operating different sets of applications.

- Application operator and SRE team:
    - The application operator team typically sets SLOs or alerts for applications, and responds to application-level issues.

- Network specialists team:
    - The network operations team manages global and local traffic in aggregate, and controls the lower layers of the network stack.

- Platform admins and operators team:
    - The platform team develops and manages a platform which application developers use to deploy applications, while meeting numerous security, governance, reliability, cost, and other constraints.
    - In the first type of organization, the platform team may also take on the application operator role.
    - In the second type of organization, the application operators are typically a distinct role.

Project permissions

The following table outlines the roles that each persona should be assigned to the various projects used in this reference architecture:

Role	Project	Minimum permissions	Expanded permissions
Application developer team *xyz*	`project-0-net`	∅	∅

	`project-2-fleet-prod`
	
	`project-3-fleet-staging`
		∅	`roles/monitoring.viewer`
			
			`roles/gkeonprem.admin`
			`roles/gkehub.viewer`
			`roles/gkehub.gatewayEditor`
			`roles/gkeonprem.viewer`

	`project-n-app-xyz`	∅	`roles/viewer`
Application operations and SRE team for application of team *xyz*	`project-0-net`	∅	`roles/viewer`

	`project-2-fleet-prod`
	
	`project-3-fleet-staging`
		`roles/monitoring.viewer`
		
		`roles/gkehub.viewer`
		`roles/gkehub.gatewayEditor`
		`roles/gkeonprem.viewer`
			`roles/monitoring.viewer`
			
			`roles/logging.viewer`
			or
			`roles/logging.viewAccessor`
			
			`roles/gkehub.viewer`
			`roles/gkehub.gatewayEditor`

			roles/gkeonprem.viewer roles/logging.viewer
	project-n-app-xyz	roles/viewer	roles/editor
Network specialists team	project-0-net	roles/viewer	roles/owner
	project-2-fleet-prod project-3-fleet-staging	∅	roles/monitoring.viewer
	project-n-app-n	∅	roles/viewer
Platform admins and operators team	project-0-net	roles/monitoring.viewer	roles/monitoring.viewer
	project-2-fleet-prod project-3-fleet-staging	roles/owner	roles/owner
	project-n-app-xyz	roles/owner	roles/owner

Some application development teams might develop multiple applications, and some application operators and SRE teams might operate apps from multiple development teams. In the previous table, xyz refers to one or more applications or groups of closely related applications. Some individuals might also work on multiple teams.

Cluster permissions

The following table outlines the permissions that each persona should be assigned to the clusters used in this reference architecture:

Role	Cluster type	Minimum permissions	Expanded permissions
Application developer and operator teams *xyz*	Admin cluster	∅	∅
	User cluster (production)	∅	Cluster read-only[1]
	User cluster (non-production)	Namespace read-only[1]	Cluster read-only Namespace read-write[1]
Network specialists team	Admin cluster	∅	∅
	User cluster (production)	∅	∅
	User cluster (non-production)	∅	∅
Platform admins and operators team	Admin cluster	Cluster admin [1]	
	User cluster (production and non-production)	Cluster admin [1]	
Google	Admin cluster	Read-only [2]	

Support (Google employees)	User cluster	∅	Read-only [2]

1: Generate with `gcloud container hub memberships generate-gateway-rbac` (ref)
2: Generate with `gcloud container hub memberships generate-gateway-rbac –anthos-support` (ref)

Applications

Namespaces and app projects

Use the following guidance for naming namespaces and projects:

- When you create a new application, give it a descriptive name prefix, such as `shoppingcart` for a shopping cart service.
- Create a project to hold Google Cloud resources related to the application, like images, such as `project-5-shoppingcart-app`.
- Compose a namespace name which is the same as the application name. In this example, the namespace would be called `shoppingcart`.
- Typically, you use the global DNS load balancer to balance incoming traffic across sites for the shopping cart service.

Some services might run at multiple sites and are fungible, meaning that any one of them can serve the same request. In this scenario, use the following pattern:

- Use the same namespace name in all clusters that receive a share of the same traffic from the global traffic manager.
- Use the same service name at each site.

As an example, you might run the `shoppingcart` service in cluster `cluster-abc01-prod` and `cluster-xyz01-prod`. Both clusters serve requests for the shopping cart portion of the same website, so both clusters use the same namespace name.

For a set of services that have similar configuration, but handle different types of requests or data, use different namespace names for each instance. For example, if three separate database instances need to be deployed to hold user, product, and sales data, put them in namespaces `user-db`, `product-db`, and `sales-db`.

During version rollout, Service Mesh can be used to do blue-green update, or Kubernetes deployment may be used for rolling update.

For a set of services that have a similar service configuration, like from the same Helm chart, but aren't fungible or won't ever see the same request stream, use different namespace names. For example, two services on different sites or the same site that serve different purposes like logistics and sales, use namespaces like `logistics-db` and `sales-db`.

Application workspaces

- Allocate application names from a global centrally managed list. Avoid reuse within your organization.
    - Namespace names are constructed from application name plus suffixes which indicate environment and location.
    - Don't use the same application or namespace name for two unrelated purposes, even across clusters. If these clusters are later combined into a single fleet, the namespace names will then collide.
- Create a new namespace when deploying an application that is new to GKE.
    - Several closely interacting applications can share a namespace, if they're developed and managed by the same team.
- Make a team or function project for each namespace.
    - Several namespaces that are developed and managed by the same people can share a project.
- For each application that needs to interact with Google Cloud resources:
    - Create a service account in the application's project, such as `project-`*`n`*`-app-`*`xyz`*.
    - Grant the workload identities in that namespace permission to access cloud resources that they need to operate, such as Cloud Storage buckets or Cloud databases.

Design and deploy applications

- When possible, refactor monolithic applications into smaller services[18].
- When building images for newly written applications, prefer distro-less images.
- Where possible, use active-active replication for applications, and use the following considerations:
    - Run at least three Pods for low traffic production applications.
    - Place these Pods behind a service.
    - When porting larger single instance applications, reduce the memory and CPU requirements proportionally.
    - Running a single Pod is acceptable for development environments where you don't need redundancy.

- Run distributed applications across clusters, with multi-cluster ingress. This approach helps distribute application load, and helps minimize the impact of a misconfiguration or failed application update in one cluster.
- Stateful applications can be deployed in containers.
    - Use pod disruption budgets (PDBs) to define your tolerance of disruptions[19].
- When deploying a new application, create a `PodMonitoring` resource[20] for each workload (statefulset, deployment) in the application's namespace. This approach sends application metrics to the operations project. Mesh service-level metrics should also go to that project.

References

1. https://docs.vmware.com/en/VMware-vSphere/6.7/com.vmware.vsphere.avail.doc/GUID-AC35EFDD-F8B7-4FAF-B946-6553D7BDBF31.html
2. https://docs.vmware.com/en/VMware-vSphere/6.7/com.vmware.vsphere.resmgmt.doc/GUID-827DBD6D-08B7-4411-9214-9E126671457F.html
3. https://opencontainers.org/
4. https://helm.sh/docs/topics/chart_repository/
5. https://cloud.google.com/kubernetes-engine/enterprise/policy-controller/docs/how-to/using-asm-security-policy
6. https://cloud.google.com/kubernetes-engine/enterprise/policy-controller/docs/latest/reference/constraint-template-library#k8spsphostfilesystem
7. https://cloud.google.com/service-mesh/docs/security/anthos-service-mesh-security-best-practices#enable-access-controls
8. https://cloud.google.com/architecture/migrations
9. https://cloud.google.com/architecture/service-meshes-in-microservices-architecture
10. https://cloud.google.com/service-mesh/docs/unified-install/off-gcp-multi-cluster-setup
11. https://cloud.google.com/service-mesh/docs/security/anthos-service-mesh-security-best-practices
12. https://cloud.google.com/anthos-config-management/docs/how-to/using-asm-security-policy#high_strictness_level
13. https://cloud.google.com/service-mesh/docs/unified-install/install-anthos-service-mesh#install_ca_service
14. https://istio.io/v1.14/docs/tasks/traffic-management/ingress/secure-ingress/
15. https://istio.io/latest/docs/tasks/traffic-management/egress/egress-gateway-tls-origination/
16. https://cloud.google.com/kubernetes-engine/distributed-cloud/bare-metal/docs/limits#dataplane_v2_ebpf_limit
17. https://cloud.google.com/logging/docs/logs-views
18. https://cloud.google.com/architecture/microservices-architecture-refactoring-monoliths
19. https://cloud.google.com/kubernetes-engine/docs/best-practices/upgrading-clusters#reduce-disruption
20. https://cloud.google.com/kubernetes-engine/distributed-cloud/vmware/docs/how-to/application-logging-monitoring