TECHNICAL WHITE PAPER:
February 2026

CI/CD Pipelines using Argo CD on vSphere Kubernetes Service

Reference Architecture

Contents

Executive Summary 3
Introduction 4
The Cloud-Native Imperative 4
Understanding CI/CD 4
Common Pain Points with Kubernetes CI/CD 4
Understanding GitOps 5
vSphere Kubernetes Service 6
Argo CD 7
Solution Architecture 9
Core Infrastructure 9
Kubernetes Environment 10
Cluster Configuration 11
Solution Validation 14
Certificates and Secured Communication 14
Harbor Configuration 14
GitLab Configuration 15
Phase 1: Initial deployment through Helm 15
Phase 2: Externalise Services 18
Phase 3: Externalising Gitaly 21
Application Repository 23
CI Pipelines 24
GitLab Runners and Executors 24
Continuous Integration Configuration 26
Continuous Delivery and Change Reconciliation with Argo CD 28
Argo CD Configuration 28
Conclusion 30
Appendix A: Example Client VM Configuration & Tooling 31
Appendix B: Example Installation of a Standard VKS Package 33

Executive Summary

This reference architecture enables organizations to accelerate application delivery through a GitOps-driven CI/CD pipeline on vSphere Kubernetes Service (VKS). By adopting Argo CD on VKS within VMware Cloud Foundation, enterprises gain consistent infrastructure management, efficient resource utilization, and streamlined Kubernetes operations.

The architecture provides DevOps teams with a clear, repeatable blueprint for implementing fully automated integration and delivery processes in Kubernetes environments. This foundation supports faster release cycles, improved developer agility, and sustained software innovation across cloud-native workloads.

Benefits of using VKS with modern applications:

Lower TCO: With VKS organizations gain the ability to reduce silos, leverage existing tools and skill sets without having to retrain staff and/or change existing processes, whilst utilizing unified lifecycle management across infrastructure components to stay up to date with the most recent patches and minimizing security risks.

Operational Simplicity: VKS is engineered for unparalleled operational simplicity, leveraging the familiarity of existing vSphere tools, skills, and workflows. This design philosophy significantly reduces the learning curve for IT teams and streamlines management processes. With VKS, organizations benefit from automated cluster provisioning, which accelerates deployment times and minimizes manual configuration errors. Furthermore, the robust capabilities of VKS extend to automated upgrades and comprehensive lifecycle management. This integrated approach ensures consistent operations, reduces overhead, and frees up valuable resources to focus on innovation rather than infrastructure maintenance.

Run and Manage Kubernetes at Scale: Effortlessly deploy and manage Kubernetes clusters at scale, leveraging a pre-packaged, Cloud Native Computing Foundation (CNCF) certified and compliant Kubernetes distribution, which integrates seamlessly with the deployed infrastructure and includes several essential core packages. VKS provides fully automated lifecycle management, streamlining operations from initial setup to ongoing maintenance and upgrades. This comprehensive approach ensures that organizations can harness the power of Kubernetes for their containerized applications with unparalleled efficiency and reliability, without the complexities typically associated with large-scale Kubernetes deployments.

Introduction

The Cloud-Native Imperative

Organizations are rapidly adopting cloud-native architectures to meet accelerating business demands. Kubernetes has emerged as the de-facto standard for container orchestration, enabling teams to build, deploy, and scale applications with unprecedented flexibility. Alongside this shift, GitOps has gained traction as a declarative, desired-state management approach that treats infrastructure and applications as versioned code, providing a single source of truth for the entire system.

Despite these advances, many enterprises struggle to translate cloud-native principles into reliable, repeatable delivery processes. The promise of agility often collides with operational complexity, leaving teams managing fragmented toolchains and inconsistent environments.

Understanding CI/CD

Continuous Integration and Continuous Delivery (CI/CD) represents a fundamental shift in how software is built and deployed. At its core, CI/CD automates the path from code commit to production deployment, enabling organizations to deliver software changes rapidly, reliably, and repeatedly.

Continuous Integration (CI) is the practice of frequently merging code changes into a shared repository, where automated builds and tests verify each integration. Rather than accumulating weeks of changes before integration, developers commit code multiple times daily. Each commit triggers an automated pipeline that compiles the code, runs unit tests, performs security scans, and produces deployable artifacts. This "fail fast" approach catches integration issues, bugs, and conflicts early when they are easiest and least expensive to fix.

Continuous Delivery (CD) extends CI by automating the deployment of validated artifacts to target environments. Once code passes CI validation, CD pipelines automatically promote it through successive environments – from development, staging and onto production, all the while applying configurations, executing deployment strategies, and verifying successful deployment. The goal is to maintain software in a perpetually deployable state, where any commit that passes automated quality gates can be released to production with minimal manual intervention.

Together, CI/CD transforms software delivery from a manual, error-prone, and time-consuming process into an automated, predictable, and repeatable workflow. Organizations practicing effective CI/CD can deploy changes in minutes rather than weeks, respond rapidly to consumer demands, and dramatically reduce the risk associated with software releases.

Common Pain Points with Kubernetes CI/CD

Snowflake Clusters and Pipeline Proliferation
Without standardized patterns, development teams create bespoke Kubernetes clusters and custom CI/CD pipelines tailored to individual applications. This proliferation leads to inconsistent configurations, duplicated effort, and significant maintenance overhead as each "snowflake" environment requires unique troubleshooting and support.

Configuration Drift Between Environments
As applications move from development through staging to production, manual interventions and ad-hoc changes introduce drift between environments. What works in testing may fail in production due to subtle

configuration mismatches, creating unpredictable deployments and extended troubleshooting cycles that erode confidence in the release process.

Manual Promotion and Fragile Scripting
Traditional code-to-production pipelines rely heavily on imperative scripts and manual approval gates to promote code across environments. These brittle, script-based workflows are error-prone, difficult to audit, and create bottlenecks that slow release velocity. When deployments fail, teams face lengthy rollback procedures and limited visibility into what changed and why. Moreover, bespoke scripting often ends up with code and processes that may be indecipherable to those outside the specialist teams and becomes a burden to manage.

Governance and Compliance Challenges
Enterprise IT organizations must balance developer autonomy with security, compliance, and governance requirements. Ad-hoc deployment processes make it difficult to enforce policies, maintain audit trails, or ensure that only approved configurations reach production environments.

Understanding GitOps

Git provides a central, version-controlled store for application code and configuration, offering an auditable history of changes via a shared repository. GitOps is an operational model for managing infrastructure and applications where Git repositories serve as the single source of truth for declarative infrastructure and application configurations. Thus, rather than executing imperative commands to deploy or modify systems, GitOps treats desired system state as code stored in these repositories. Automated processes continuously monitor the repositories and ensure running systems match the declared configuration.

The GitOps workflow follows a simple principle: **declare what you want, commit it to Git, and let automation make it so**. When a developer or operator wants to deploy an application, update a configuration, or modify infrastructure, they make changes to files in the repository. An automated agent (in this case Argo CD) detects these changes and reconciles the actual state of the target environment with the desired state defined in Git.

Key GitOps Principles:

Git as Single Source of Truth
Every configuration, deployment manifest, and infrastructure definition lives in the Git repository. This creates a complete, versioned history of all changes, enabling teams to understand what's deployed, when it changed, and who authorized the modification.

Declarative Configuration
As per the Kubernetes paradigm, systems are described declaratively: defining the desired end state rather than the steps to achieve it. A Kubernetes deployment manifest describes what containers should run (and not the sequence of kubectl commands to create them).

Automated Synchronization
GitOps tools continuously compare actual running system state against Git-declared state. When they diverge, automation either alerts operators or automatically corrects the drift. This ensures systems remain in their intended configuration without manual intervention.

Pull-Based Deployment
Unlike traditional "push" models where CI tools directly deploy to production, GitOps uses a "pull" approach. Agents running inside target environments pull changes from Git, eliminating the need to grant external CI systems production access and improving security posture.

vSphere Kubernetes Service

VMware Cloud Foundation with vSphere Kubernetes Service (VKS) provides an enterprise-grade Kubernetes runtime built directly into VMware Cloud Foundation (VCF). With CNCF certified Kubernetes, VKS enables platform engineers to deploy and manage Kubernetes clusters while leveraging a comprehensive set of cloud services in VCF. Cloud admins benefit from the support for N-2 Kubernetes versions, enterprise grade security, and simplified lifecycle management for modern apps adoption.

Key benefits include:

- **Built-in Governance and Security**: VKS as part of VMware Cloud Foundation, provides centralized policy enforcement, role-based access control, and network micro-segmentation through NSX, ensuring Kubernetes workloads meet enterprise security standards.
- **Consistent Infrastructure Management**: VKS leverages the same vSphere infrastructure that enterprises already trust, eliminating the operational complexity of managing separate Kubernetes and VM stacks.
- **Unified Networking and Storage**: VKS inherits vSphere's mature networking and storage capabilities, simplifying connectivity between cloud-native applications and existing enterprise systems.
- **Operational Familiarity**: IT teams can manage Kubernetes clusters using familiar vSphere tools and workflows, reducing the learning curve and operational risk associated with adopting container platforms.

By building on VMware Cloud Foundation, organizations establish a standardized, supportable Kubernetes foundation that bridges cloud-native development practices with enterprise operational requirements.

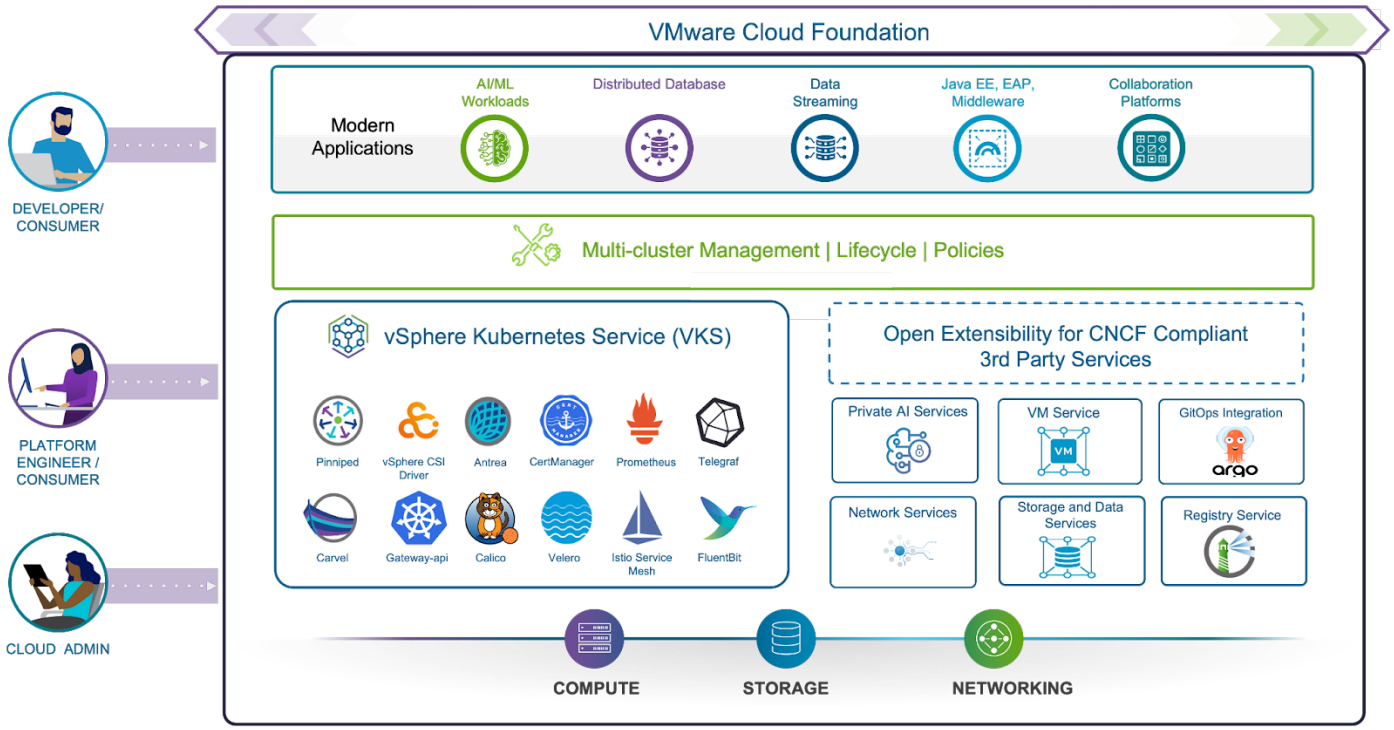

Figure 1: VMware Cloud Foundation with vSphere Kubernetes Service

Argo CD

Argo CD is a declarative, GitOps continuous delivery tool for Kubernetes. GitOps represents a fundamental shift from traditional "push-based" CI/CD models to a declarative, "pull-based" approach where the desired state of applications and infrastructure is defined in Git repositories. Argo CD, a leading GitOps continuous delivery tool for Kubernetes, operationalizes this model by continuously monitoring Git for changes and automatically synchronizing cluster state to match the declared configuration.

Key Advantages Over Classic CI-Only Setups:

Declarative Desired State Management
Rather than executing imperative deployment scripts, GitOps defines what the system *should* look like. Argo CD continuously reconciles actual cluster state with the desired state in Git, automatically correcting drift and ensuring consistency across environments without manual intervention.

Git as the Single Source of Truth
Every configuration change, application update, and infrastructure modification is versioned in Git. This provides complete auditability, enables easy rollbacks to any previous state, and eliminates the "what's actually running in production?" problem that plagues script-based deployments.

Automated Synchronization and Self-Healing
Argo CD continuously monitors both Git repositories and Kubernetes clusters. When changes are committed to Git, Argo CD automatically deploys them. If someone manually modifies cluster resources, Argo CD detects the drift and can automatically restore the declared state, preventing unauthorized changes and configuration creep.

Enhanced Developer Experience
Developers interact with familiar Git workflows—pull requests, code reviews, and merge commits—to deploy applications. This removes the need to understand complex CI/CD pipelines or kubectl commands, lowering barriers to deployment and accelerating development velocity.

Separation of Concerns
CI pipelines focus on building, testing, and packaging application artifacts, while Argo CD handles deployment. This clear separation simplifies pipeline design, improves security by limiting CI tool access to production clusters, and enables independent scaling of build and deployment infrastructure.

Multi-Environment Consistency
GitOps naturally supports multiple environments (dev, staging, production) through Git branching strategies or separate repositories. Argo CD ensures each environment remains synchronized with its declared configuration, eliminating the drift that occurs when environments are managed through manual processes or environment-specific scripts

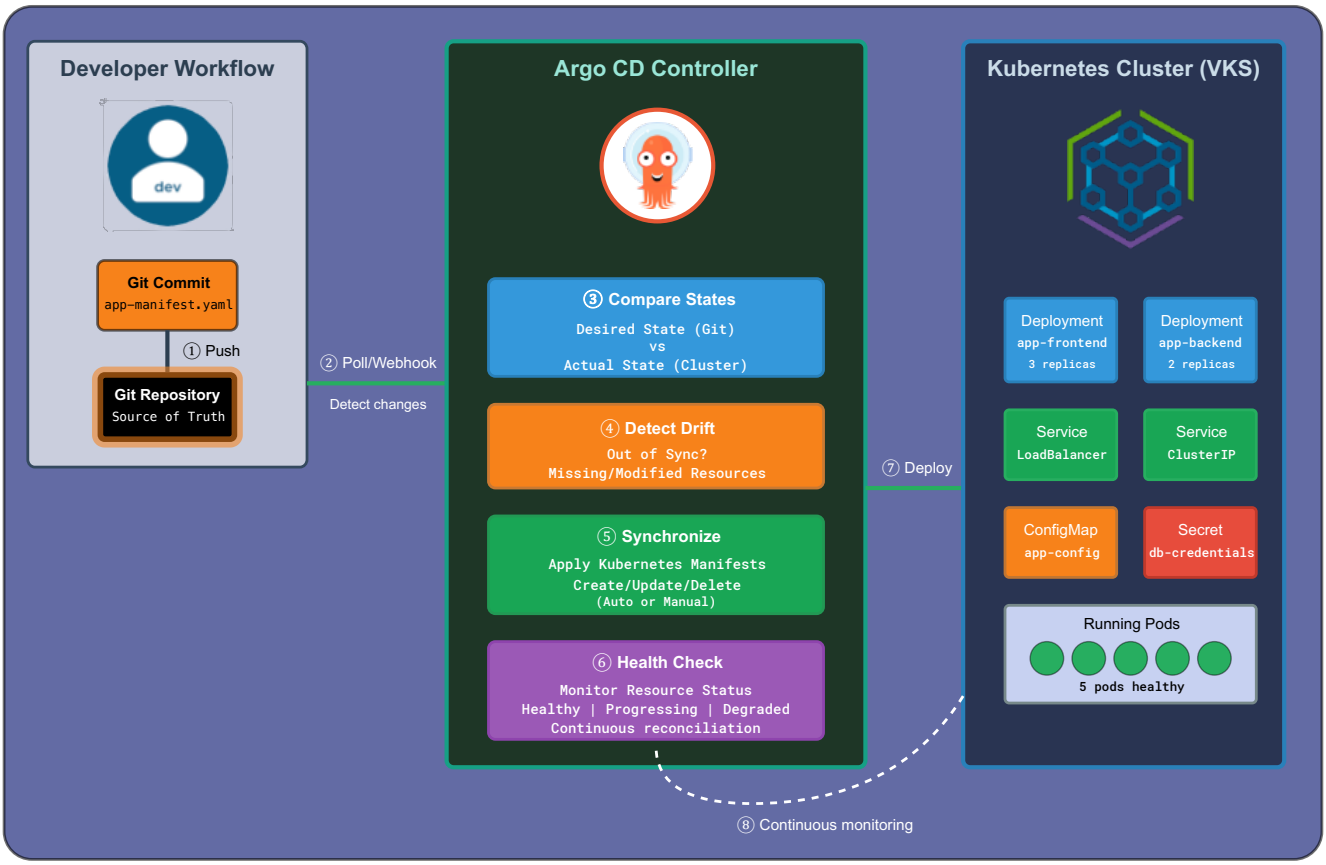

Figure 2: CI/CD Workflow with Argo CD

Solution Architecture

Core Infrastructure

This paper is based on a VCF 9.0 environment, integrating compute, networking, and storage to provide an holistic private cloud platform. For component details, visit:
https://techdocs.broadcom.com/us/en/vmware-cis/vcf/vcf-9-0-and-later/9-0/release

Figure 3: Core Infrastructure

Component	Version	Notes
VCF	9.0.0	3x Hosts — Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz 1023.62 GB Memory
Supervisor	v1.30.10	build 24845085

Table 1: Core Infrastructure Components

Kubernetes Environment

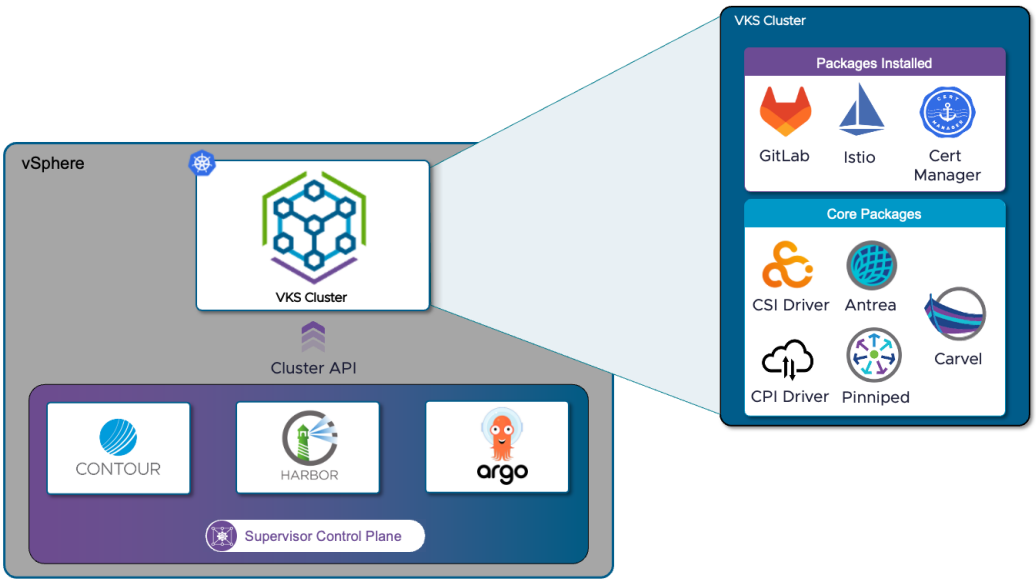

Figure 4: Supervisor Control Plane and vSphere Kubernetes Service

At its core, vSphere provides a consistent, highly available Supervisor control plane that leverages ESX hosts directly to provide essential services, including the Harbor container registry, the Contour ingress controller, and the Argo CD GitOps utility. For more information on vSphere Supervisor services and installation, visit: https://techdocs.broadcom.com/us/en/vmware-cis/vcf/vsphere-supervisor-services-and-standalone-components/latest.html

This control plane is responsible for the automated deployment and lifecycle management of CNCF-compliant Kubernetes clusters via the vSphere Kubernetes Service (VKS). These VKS clusters are fully CNCF-conformant and include core packages such as Antrea for networking and the Cloud Storage Interface driver to provide storage. Moreover, Broadcom also provides a set of standard open-source packages for VKS, distributed in Carvel format and validated against supported VKS cluster versions. For further details see the release notes: https://techdocs.broadcom.com/us/en/vmware-cis/vcf/vsphere-supervisor-services-and-standalone-components/latest/release-notes/vks-standard-packages-release-notes.html

Component	Version	Notes
vSphere Kubernetes Service	3.4.1+v1.33	Control plane Service
Harbor	2.13.1+vmware.1-vks.1	Control plane Service
Argo CD	1.0.1	Control plane Service
Contour	1.32.0+vmware.1-vks.1	Control plane Service
Local Consumption Interface	9.0.0+8594cb6b	Control plane Service

Table 2: Control Plane Components

Cluster Configuration

The target Kubernetes cluster was provisioned within vCenter using the Local Consumption Interface (see https://techdocs.broadcom.com/us/en/vmware-cis/vcf/vsphere-supervisor-services-and-standalone-components/latest/using-supervisor-services/using-the-local-consumption-interface.html) leveraging Cluster API.

The cluster comprised of three control plane nodes and three worker nodes. Storage was provided by vSAN Express Storage Architecture (ESA), using the default RAID 5 storage policy. The vSphere CSI driver exposed this policy to an adjacent Kubernetes storage class, 'vsan-esa-default-policy-raid5'. Node resources were configured using a *guaranteed-large* profile, with 100% reservation (of the allocated 4 vCPU and 16 GiB of RAM per node). In addition, each node was associated with an extra 800 Gi persistent volume.

The cluster was then managed using a client VM. Refer to Appendix A for example client VM configuration and tooling

Component	Configuration	Notes
Kubernetes Control Plane	3 Replicas — 4x vCPU 16GB RAM vSAN RAID 5	VM Class: "guaranteed large" Storage Class: "vsan-esa-default-policy-raid5"
Kubernetes Worker Nodes	3 Replicas — 4x vCPU 16GB RAM vSAN RAID 5 + Extra 800Gi Volume	VM Class: "guaranteed large" Storage Class: "vsan-esa-default-policy-raid5"
Kubernetes Release	v1.33.3+vmware.1-fips	
OS Image	Photon	
Cluster Domain	cluster.local	
CIDR Blocks	Pods: 192.168.156.0/20 Services: 10.96.0.0/12	

Table 3: Kubernetes Cluster Configuration Summary

```yaml
# vks-cluster.yaml

apiVersion: cluster.x-k8s.io/v1beta1
kind: Cluster
metadata:
  name: kubernetes-cluster-kmnr
  namespace: supervisor-namespace
spec:
  clusterNetwork:
    pods:
      cidrBlocks:
        - 192.168.156.0/20
    services:
      cidrBlocks:
        - 10.96.0.0/12
    serviceDomain: cluster.local
  topology:
    class: builtin-generic-v3.4.0
    version: v1.33.3---vmware.1-fips-vkr.1
    variables:
      - name: kubernetes
        value:
          certificateRotation:
            enabled: true
            renewalDaysBeforeExpiry: 90
      - name: vmClass
        value: guaranteed-large
      - name: storageClass
        value: vsan-esa-default-policy-raid5
    controlPlane:
      replicas: 3
      metadata:
        annotations:
          run.tanzu.vmware.com/resolve-os-image: os-name=photon
    workers:
      machineDeployments:
        - class: node-pool
          name: kubernetes-cluster-kmnr-nodepool-nve9
          replicas: 3
          metadata:
            annotations:
              run.tanzu.vmware.com/resolve-os-image: os-name=photon
          variables:
            overrides:
              - name: volumes
                value:
                  - name: vol-5be3
                    mountPath: /var/lib/containerd
                    storageClass: vsan-esa-default-policy-raid5
                    capacity: 800Gi
```

Kubernetes Packages

As noted earlier, VKS clusters include a default set of core packages, such as Antrea for networking and Pinniped for authentication. For this deployment, the Istio and cert-manager standard packages were also installed, while the GitLab components (Operator and Runner) was installed separately using Helm (for details see: https://docs.gitlab.com/operator/installation/?tab=Helm+Chart)

To install standard packages on a VKS cluster, refer to the documentation: https://techdocs.broadcom.com/us/en/vmware-cis/vcf/vsphere-supervisor-services-and-standalone-components/latest/managing-vsphere-kuberenetes-service-clusters-and-workloads/installing-standard-packages-on-tkg-service-clusters.html.

As an example, Appendix B details the installation of the Istio package.

Package	Version	Notes
Antrea	2.3.1+vmware.1-tkg.1	VKS core package
Gateway API	1.2.1+vmware.2-tkg.1	VKS core package
Guest-cluster auth-service	1.4.2+vmware.1-tkg.1	VKS core package
Metrics Server	0.7.2+vmware.7-fips-tkg.1	VKS core package
Pinniped	0.39.0+vmware.2-tkg.1	VKS core package
Secretgen Controller	0.19.1+vmware.2-fips-tkg.1	VKS core package
vSphere CPI	1.33.0+vmware.1-tkg.1	VKS core package
vSphere CSI	3.5.0+vmware.1-tkg.1	VKS core package
Istio	1.25.3+vmware.1-vks.2	VKS standard package
Cert-Manager	1.18.2+vmware.1-vks.1	VKS standard package
GitLab Operator	2.4.1	Helm package
GitLab Runner	18.4.0	Helm package

Table 4: Kubernetes Cluster Packages

Solution Validation

Certificates and Secured Communication

Secure TLS connections rely on signed certificates and reliable name resolution. Each infrastructure component must be able to reach a DNS server, and DNS records must be created for each exposed service (for example, Harbor and Argo CD). A certificate authority (CA) then issues certificates that cryptographically attest to the identity of these services.

To achieve this, first a Certificate Signing Request (CSR) must be generated for the CA to issue certificates against. Typically OpenSSL is used:

```
# Create a CSR using OpenSSL

FQDN="harbor.content.tmm.broadcom.lab"
openssl req -new -newkey rsa:2048 -nodes \
  -keyout harbor.key \
  -out harbor.csr \
  -subj "/CN=${FQDN}" \
  -addext "subjectAltName=DNS:${FQDN}"
```

Harbor Configuration

Here, we use Harbor as a Supervisor service, which allows for a convenient, central place for images. Moreover, VKS clusters created under the same control plane are automatically configured to trust the Supervisor-deployed Harbor registry.

To use custom certificates, we add the TLS details into the Harbor configuration file. For sample configuration and detailed configuration information see: https://techdocs.broadcom.com/us/en/vmware-cis/vcf/vsphere-supervisor-services-and-standalone-components/latest/using-supervisor-services/installing-and-configuring-harbor-and-contour/install-harbor-with-a-customized-certificate.html

```
# In the Supervisor Context, get the Harbor namespace
# For example, 'svc-harbor-domain-c25'

HARBOR_NS=<harbor namespace>


# Check Harbor tls certificate

kubectl -n "$HARBOR_NS" get secret harbor-tls \
  -o jsonpath='{.data.tls\.crt}' | base64 -d
```

By default, Harbor is deployed with a 1Gi database (this can be configured at install time in the data values manifest). Here we increase this size post-deployment by patching the persistent volume (the filesystem will automatically be extended):

```
# Add more storage to Harbor (2TiB)
kubectl -n "$HARBOR_NS" patch pvc harbor-registry \
  -p '{"spec":{"resources":{"requests":{"storage":"2Ti"}}}}'
```

For production environments where high-availability and scale are important, a separate Postgres database is recommended. This can be configured in the data values file, for example:

```
# Configure external PostgreSQL database
# Excerpt from Harbor values
database:
  type: external
  external:
    host: 10.163.44.42
    port: "5432"
    username: postgres
    password: "Harbor Database User Password"
    coreDatabase: "harbor-database"
    sslmode: disable
```

For further production ready recommendations for Harbor, visit https://blogs.vmware.com/cloud-foundation/2025/12/02/making-harbor-production-ready-essential-considerations-for-deployment/

GitLab Configuration

Phase 1: Initial deployment through Helm

In the initial phase, GitLab is deployed into a VKS workload cluster by using the official GitLab Helm chart. At this stage, all stateful services (PostgreSQL, Redis, Gitaly, Praefect, and MinIO) run inside the Kubernetes cluster using the chart defaults. Note that this initial configuration is explicitly not intended for production use. Here, we will use this deployment as a functional baseline that we will progressively harden in phases in line with GitLab's Cloud Native Hybrid reference architectures. For further information on installation and configuration of GitLab, visit https://docs.gitlab.com/

By default, the GitLab Helm chart references images hosted on public registries. In most enterprise environments, direct access to public registries such as Docker Hub is restricted, rate-limited, or prohibited entirely. To align with corporate security and compliance policies, all GitLab images should be mirrored into an internal container registry (such as the Harbor registry deployed in this deployment). The Helm configuration for GitLab can be set so that all components pull from this trusted source.

The baseline deployment integrates with the platform's existing PKI and container registry services from day one:

- **Internal container registry** – All GitLab images are sourced from the Supervisor-hosted Harbor registry. Images are mirrored into Harbor, and the chart's `global.image.registry` and `global.image.pullSecrets` values are set so that every GitLab component pulls from this internal, trusted registry rather than from public registries.

- **Custom certificate authority trust** – The Harbor CA certificate is provided to GitLab via a Kubernetes secret and referenced through `global.certificates.customCAs`. This causes the

GitLab pods to import the internal CA into their trust stores, allowing GitLab services to connect securely to Harbor and other internal TLS endpoints that use the same CA.

- **Ingress TLS** – Certificates issued by the internal CA are imported into the GitLab namespace as a Kubernetes TLS secrets. The Helm chart is configured to use these secret via `global.ingress.tls.secretName`, so that all GitLab HTTP(S) endpoints are exposed over TLS through the platform ingress controller.

With these settings, the Phase 1 deployment delivers a fully functional GitLab instance running on VKS, fronted by the platform ingress layer, and integrated with the standard internal registry and PKI. Later phases focus on externalising stateful services (PostgreSQL, Redis, Gitaly, and object storage) to align with GitLab's supported hybrid architecture while preserving this integration.

```
# Create gitlab namespace
kubectl create namespace gitlab-system

# Create secret with registry details
kubectl -n gitlab-system create secret docker-registry gitlab-regcred \
  --docker-server=harbor.supervisor.lab \
  --docker-username=robot$gitlab \
  --docker-password='<redacted>' \
  --docker-email=gitlab@lab

# Create generic secret with registry certificate
kubectl -n gitlab-system create secret generic harbor-ca \
  --from-file=harbor-ca.crt=/local/path/to/cert/harbor-ca.crt
```

GitLab has four separate endpoints that will be fronted by an ingress. Each endpoint will need a separate certificate generated (for a lab environment, it is acceptable to have a generic wildcard certificate):

- gitlab-webservice-default
- gitlab-kas
- gitlab-registry
- gitlab-minio

```
# Create tls secrets for gitlab endpoints
kubectl -n gitlab-system create secret tls gitlab-tls \
  --cert=gitlab.crt \
  --key=gitlab.key

kubectl -n gitlab-system create secret tls kas-tls \
  --cert=gitlab.crt \
  --key=kas.key

kubectl -n gitlab-system create secret tls registry-tls \
  --cert=gitlab.crt \
  --key=registry.key

kubectl -n gitlab-system create secret tls minio-tls \
  --cert=gitlab.crt \
  --key=minio.key
```

The secrets can then be referenced in a manifest, which we will later use with Helm:

```
# gitlab-baseline.yaml
#
# Baseline GitLab deployment on VKS: Phase 1
# - One host per service (gitlab/registry/minio/kas)
# - One TLS secret per host
# - Images pulled from internal Harbor
# - Harbor CA injected into GitLab pods
#
# Note: PostgreSQL, Redis, Gitaly, Praefect & MinIO are
# in-cluster. Later phases externalise these services

global:
  hosts:
    domain: content.tmm.broadcom.lab
    https: true

  ingress:
    configureCertmanager: false
    tls:
      enabled: true

  image:
    registry: harbor.supervisor.lab
    pullPolicy: IfNotPresent
    pullSecrets:
      - name: gitlab-regcred

  certificates:
    customCAs:
      - secret: harbor-ca

gitlab:
  webservice:
    ingress:
      tls:
        # cert for gitlab.content.tmm.broadcom.lab
        secretName: gitlab-tls
  kas:
    ingress:
      tls:
        # cert for kas.content.tmm.broadcom.lab
        secretName: kas-tls

registry:
  ingress:
    tls:
      # cert for registry.content.tmm.broadcom.lab
      secretName: registry-tls

minio:
  ingress:
    tls:
      # cert for minio.content.tmm.broadcom.lab
      secretName: minio-tls
```

The GitLab operator can then be installed using Helm with the above manifest:

```
# Install gitlab-operator:
helm repo add gitlab https://charts.gitlab.io
helm repo update
helm install gitlab-operator gitlab/gitlab-operator \
  --namespace gitlab-system \
  --values gitlab-baseline.yaml
```

At this point, the GitLab interface should be available via a browser and the TLS certificates should be valid.

Phase 2: Externalise Services

In Phase 2, the deployment is moved closer to GitLab's Cloud Native Hybrid reference architecture by externalising the core stateful services: PostgreSQL, Redis, and object storage. The GitLab application tier continues to run in the VKS cluster, but now connects to production-grade services that are deployed on dedicated infrastructure.

Therefore, for the purposes of this paper, we assume that the following production services are available to use:

PostgreSQL: see https://docs.gitlab.com/administration/reference_architectures/50k_users/#provide-your-own-postgresql-instance

Redis: see https://docs.gitlab.com/administration/reference_architectures/50k_users/#provide-your-own-redis-instances

MinIO: see https://docs.gitlab.com/administration/reference_architectures/50k_users/#configure-the-object-storage. Note that MinIO can be deployed easily on vSAN. For more information vist: https://www.min.io/solutions/vmware

GitLab recommends using separate buckets for each data type. Below, we assume MinIO is configured with at least the four core buckets:

- gitlab-artifacts-storage
- gitlab-lfs-storage
- gitlab-uploads-storage
- gitlab-packages-storage

The GitLab Helm chart's embedded PostgreSQL (Bitnami), Redis, and MinIO sub-charts are disabled, and the chart is configured to use the external endpoints instead. Connection details are provided (via the global.psql, global.redis, and global.appConfig.object_store settings) with passwords and access keys injected from Kubernetes secrets.

After this phase, all database, cache, and object storage data resides outside the Kubernetes cluster. The in-cluster components are treated as stateless workloads that can be scaled or rescheduled without impacting data durability. Gitaly remains in-cluster at this stage and is externalised in Phase 3 to complete the Cloud Native Hybrid pattern.

For access the external services, we create generic secrets with the login details:

```
# External Postgres password for the gitlab user
kubectl -n gitlab-system create secret generic gitlab-postgresql-password \
  --from-literal=postgres-password='REDACTED-GITLAB-DB-PASSWORD'

# External Redis password (single DB or cluster)
kubectl -n gitlab-system create secret generic gitlab-redis-secret \
  --from-literal=redis-password='REDACTED-REDIS-PASSWORD'

# Access keys for external MinIO / S3-compatible object storage
kubectl -n gitlab-system create secret generic gitlab-object-storage \
  --from-literal=accesskey='REDACTED-ACCESS-KEY' \
  --from-literal=secretkey='REDACTED-SECRET-KEY'
```

Next, we define a manifest to update GitLab with the external services:

```
# gitlab-external-core.yaml
#
# Phase 2 – External PostgreSQL, Redis and object storage
#

# Disable the embedded PostgreSQL and Redis charts
postgresql:
  install: false

redis:
  install: false

global:

  # External PostgreSQL
  psql:
    host: gitlab-db.internal.lab
    port: 5432
    database: gitlabhq_production
    username: gitlab
    password:
      secret: gitlab-postgresql-password
      key: postgres-password

  # External Redis
  redis:
    host: redis.gitlab-cache.internal.lab
    port: 6379
    auth:
      enabled: true
      secret: gitlab-redis-secret
      key: redis-password

  # External object storage (MinIO / S3-compatible)
  appConfig:
    object_store:
      enabled: true
      proxy_download: false
      connection:
```

```
      secret: gitlab-object-storage
      key: connection
    objects:
      artifacts:
        bucket: gitlab-artifacts-storage
      lfs:
        bucket: gitlab-lfs-storage
      uploads:
        bucket: gitlab-uploads-storage
      packages:
        bucket: gitlab-packages-storage
```

These values are then added to the installation. Note that the baseline values (from the original install) is required (Helm will merge the requests).

```
# Update Helm installation with Phase 2 values
helm upgrade gitlab gitlab/gitlab \
  --namespace gitlab-system \
  --values gitlab-baseline.yaml \
  --values gitlab-external-core.yaml
```

Phase 3: Externalising Gitaly

In Phase 3, the final stateful GitLab component, Gitaly (the Git storage service), is moved out of the Kubernetes cluster to dedicated virtual machines. This aligns the deployment with GitLab's production guidance, which explicitly does not support running Gitaly inside Kubernetes and recommends an external highly-available Praefect cluster. By decoupling Gitaly from the Kubernetes cluster, we prevent storage I/O contention from impacting the CI/CD runners and web services, resulting in a more stable and responsive pipeline environment. For more details, visit:
https://docs.gitlab.com/administration/gitaly/praefect and
https://docs.gitlab.com/administration/reference_architectures/50k_users/#configure-gitaly-cluster-praefect

Here we leverage the inherent availability and scalability of VMware Cloud Foundation to run Gitaly Cluster (Praefect) on dedicated VMs backed by vSAN, while keeping the GitLab application tier in Kubernetes. In particular, this design addresses the following challenges:

1. Enhanced availability and fault tolerance

Deploying Praefect nodes as vSphere VMs allows the design to inherit the maturity of vSphere High Availability (HA) and Distributed Resource Scheduler (DRS). Instead of relying on Kubernetes pod eviction and rescheduling for a critical routing service, vSphere HA provides rapid, infrastructure-level recovery in the event of host failure. Anti-affinity rules ensure that Praefect VMs are placed on different ESXi hosts, so a single host failure cannot remove all Praefect instances. Combined with Praefect's own awareness of Gitaly node health, this delivers a resilient front end to the Gitaly cluster.

2. Scalable, high-performance repository storage with vSAN

Gitaly performance is tightly coupled to disk I/O, as every push, fetch, and merge translates into a large number of small file operations. Hosting Gitaly VMs on vSAN gives the platform team fine-grained control over how that storage behaves:

Data locality and performance: vSAN is integrated into the ESXi hypervisor, providing low-latency access paths between Gitaly VMs and their backing storage. This is well suited to the random I/O patterns of Git repositories.

Elastic scalability: capacity and performance can be scaled out non-disruptively by adding disks or hosts to the VCF cluster. Gitaly simply sees additional I/O headroom; there is no need to re-shard repositories or redesign the storage layout.

Storage Policy Based Management (SPBM): administrators can assign dedicated vSAN storage policies (for example specific RAID/FTT settings) to the Gitaly VMs, ensuring that repository data is protected and replicated according to defined availability or compliance SLAs, independently of the policies used for the Kubernetes worker nodes.

The Helm chart's in-cluster Gitaly sub-chart is disabled by setting `global.gitaly.enabled=false`. The chart is then configured to use one or more external Gitaly endpoints via `global.gitaly.external`, with a shared authentication token provided through `global.gitaly.authToken`.

```
# Add the token created as part of the Gitaly installation
kubectl -n gitlab-system create secret generic gitlab-gitaly-token \
  --from-literal=token='REDACTED-TOKEN'
```

TLS encryption is enabled on the Praefect endpoints, and the GitLab Rails nodes are configured to connect over TLS and to trust the internal CA that signs the Gitaly certificates. For more information on Gitaly TLS support, visit : https://docs.gitlab.com/administration/gitaly/tls_support/ and https://docs.gitlab.com/administration/gitaly/praefect/configure/#enable-tls-support

Prior to disabling the in-cluster Gitaly chart, all repositories are migrated to the external Gitaly nodes by using GitLab's supported mechanisms (either the repository storage moves API or a backup/restore cycle, depending on scale and downtime requirements).

We then define our manifest for external Gitaly:

```
# gitlab-external-gitaly.yaml
#
# Phase 3 — External Gitaly on VMs
#


global:
  gitaly:

    # Disable the embedded Gitaly chart
    enabled: false

    # Shared auth token used by all Gitaly clients and servers
    authToken:
      secret: gitlab-gitaly-token
      key: token

    # External Gitaly endpoints
    external:
      - name: default
        hostname: gitaly-1.gitlab-git.internal.lab
        port: 8075
        tlsEnabled: true
```

Again, the installation is updated with Helm, defining all three values files:

```
# Update Helm installation with Phase 3 values
helm upgrade gitlab gitlab/gitlab \
  --namespace gitlab-system \
  --values gitlab-baseline.yaml \
  --values gitlab-external-core.yaml
  --values gitlab-external-gitaly.yaml
```

Application Repository

Now that Git has been setup, we can populate the repository with application data. Here, we make use of the (albeit slightly modified) Google microservices demo, available publicly at: https://github.com/GoogleCloudPlatform/microservices-demo

This consists of several containers that interact together to form a shopping cart webpage. Our CI/CD process needs to build the app components from code and create a Helm chart for installation.

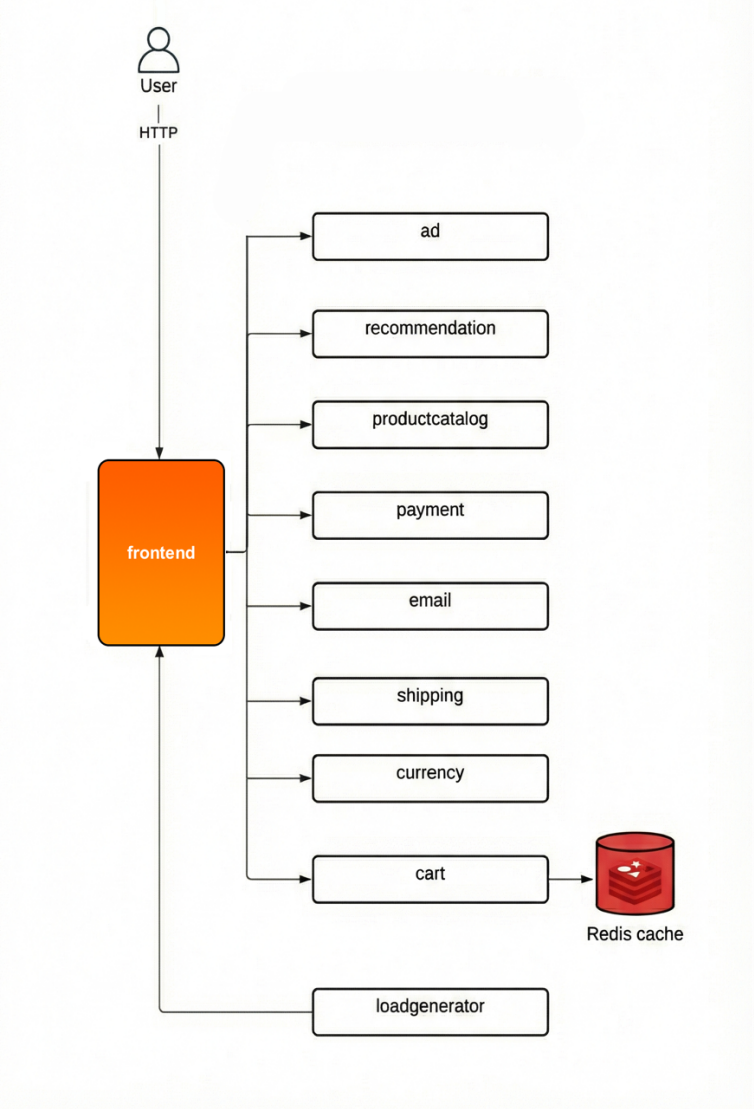

Figure 4: Application Diagram

CI Pipelines

In this model, we deliberately limit the scope of the integration pipeline to only compile the application code and publish the container image to the registry. Thus, no deployment logic is embedded in the CI system: instead, deployments are triggered implicitly when configuration changes are merged into Git. The Continuous Deployment (see next section) using Argo CD will reconcile those changes to the VKS cluster.

GitLab Runners and Executors

In this architecture, GitLab Runners are deployed as pods within the Kubernetes cluster. These runners dynamically spawn ephemeral executors for each pipeline job to perform builds.

Similar to the GitLab operator, GitLab Runners can be installed using an official Helm chart. For more information visit: https://docs.gitlab.com/runner/install/kubernetes/

First, a new runner must be registered in the GitLab UI and a registration token generated. For details visit: https://docs.gitlab.com/ci/runners/runners_scope/#create-an-instance-runner-with-a-runner-authentication-token

Similar to the GitLab operator install, we create a namespace for the runners and secret for the certificates:

```
# Create gitlab namespace
kubectl create namespace gitlab-runners

# Create generic secret for certificates
kubectl -n gitlab-runners create secret generic ad-ca \
  --from-file=gitlab.crt=/path/to/gitlab-server-cert.crt \
  --from-file=registry.crt=/path/to/registry-server-cert.crt
```

The Helm values need to reference the secret created for the runner and also inject this certificates into the executor pods. Here we mount the secret into the executor pods (via the TOML configuration) and run a script to update the certificates. For more information see https://docs.gitlab.com/runner/configuration/tls-self-signed/

We define our values manifest:

```
# gitlab-runner-values.yaml

gitlabUrl: https://gitlab.content.tmm.broadcom.lab
runnerRegistrationToken: ""  # empty; retained for compatibility
runnerToken: "my-runner-token"

# Used by the runner *deployment* so it can talk to GitLab/Registry over TLS
certsSecretName: ad-ca

image:
  registry: harbor-test.content.tmm.broadcom.lab
  image: library/gitlab-runner
  tag: ubuntu-4780c7e1  # latest available
  imagePullPolicy: IfNotPresent

concurrent: 10
checkInterval: 30
```

```
runners:
  # Define TOML fragment
  config: |
    [[runners]]
      name = "k8s-ubuntu-buildkit"
      executor = "kubernetes"
      request_concurrency = 4
      [runners.kubernetes]
        namespace = "gitlab-runners"
        image = "buildkit:latest-ubuntu"
        helper_image = "gitlab-runner-helper:x86_64-latest"
        poll_timeout = 600
        privileged   = true
      [[runners.kubernetes.volumes.secret]]
        name       = "ad-ca"
        mount_path = "/usr/local/share/ca-certificates"
        read_only  = true

  env:
    SSL_CERT_DIR: "/usr/local/share/ca-certificates"
    FF_USE_ADAPTIVE_REQUEST_CONCURRENCY: "true"

securityContext:
  runAsNonRoot: true
  readOnlyRootFilesystem: false
  allowPrivilegeEscalation: false
  capabilities:
    drop: ["ALL"]

podSecurityContext:
  runAsUser: 999
  fsGroup: 999

rbac:
  create: true
  podSecurityPolicy:
    enabled: false

serviceAccount:
  create: true
  name: gitlab-runner
```

The GitLab runner can then be installed using Helm with the above manifest:

```
# Install gitlab-runner:
helm install gitlab-runner gitlab/gitlab-runner \
  --namespace gitlab-runners \
  --values gitlab-runner-values.yaml
```

Continuous Integration Configuration

There are two stages: first build the application images from Docker files and then update the Helm chart. Here we use BuildKit (see https://docs.gitlab.com/ci/docker/using_buildkit/) to create the application images:

```
# .gitlab-ci.yaml

stages:
  - build
  - update

variables:
  IMAGE_TAG: "${CI_COMMIT_SHA}"
  REPO_NAME: "app"
  REGISTRY: "harbor.content.tmm.broadcom.lab"
  BUILDPLATFORM: "linux/amd64"
  BUILDKIT_PROGRESS: plain

# Build and push microservice images with BuildKit (daemonless)
build_image:
  stage: build

  before_script:
    - update-ca-certificates
  tags: ["k8s"]
  script:

    # Docker auth for Harbor (BuildKit uses the standard Docker config)
    - mkdir -p "$HOME/.docker"
    - |
      cat > "$HOME/.docker/config.json" <<EOF
      {
        "auths": {
          "${REGISTRY}": {
            "username": "${CI_REGISTRY_USER}",
            "password": "${CI_REGISTRY_PASSWORD}"
          }
        }
      }
      EOF

    # Loop microservices and build with BuildKit
    - |
      for dir in microservices-demo/src/*/; do
        service=$(basename "$dir")
        dest="${REGISTRY}/${REPO_NAME}/${service}:${CI_COMMIT_SHA}"
        cache_repo="${REGISTRY}/${REPO_NAME}/${service}-cache"
        echo "Building ${service} -> ${dest}"
        # BuildKit daemonless build:
        /usr/bin/buildctl-daemonless.sh build \
          --frontend dockerfile.v0 \
          --local context="${build_context}" \
          --local dockerfile="${build_context}" \
          --opt filename=Dockerfile \
          --opt platform="${BUILDPLATFORM}" \
          --opt build-arg:BUILDPLATFORM="${BUILDPLATFORM}" \
          --import-cache type=registry,ref="${cache_repo}" \
```

```
          --export-cache type=registry,ref="${cache_repo}",mode=max \
          --output type=image,name="${dest}",push=true
      done

# Update Helm chart with the new image tag and push back to Git
update_helm_chart:
  stage: update
  tags: ["k8s"]
  before_script:
    - update-ca-certificates
    - apt update
    - apt install -y git curl bash
    - git config --global user.email "gitlab_admin_977e50@example.com"
    - git config --global user.name "Administrator"
  script:
    - echo "Cloning Helm repo..."
    - git clone
"https://Administrator:${HELM_REPO_PAT}@gitlab.content.tmm.broadcom.lab/root/app-
code.git"
    - cd app-code
    - git checkout main
    - cd helm-charts
    - yq eval ".images.tag = \"${CI_COMMIT_SHA}\"" -i values.yaml
    - git add values.yaml
    - git commit -m "Update image tag to ${CI_COMMIT_SHA} for ${CI_PROJECT_NAME}"
  - git push origin main
```

Continuous Delivery and Change Reconciliation with Argo CD

In this architecture, Continuous Delivery is implemented using Argo CD and a GitOps operating model. As we saw in the previous section, the Git-based CI pipeline is responsible for building and then publishing container images to the image registry (as well as updating the declarative configuration in the repository).

Argo CD continuously monitors the destination Kubernetes cluster (here our VKS cluster) and compares the declared configuration with the live state in the target clusters. Then, when a change is merged into Git, such as: an updated image tag, an altered replica count, or a new Kubernetes resource, Argo CD detects the drift and *reconciles* the destination Kubernetes cluster to match the *desired* state.

This reconciliation loop delivers several key benefits:

- **Deterministic deployments**: the configuration applied to a cluster *always* corresponds to a specific Git commit, enabling precise audit and rollback.
- **Automated drift correction**: unauthorised or manual changes made directly in the cluster are detected and can be automatically reverted to the Git-defined state.
- **Environment promotion via Git**: promotion between environments is driven through Git operations (branching, pull requests, tag changes), rather than ad-hoc imperative scripts

Therefore, in contrast with the Git-ops based continuous integration model (CI) whereby artefacts are published to a repository, Argo CD here is responsible for both continuously updating these changes into the target cluster and ensuring the target state is as defined.

Argo CD Configuration

The Argo CD CLI is typically used to configure Argo CD. A customised binary is available specifically for the Supervisor service and can be downloaded from the Broadcom support portal. Here we install the binary on our Linux VM and add our Kubernetes (VKS) cluster as an endpoint.

```
# Download the Argo CD CLI from the Broadcom support portal
# support.broadcom.com > login > my downloads

# Login to argo
argocd login <argo IP>


# Add Kubernets cluster
argocd cluster add kubernetes-cluster-kmnr


# Add Git repository
argocd repo add https://<GITLAB_URL>/<PROJECT_PATH>/app.git \
  --username <GITLAB_USERNAME> \
  --password <GITLAB_PERSONAL_ACCESS_TOKEN>
```

Argo CD adds several CRDs (a full list can be found at https://techdocs.broadcom.com/us/en/vmware-cis/vcf/vsphere-supervisor-services-and-standalone-components/latest/using-supervisor-services/using-argo-cd-service/argocd-custom-resrouce-reference.html).

We use the *argocd-application-controller* CRD to define where the app is placed (i.e. our VKS cluster) and the source repository:

```
# argo-my-app.yaml

apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: designlib-app
  namespace: argo-cd
spec:
  destination:
    server: https://10.163.44.39:6443
    namespace: my-app
  project: default
  source:
    repoURL: https://gitlab.content.tmm.broadcom.lab/root/app-code.git
    path: helm-charts
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

Thus, Argo CD will compare the running state (on the VKS cluster) to the target state (the Git repository) and reconciliate as needed. Any changes to the Helm chart is reflected with an updated installation on the destination cluster.

Note that Argo CD can also be used to manage VKS clusters and vSphere resources. For more details visit https://techdocs.broadcom.com/us/en/vmware-cis/vcf/vsphere-supervisor-services-and-standalone-components/latest/using-supervisor-services/using-argo-cd-service/manage-resources-in-vsphere-namespaces-with-argo-cd.html

Conclusion

With Harbor, Argo CD, Gitlab and the destination VKS cluster configured, we now have a full end-to-end CI/CD pipeline powered by GitOps.

Updating the application code on the Git Repository triggers the Gitlab runner to build new containers and package these in a Helm chart. Argo compares the Helm chart in the repository to the running app on the VKS cluster and triggers a re-install as needed.

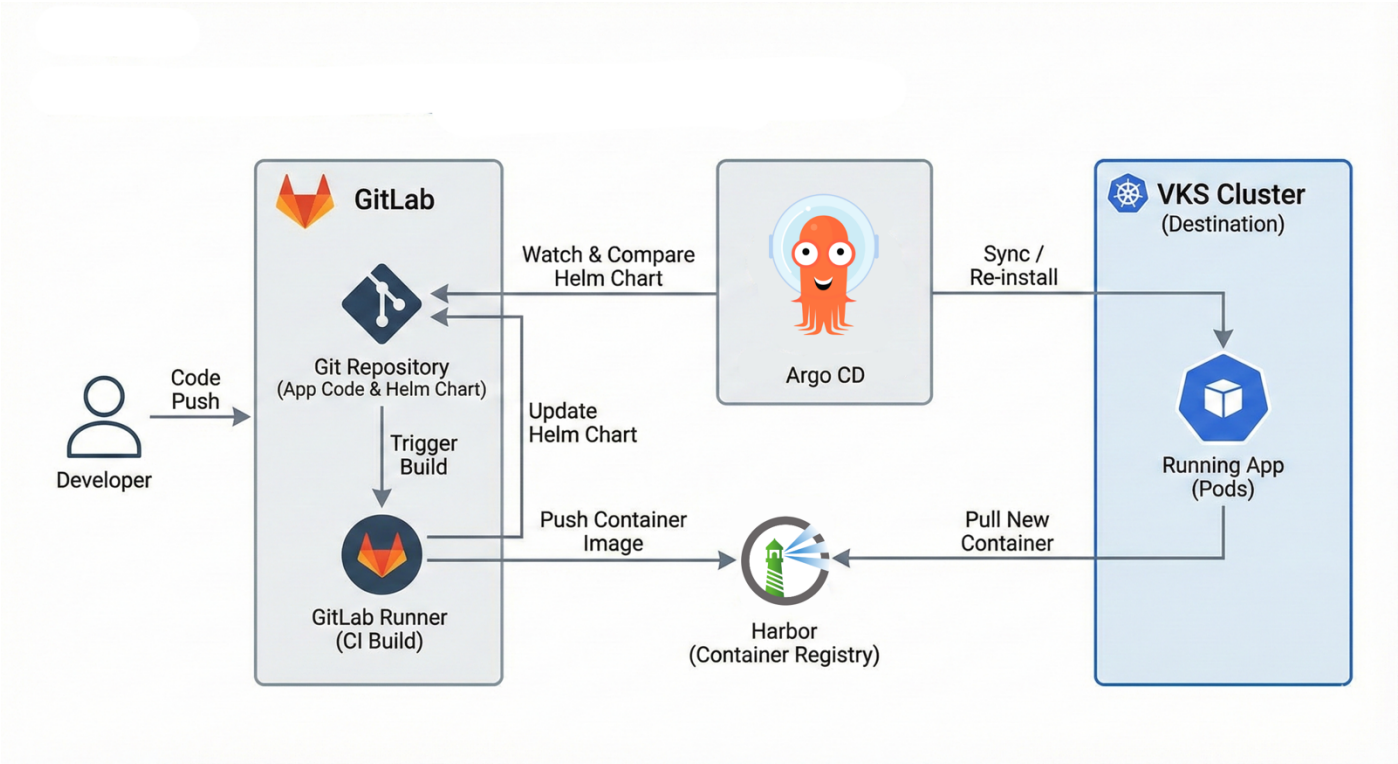

Figure 4: End-to-end CI/CD Pipeline

In this paper, we have demonstrated the ease of using vSphere Kubernetes Service (VKS) in conjunction with GitOps pipelines and Argo CD. We have created a resilient environment where infrastructure and application code are managed as a single, declarative source of truth. This end-to-end workflow transforms the complex task of microservice management into a streamlined, predictable process, providing a robust foundation for modern cloud-native applications.

Appendix A:
Example Client VM Configuration & Tooling

Here, we used an Ubuntu Jammy cloud VM. At the time of writing, the OVA image can be obtained from: https://cloud-images.ubuntu.com/jammy/current/jammy-server-cloudimg-amd64.ova

The VCF command-line can be downloaded and installed thus:

```
# Download VCF CLI & install (version 9.0)
curl -fsSL https://packages.broadcom.com/artifactory/vcf-distro\
  /vcf-cli/linux/amd64/v9.0.0/vcf-cli.tar.gz | tar xz

sudo cp vcf-cli-linux_amd64 /usr/local/bin/vcf

# (Optional) Add autocomplete for VCF command line (bash shell)
echo "source <(vcf completion bash)" >> ~/.bashrc

# Create context & login
vcf context create --endpoint=<supervisor endpoint> \
  --username administrator@vsphere.local

# Use context
vcf context use supervisor-namespace
```

To trust our connection to vCenter, we install the certificate:

```
# Get vCenter certs & install
# Download the zip file to /tmp using curl (insecure mode required)
VCENTER_IP=<vCenter IP>
curl -k -fsSL -o /tmp/vccert.zip https://${VCENTER_IP}/certs/download.zip

# Unzip and copy to SSL directory
unzip /tmp/vccert.zip -d /tmp
sudo cp /tmp/certs/lin/* /etc/ssl/certs

# Update system certs
sudo update-ca-certificates
```

Optionally install Kubectl and vSphere plugin:

```
# (Optional) Get Kubectl & vSphere plugin
# Download the zip file to /tmp using curl
SUPERVISOR_ENDPOINT="<supervisor endpoint IP>"
curl -fsSL -o /tmp/vsphere-plugin.zip \
  https://${SUPERVISOR_ENDPOINT}/wcp/plugin/linux-amd64/vsphere-plugin.zip
```

```
# Unzip and copy to local bin directory
unzip /tmp/vsphere-plugin.zip -d /tmp
sudo cp /tmp/bin/* /usr/local/bin

# (Optional) Add autocomplete and shorthand 'k' for Kubectl (bash shell)
cat << 'EOF' >> ~/.bashrc
if command -v kubectl &> /dev/null; then
    source <(kubectl completion bash)
    alias k=kubectl
    complete -o default -F __start_kubectl k
fi
EOF
source ~/.bashrc
```

Finally, we install Helm by following the instructions:
https://helm.sh/docs/intro/install/

```
# Get Helm using download script
curl -fsSL -o /tmp/get_helm.sh \
  https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-4

chmod +x /tmp/get_helm.sh

/tmp/get_helm.sh
```

Appendix B: Example Installation of a Standard VKS Package

Refer to the standard package release notes, https://techdocs.broadcom.com/us/en/vmware-cis/vcf/vsphere-supervisor-services-and-standalone-components/latest/release-notes/vks-standard-packages-release-notes.html

```
# Change context to the VKS cluster
vcf context use vkscluster

# Create a namespace for the packages
kubectl create ns packages

# Get the package repo url, for example:
RELEASE_NOTES="https://techdocs.broadcom.com/us/en/vmware-cis/vcf/vsphere-supervisor-services-and-standalone-components/latest/release-notes/vks-standard-packages-release-notes.html"

URL=$(curl -sq "$RELEASE_NOTES" \
  | grep -o 'projects\.packages\.broadcom\.com[^<]*' | head -1)

# Add the standard package repo, for example:
vcf package repository add vks-repo --url $URL -n packages

# Get the available Istio version from the repo
vcf package available get istio.kubernetes.vmware.com -n packages

# Obtain the default values file
vcf package available get istio.kubernetes.vmware.com/1.25.3+vmware.1-vks.1 \
  --default-values-file-output istio-data-values.yaml -n packages

# Install the Istio VKS Package
vcf package install istio -p istio.kubernetes.vmware.com \
  --version 1.25.3+vmware.1-vks.1 --values-file istio-example.yaml -n packages
```

Note: for VKS versions 3.5 onwards, a new mechanism using add-ons is available (see https://techdocs.broadcom.com/us/en/vmware-cis/vcf/vsphere-supervisor-services-and-standalone-components/latest/managing-vsphere-kuberenetes-service-clusters-and-workloads/managing-add-ons-in-vks-clusters.html)

vmware by Broadcom

Item No: vmw-bc-wp-tech-temp-a4-word-2024 1/24