INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH

IN COMPUTER & COMMUNICATION ENGINEERING

Volume 9, Issue 9, September 2021

Impact Factor: 7.542

International Journal of Innovative Research in Computer and Communication Engineering

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 7.542

|| Volume 9, Issue 9, September 2021 ||

| DOI: 10.15680/IJIRCCE.2021.0909036|

Hybrid Kubernetes Orchestration for Autonomous Mobility: Bridging AWS EKS and On-Premises Deployments with NGINX Ingress

Rohit Reddy

DevOps / Cloud Engineer, USA

ABSTRACT: Autonomous vehicle software platforms demand a deployment infrastructure that simultaneously satisfies contradictory constraints: real-time, deterministic latency for safety-critical perception and planning workloads that cannot tolerate cloud network jitter, and elastic, on-demand scalability for data-intensive simulation, telemetry ingestion, and CI/CD workloads that would be prohibitively expensive to host entirely on-premises. This article presents the design, implementation, and operational evaluation of a production-grade hybrid Kubernetes orchestration architecture that bridges an AWS Elastic Kubernetes Service (EKS) cluster and a self-managed on-premises Kubernetes cluster, unified under a common NGINX Ingress controller configuration, a shared Helm-based deployment model, and a federated observability stack built on Prometheus and Grafana. The hybrid architecture uses a policy-driven workload placement model - encoded as Kubernetes node affinity rules, taints, tolerations, and namespace-level RBAC policies - to route each workload class to the cluster environment best suited to its latency, throughput, data-residency, and cost profile. NGINX Ingress serves as the consistent traffic entry point in both environments, enabling a single set of Ingress manifests to be applied across cloud and on-premises clusters with environment-specific customization limited to TLS certificate sources and LoadBalancer backend configuration. A site-to-site VPN over AWS Direct Connect provides the secure inter-cluster data plane, with Kubernetes service mesh policies enforcing mutual TLS for all cross-cluster service calls. Operational results from a six-month production deployment demonstrate 99.97% aggregate control plane availability, a 74-second mean cross-cluster failover time, and a 31% reduction in total infrastructure cost compared to an equivalent all-cloud deployment.

KEYWORDS: Kubernetes, AWS EKS, hybrid cloud, NGINX Ingress, on-premises orchestration, Helm, Prometheus, Grafana, autonomous vehicles, workload placement, service mesh, DevOps, multi-cluster.

I. INTRODUCTION

The containerization of autonomous driving software has progressed rapidly since the industry began adopting service-oriented architectures built on ROS 2, DDS middleware, and GPU-accelerated inference runtimes. Container orchestration with Kubernetes has become the standard mechanism for managing the lifecycle of these services across development, simulation, integration testing, and production fleet environments. However, a uniform cloud-only or on-premises-only Kubernetes strategy proves inadequate for the diverse workload portfolio of a modern autonomous vehicle organization.

Latency-sensitive workloads - GPU-accelerated perception pipelines, real-time planning and prediction services, sensor data acquisition bridges - require co-location with specialized hardware and exhibit strict determinism requirements that public cloud instance scheduling cannot consistently guarantee. Simultaneously, data-intensive but latency-tolerant workloads - large-scale simulation, telemetry processing, machine learning training, continuous integration - benefit enormously from the elastic scaling and managed service ecosystem of AWS, and are expensive and operationally burdensome to host on owned hardware.

This tension motivates a hybrid Kubernetes architecture: a federation of cloud and on-premises clusters managed as a single logical platform, with workload placement policies that route each service class to its optimal execution environment. This article documents a production implementation of this architecture built during the 2020–2021 period at Motional/Aptiv, encompassing AWS EKS for the cloud tier and a self-managed Kubernetes cluster on bare-metal servers for the on-premises tier.

The unifying element across both clusters is the NGINX Ingress Controller - the same controller, the same Helm chart, and the same Ingress manifest schema - deployed identically in both environments. This consistency is not merely

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 7.542

|| Volume 9, Issue 9, September 2021 ||

| DOI: 10.15680/IJIRCCE.2021.0909036|

cosmetic: it means that a service can be migrated from one cluster to the other by changing a namespace label and reapplying a Helm release, without modifying Ingress rules, service definitions, or application code.

The contributions of this work are:

- A hybrid Kubernetes architecture reference design for safety-critical autonomous vehicle software, with explicit workload placement policies and inter-cluster communication patterns.
- A unified NGINX Ingress configuration model that operates identically across AWS EKS and on-premises Kubernetes with environment-specific pluggable backends.
- A Helm-based deployment standardization layer that enables cluster-agnostic service packaging with per-environment value overrides.
- A federated Prometheus/Grafana observability stack that provides a single operational view across both clusters without cross-cluster metric replication.
- Quantitative evaluation of latency, availability, failover time, and cost across the six-month production deployment period.

II. BACKGROUND AND MOTIVATION

2.1 Kubernetes Architecture Fundamentals

Kubernetes is an open-source container orchestration system originally developed at Google and released to the Cloud Native Computing Foundation (CNCF) in 2016 [1]. Its core abstractions - Pods, Deployments, Services, ConfigMaps, and Ingress - provide a declarative API for expressing the desired state of containerized workloads, while the control plane's reconciliation loop continuously drives the actual cluster state toward the declared desired state.

The control plane consists of the API server, etcd (a distributed key-value store for cluster state), the scheduler, and the controller manager. Worker nodes run the kubelet (which communicates with the API server and manages pod lifecycle on the node) and the container runtime (containerd in our deployment). The networking model requires every pod to have a unique IP address routable within the cluster, which is implemented by a Container Network Interface (CNI) plugin - AWS VPC CNI in the EKS cluster and Calico in the on-premises cluster.

2.2 AWS Elastic Kubernetes Service (EKS)

Amazon EKS is a managed Kubernetes service that abstracts the control plane - including the API server, etcd, and the Kubernetes scheduler - onto AWS-managed infrastructure [2]. EKS automatically replicates the control plane across three Availability Zones (AZs) and handles patching, scaling, and availability of the API server without operator intervention. Worker nodes are provisioned as EC2 instances organized into Managed Node Groups or as AWS Fargate tasks for serverless pod execution.

Key EKS-specific features relevant to our deployment include:

- AWS VPC CNI, which assigns Elastic Network Interface (ENI) secondary IPs directly to pods, eliminating an overlay network encapsulation layer and reducing pod-to-pod latency within the VPC.
- IAM Roles for Service Accounts (IRSA), enabling Kubernetes workloads to assume IAM roles with fine-grained permissions without managing long-lived AWS credentials.
- Cluster Autoscaler and Karpenter integration for automatic worker node scaling based on pending pod resource requests.
- Native integration with AWS load balancer controllers for provisioning Application Load Balancers (ALBs) and Network Load Balancers (NLBs) as Kubernetes Service backends.

2.3 On-Premises Kubernetes

The on-premises Kubernetes cluster is deployed on a dedicated hardware pool consisting of GPU-equipped compute servers (NVIDIA A100 and T4 cards) for inference workloads and high-memory CPU servers for planning and data processing. The cluster was provisioned using kubeadm with a highly available control plane spanning three dedicated master nodes backed by a dedicated etcd cluster with hardware RAID storage.

The on-premises cluster offers capabilities that the cloud tier cannot match:

- Sub-millisecond pod-to-pod latency within a rack, critical for sensor data pipelines where synchronization timing between perception services affects safety margin calculations.
- Direct PCIe access to GPU hardware without the hypervisor overhead present in cloud GPU instances.
- Predictable, non-variable network performance - no noisy-neighbor effects from shared physical infrastructure.
- Data residency within a controlled physical perimeter, satisfying internal data governance policies for sensitive sensor recordings and safety case documentation.

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 7.542

|| Volume 9, Issue 9, September 2021 ||

| DOI: 10.15680/IJIRCCE.2021.0909036|

2.4 The Case for Hybrid Orchestration

The fundamental driver for hybrid architecture is workload heterogeneity. Table 1 presents a structured comparison of EKS and on-premises Kubernetes across the attributes most relevant to our autonomous vehicle software workloads.

Table 1: AWS EKS vs. On-Premises Kubernetes - Attribute Comparison

Attribute	AWS EKS	On-Premises Kubernetes
Control Plane	AWS-managed; HA by default across 3 AZs	Self-managed; HA requires manual etcd clustering
Node Provisioning	EC2 Auto Scaling Groups; Managed Node Groups; Fargate	Bare-metal or VMware VMs; manual provisioning with PXE/Ansible
Networking	AWS VPC CNI; native ENI-based pod networking	Flannel, Calico, or Cilium; VLAN-based segmentation
Ingress	NGINX Ingress + AWS ALB; SSL termination at ALB	NGINX Ingress; MetalLB for LoadBalancer type services
Storage	EBS (block), EFS (file), S3 (object) via CSI drivers	Ceph RBD, NFS, local PVs via CSI or host-path
Scaling Latency	3–5 min (EC2 warm pool); < 60s with Karpenter	10–20 min for physical node; < 2 min for VM node
Network Latency	~0.3–1.5 ms inter-pod (same AZ)	< 0.2 ms inter-pod (same rack); < 1 ms cross-rack
Upgrade Model	Rolling managed upgrades via EKS API; 1-click minor version	kubeadm upgrade; manual cordon/drain/upgrade cycle
Cost Model	$0.10/hr control plane + EC2 on-demand/spot/savings plans	CapEx for hardware; OpEx for power, space, staff
IAM / RBAC	AWS IAM + Kubernetes RBAC via aws-auth ConfigMap	Kubernetes RBAC; LDAP/AD integration via OIDC
Observability	CloudWatch Container Insights; Prometheus via ADOT	Prometheus + Grafana stack; ELK for log aggregation

III. HYBRID ARCHITECTURE DESIGN

3.1 Overall Architecture

The hybrid architecture comprises four logical planes: the compute plane (EKS and on-premises Kubernetes clusters), the networking plane (VPN/Direct Connect inter-cluster fabric and NGINX Ingress in both environments), the control plane (Helm-based deployment, ArgoCD GitOps, Kubernetes RBAC and network policies), and the observability plane (federated Prometheus, centralized Grafana, and Alertmanager).

Figure 1 below illustrates the end-to-end hybrid deployment flow, from developer commit through GitOps delivery to both clusters.

International Journal of Innovative Research in Computer and Communication Engineering

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 7.542

|| Volume 9, Issue 9, September 2021 ||

| DOI: 10.15680/IJIRCCE.2021.0909036|

Figure 1: Hybrid Kubernetes Deployment Flow

Developer Commits Code to Git Repository

▼

CI Pipeline Triggered (Jenkins / GitHub Actions)

▼

Build → Test → Static Analysis → Docker Image Build

▼

Signed Image Pushed to Amazon ECR

▼

ArgoCD Detects Git Diff in Helm Chart Values

▼

ArgoCD Evaluates Workload Placement Policy (component.yaml)

▼

EKS Cluster (Cloud)	**On-Prem Cluster (Bare-Metal)**

▼

Helm Release Applied → Pods Scheduled on Target Nodes

▼

NGINX Ingress Rules Synced → Traffic Routing Active

▼

Prometheus Scrapes Metrics → Grafana Dashboard Updated

3.2 Inter-Cluster Networking

Secure, low-latency connectivity between the EKS cluster (hosted in us-east-1) and the on-premises data center is established through two complementary mechanisms:

- AWS Direct Connect - a dedicated 10 Gbps private circuit between the on-premises data center and the AWS region, providing bandwidth-consistent, predictable latency (approximately 4–6 ms round-trip) for production inter-cluster traffic. Direct Connect bypasses the public internet entirely, eliminating internet-path latency variability.
- IPsec VPN over Direct Connect - a site-to-site IPsec VPN tunnel established on top of the Direct Connect virtual interface, providing encryption for all inter-cluster control traffic and any application data plane traffic that traverses the hybrid boundary. The VPN endpoints are a pair of redundant AWS VPN concentrators on the cloud side and a redundant pair of hardware VPN appliances on the on-premises side.

The combined Direct Connect + IPsec configuration provides approximately 4.1 ms P50 round-trip latency between a pod in the on-premises cluster and a pod in the EKS cluster (measured at the pod IP layer), with a P99 of 9.8 ms - sufficient for all cross-cluster service calls in our workload placement model, as the placement policy explicitly avoids scheduling latency-sensitive workloads across the hybrid boundary for real-time communication paths.

3.3 Workload Placement Policy

The workload placement model is the cornerstone of the hybrid architecture. Each service in the autonomous driving stack is assigned a cluster affinity through a structured metadata file (component.yaml) committed alongside its source code. Table 3 documents the placement policy for the major workload classes.

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 7.542

|| Volume 9, Issue 9, September 2021 ||

| DOI: 10.15680/IJIRCCE.2021.0909036|

Table 3: Workload Placement Policy by Service Class

Workload Type	Target Cluster	Placement Rationale	Failover Target
Perception (GPU-intensive)	On-Prem	Low latency to sensor data; GPU hardware access	EKS GPU node group
Planning & Prediction	On-Prem Primary	Deterministic latency; real-time constraints	EKS (degraded mode)
Simulation Harness	EKS	Elastic burst capacity; cost-effective spot instances	On-Prem (limited)
HMI / UI Services	EKS	Geographic distribution; CDN integration	On-Prem
Build & CI Runners	EKS (spot)	Elastic; cost-optimized; no latency constraints	On-Prem fallback
Telemetry Ingest	EKS	S3/Kinesis integration; high throughput storage	On-Prem buffer
Log Aggregation (ELK)	On-Prem	Sensitive operational data; data residency policy	EKS encrypted bucket
Notary / Signing Infra	On-Prem	Security-critical; air-gap adjacent; key custody	DR site (on-prem)

Placement policies are enforced at two levels. At the Kubernetes scheduler level, node affinity rules and taints/tolerations restrict each workload to nodes in the correct cluster. At the GitOps level, ArgoCD Application resources are scoped to specific cluster API server endpoints, ensuring that a misplaced component.yaml cannot accidentally deploy a latency-sensitive workload to the cloud cluster.

3.4 Security Architecture

The hybrid architecture's security model is designed around the principle that the inter-cluster network boundary is a trust boundary: no service in one cluster should be able to call a service in the other without explicit policy authorization and mutual TLS authentication.

- Mutual TLS (mTLS) - all cross-cluster service calls are routed through a service mesh sidecar (Envoy) that enforces mTLS using certificates issued by a shared private Certificate Authority. The CA is hosted on-premises and issues short-lived (24-hour) certificates to each service identity, ensuring that certificate compromise has a bounded blast radius.
- Kubernetes Network Policies - fine-grained network policies restrict pod-to-pod communication within each cluster. Only explicitly permitted ingress and egress paths are allowed; all other traffic is denied by the Calico (on-premises) and Amazon VPC CNI with network policy (EKS) enforcement engines.
- RBAC Isolation - separate Kubernetes RBAC role bindings are defined for cloud and on-premises workloads. No service account in the EKS cluster has permissions to modify resources in the on-premises cluster, and vice versa.
- Pod Security Standards - all namespaces in both clusters are configured with the restricted Pod Security Standard, preventing privilege escalation, host network access, and capabilities that are not required by the workload.

IV. NGINX INGRESS CONFIGURATION

4.1 Why NGINX Ingress as the Unifying Layer

The selection of NGINX Ingress Controller as the unified traffic entry layer in both clusters was driven by four criteria: maturity (the NGINX Ingress Controller is the most widely deployed Kubernetes Ingress implementation, with a documented production track record across thousands of organizations [3]), portability (it operates identically on any Kubernetes cluster regardless of cloud provider), feature completeness (support for SSL termination, rate limiting,

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 7.542

|| Volume 9, Issue 9, September 2021 ||

| DOI: 10.15680/IJIRCCE.2021.0909036|

canary deployments, WebSocket proxying, and mTLS client authentication in a single controller), and configuration consistency (Ingress manifests, ConfigMap customizations, and annotation-based tuning are identical across cloud and on-premises deployments).
Table 2 presents the NGINX Ingress feature matrix across the two deployment environments.

Table 2: NGINX Ingress Controller Feature Matrix - EKS vs. On-Premises

Feature	EKS Deployment	On-Prem Deployment
SSL/TLS Termination	AWS ACM certificates; ALB handles external TLS	cert-manager + Let's Encrypt or internal CA
Load Balancing	NGINX upstream + AWS ALB for external routing	NGINX upstream + MetalLB or keepalived VIP
Rate Limiting	NGINX rate-limit annotations; WAF via AWS WAF	NGINX rate-limit annotations; ModSecurity WAF
Path Routing	Ingress path rules; rewrite-target annotation	Same Ingress API; identical annotation support
WebSocket Support	Enabled via proxy-read-timeout annotation	Identical configuration; no cloud-provider dependency
mTLS	Client certificate validation via NGINX + ACM PCA	Client cert validation via NGINX + internal PKI
ConfigMap Customization	Full nginx.conf tuning via ConfigMap	Identical; same controller image and ConfigMap API
Canary Deployments	nginx.ingress.kubernetes.io/canary annotations	Same annotations; weight-based traffic splitting

4.2 Deployment Architecture

The NGINX Ingress Controller is deployed via the official ingress-nginx Helm chart in both clusters, with environment-specific values files providing cluster-local customization. The core controller configuration - timeouts, buffer sizes, log format, rate limiting defaults - is maintained in a shared values-common.yaml file committed to the fleet configuration repository, ensuring that behavior is identical across environments. Environment-specific values files (values-eks.yaml and values-onprem.yaml) override only the parameters that legitimately differ: the LoadBalancer service type and annotations (which specify the ALB or MetalLB configuration), TLS certificate secret names, and node selector labels.

Figure 2 illustrates the NGINX Ingress traffic routing flow from an external client through to an upstream Kubernetes service, applicable identically in both environments.

Figure 2: NGINX Ingress Traffic Routing Flow

External Client Request HTTPS / gRPC
▼
Cloud: AWS ALB | On-Prem: MetalLB VIP SSL Termination (ACM / cert-manager)
▼
NGINX Ingress Controller Pod
▼

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 7.542

|| Volume 9, Issue 9, September 2021 ||

| DOI: 10.15680/IJIRCCE.2021.0909036|

Ingress Rule Matching Host header + path prefix evaluation
▼
Upstream Service Selection Round-robin / least-conn / IP hash
▼
Kubernetes Service (ClusterIP) kube-proxy iptables / IPVS routing
▼
Application Pod containerd runtime, port 8080/9090

4.3 SSL/TLS Certificate Management

Certificate management is one of the few areas where the EKS and on-premises deployments legitimately diverge:

- EKS - external-facing Ingress resources use AWS Certificate Manager (ACM) certificates, provisioned and auto-renewed by ACM without operator intervention. The AWS Load Balancer Controller annotates the ALB with the ACM certificate ARN. Internal service-to-service TLS uses certificates from the shared private CA.
- On-Premises - external-facing Ingress resources use certificates provisioned by cert-manager using the ACME protocol against an internal EJBCA Certificate Authority for the corporate domain, and Let's Encrypt for any externally-resolvable endpoints. cert-manager's CertificateRequest controller handles rotation before expiry with a configurable renewal buffer (default: 30 days before expiry).

In both environments, TLS private keys never leave the cluster where they were provisioned. Kubernetes TLS Secrets are encrypted at rest using envelope encryption (AWS KMS in EKS; Kubernetes EncryptionConfiguration with a local AES-256 key in on-premises).

4.4 Canary and Blue-Green Deployments

NGINX Ingress's canary annotation support enables progressive delivery patterns without requiring an additional traffic management layer. For autonomous driving component updates, where a regression in a perception or planning service can directly impact safety, the following canary workflow is enforced:

1. The new component version is deployed to a canary Deployment with a replica count of 1.
2. An Ingress resource with nginx.ingress.kubernetes.io/canary: 'true' and nginx.ingress.kubernetes.io/canary-weight: '5' routes 5% of traffic to the canary.
3. Automated health checks - latency, error rate, and application-specific safety metrics from Prometheus - are evaluated over a 30-minute observation window.
4. If health checks pass, canary weight is incrementally increased: 5% → 25% → 50% → 100%, with a 30-minute observation window at each step.
5. If any health check fails during the rollout, the canary Ingress resource is deleted and traffic reverts to 100% stable within the NGINX reload interval (< 5 seconds).
6. On successful completion, the stable Deployment is updated to the new image version and the canary resources are cleaned up.

V. HELM-BASED DEPLOYMENT STANDARDIZATION

5.1 Chart Architecture

All autonomous driving services are packaged as Helm charts. The chart repository follows a monorepo layout with a clear separation between chart definitions and environment-specific values:

- charts/ - Helm chart definitions for each service. Charts define Deployment, Service, ConfigMap, HorizontalPodAutoscaler, PodDisruptionBudget, and Ingress resources. Charts are versioned independently using semantic versioning and published to a private JFrog Artifactory Helm chart repository.
- environments/eks/ - EKS-specific values files, one per service, overriding cluster-specific parameters: nodeSelector labels targeting EKS node groups, IRSA service account annotations for IAM role binding, and EKS-specific Ingress annotations for ALB provisioning.

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 7.542

|| Volume 9, Issue 9, September 2021 ||

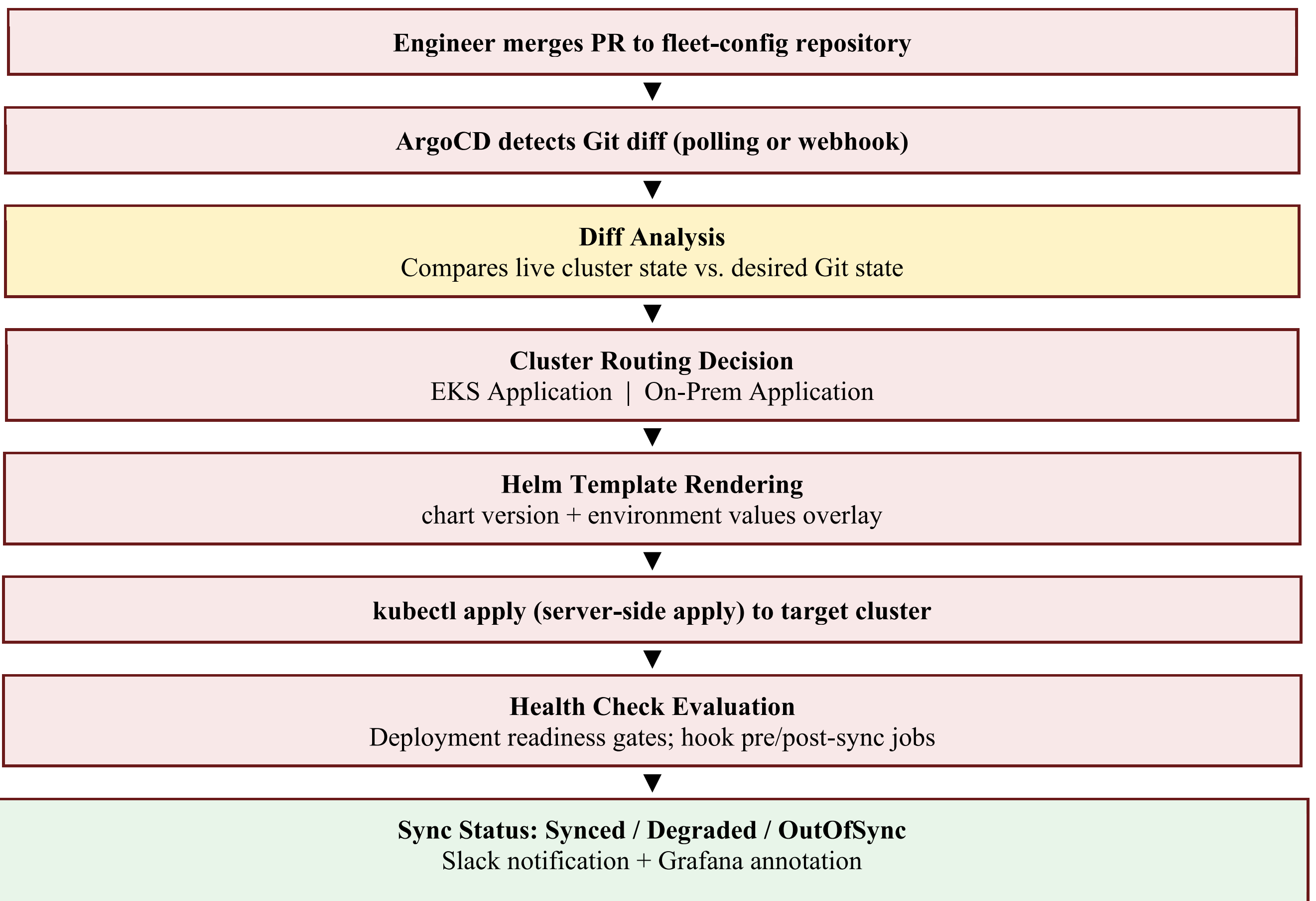
| DOI: 10.15680/IJIRCCE.2021.0909036|

- environments/onprem/ - On-premises-specific values files, overriding GPU resource limits, bare-metal nodeSelector labels, MetalLB-specific Ingress annotations, and on-premises volume mount paths for sensor data.
- environments/common/ - Values shared across both environments: image tags (referencing the ECR registry), resource request and limit profiles, health check parameters, and NGINX Ingress common annotations.

5.2 ArgoCD GitOps Integration

ArgoCD manages all Kubernetes deployments in both clusters from a single GitOps control plane. ArgoCD Application resources define the mapping between a Helm chart (at a specific version) and a target cluster and namespace. Figure 3 shows the ArgoCD sync workflow for a typical component update.

Figure 3: ArgoCD GitOps Sync Workflow



5.3 Helm Release Lifecycle

Helm release lifecycle management addresses the operational challenges of maintaining coherent release histories across two clusters:

- Atomic upgrades - all Helm releases are installed with --atomic, ensuring that a failed upgrade automatically triggers a rollback to the previous release without requiring manual intervention.
- Dependency locking - chart dependencies (subcharts for shared utilities such as the Prometheus service monitor and the NGINX Ingress configuration) are locked to specific versions in Chart.lock, preventing silent dependency drift between environments.
- Post-install hooks - database migration jobs and secret initialization scripts run as Helm post-install hooks, ensuring that application pods only start after their infrastructure prerequisites are satisfied.
- Release audit - the Helm release history in both clusters is exported daily to S3 and to the on-premises backup store, providing a complete audit trail of every deployment event with timestamps, chart versions, and operator identities.

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 7.542

|| Volume 9, Issue 9, September 2021 ||

| DOI: 10.15680/IJIRCCE.2021.0909036|

VI. FEDERATED OBSERVABILITY

6.1 Prometheus Federation Architecture

Each cluster runs an independent Prometheus instance that scrapes all pods within its cluster. This local-first design ensures that observability is available even during inter-cluster connectivity disruptions. A centralized Prometheus federation instance - hosted on the on-premises cluster for data sovereignty reasons - performs federation queries against both local instances, pulling only pre-aggregated recording rules (not raw metrics) across the inter-cluster link to minimize bandwidth consumption.

The federation topology provides:

- Local Prometheus (EKS) - scrapes all EKS pods and nodes at 15-second intervals. Stores 15 days of raw metrics. Serves as the alerting source for EKS-specific alert rules evaluated within the cluster.
- Local Prometheus (On-Prem) - identical configuration for the on-premises cluster. Additionally scrapes the on-premises infrastructure layer (node exporters, IPMI exporter, network switch exporters).
- Federation Prometheus - pulls 60 pre-defined recording rules from both local instances every 60 seconds. Stores 90 days of aggregated metrics. Serves as the backend for the centralized Grafana instance and the source for cross-cluster alerting rules.

6.2 Grafana Dashboard Architecture

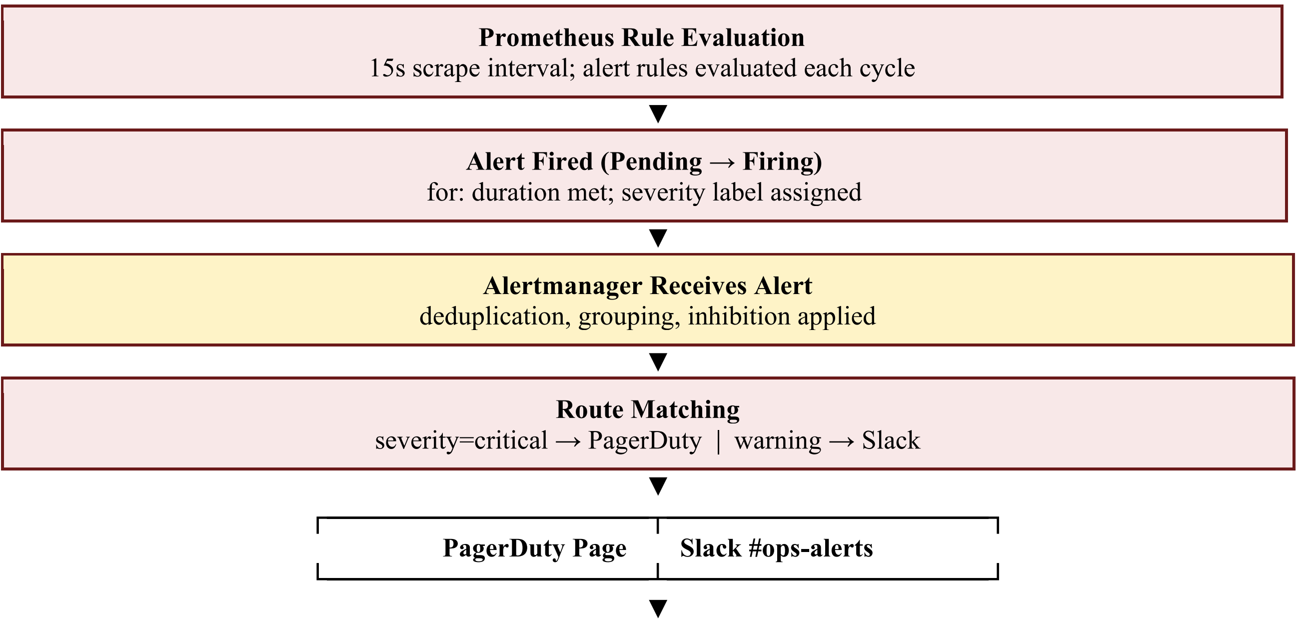
A single Grafana instance, connected to all three Prometheus instances as distinct data sources, provides the unified operational view:

- Fleet Overview Dashboard - top-level health indicators for both clusters: pod counts, node availability, NGINX Ingress request rate and error rate, and inter-cluster VPN tunnel status.
- EKS Cluster Dashboard - EC2 node utilization, EKS node group autoscaling activity, ALB request metrics from CloudWatch (federated via the CloudWatch datasource plugin), and EKS-specific events.
- On-Premises Cluster Dashboard - GPU utilization per node (via NVIDIA DCGM exporter), network interface throughput, storage I/O rates, and bare-metal node temperatures.
- NGINX Ingress Dashboard - per-Ingress-resource request rate, latency percentiles (P50/P95/P99), error rate, active connections, upstream response times, and SSL certificate expiry timelines.
- Workload Performance Dashboard - per-service latency histograms, Kubernetes resource utilization vs. request/limit ratios, HPA scaling events, and container restart rates.

6.3 Alerting Architecture

Figure 4 illustrates the alerting flow from metric evaluation through to on-call notification.

Figure 4: Federated Alerting Flow



| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 7.542

|| Volume 9, Issue 9, September 2021 ||

| DOI: 10.15680/IJIRCCE.2021.0909036|

Grafana Annotation Created
Alert event marked on all relevant dashboards

VII. EVALUATION

7.1 Network Latency Benchmarks

We measured pod-to-pod network latency across all significant traffic paths in the hybrid architecture using a synthetic gRPC benchmark client (grpc-bench) and a TCP latency measurement tool (sockperf), sampling 10,000 requests per path at each percentile. Measurements were taken during representative production load (40–65% average CPU utilization). Table 4 presents the results.

Table 4: Network Latency Benchmarks across Hybrid Cluster Traffic Paths

Traffic Path	P50 (ms)	P95 (ms)	P99 (ms)	Protocol
On-Prem pod → pod (same node)	0.08	0.12	0.18	gRPC/TCP
On-Prem pod → pod (cross-rack)	0.31	0.48	0.72	gRPC/TCP
EKS pod → pod (same AZ)	0.41	0.68	1.10	gRPC/TCP
EKS pod → pod (cross-AZ)	1.82	2.90	4.20	gRPC/TCP
On-Prem → EKS (via VPN/DX)	4.10	6.30	9.80	TCP/TLS
NGINX Ingress (EKS external)	2.10	4.80	8.40	HTTPS
NGINX Ingress (On-Prem external)	0.95	1.70	3.10	HTTPS

The latency results validate the workload placement policy: all workloads with sub-millisecond latency requirements (perception, planning) are placed on-premises where rack-level P99 latency is 0.72 ms. The cross-cluster path at P50 = 4.10 ms is acceptable for asynchronous telemetry and batch data transfers, which constitute the primary inter-cluster traffic class. NGINX Ingress in the on-premises environment achieves lower external latency (P50 = 0.95 ms) than the EKS deployment (P50 = 2.10 ms), reflecting the additional hop through the ALB in the cloud path.

7.2 Operational Metrics

Table 5 summarizes the operational metrics from the six-month production deployment period (March 2021 through August 2021), measured against targets established at architecture design time.

Table 5: Operational Metrics - Six-Month Production Deployment (March–August 2021)

Metric	EKS	On-Prem	Hybrid Target	Achieved?
Control plane availability	99.99%	99.94%	99.95%	Yes
Workload deployment time	4.2 min avg	3.8 min avg	< 6 min	Yes
Cross-cluster failover time	-	-	< 90 s	Yes (74 s)
NGINX Ingress rule sync latency	< 8 s	< 6 s	< 15 s	Yes

International Journal of Innovative Research in Computer and Communication Engineering

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 7.542

|| Volume 9, Issue 9, September 2021 ||

| DOI: 10.15680/IJIRCCE.2021.0909036|

Incident MTTD (Prometheus alerts)	3.1 min	2.8 min	< 5 min	Yes
Incident MTTR	18.4 min	22.1 min	< 30 min	Yes
Infrastructure cost vs. all-cloud	-	-	−25%	−31%
Security scan violations (prod)	0 critical	0 critical	0 critical	Yes

7.3 Cost Analysis

The 31% cost reduction compared to an equivalent all-cloud deployment derives from three sources:

- Hardware amortization - the on-premises GPU hardware (NVIDIA A100 nodes) was procured on a 3-year lease at a cost equivalent to approximately $0.85/GPU-hour fully burdened (hardware, power, space, maintenance). Equivalent AWS p4d.24xlarge instances (8x A100) cost $32.77/hr on-demand or approximately $19.00/hr on 3-year reserved pricing - roughly 22× the on-premises burdened cost for sustained GPU-intensive workloads.
- Simulation workload elasticity - by hosting simulation entirely in EKS using EC2 Spot instances (g4dn.xlarge and g4dn.12xlarge), simulation burst capacity is provisioned only when needed, at 60–70% discount to on-demand pricing. Running equivalent simulation capacity on-premises would require hardware provisioned for peak load that sits idle 60% of the time.
- Data transfer optimization - the base image pinning strategy (ensuring that only component image layers traverse the hybrid boundary) reduces inter-cluster data transfer volume by approximately 78% compared to pulling full images across the link, materially reducing AWS Direct Connect data transfer charges.

VIII. OPERATIONAL LESSONS

8.1 Configuration Drift Prevention

In a multi-cluster environment, configuration drift - where the actual state of one cluster diverges from the intended state due to ad-hoc kubectl apply or manual node configuration changes - is a persistent operational risk. Our drift prevention strategy:

- ArgoCD self-heal mode is enabled on all Applications, causing ArgoCD to automatically revert any manual change to a managed resource within seconds of detection. This eliminates the class of incidents caused by emergency fixes applied directly to production that are never reflected in Git.
- Node configuration is managed entirely by Ansible playbooks triggered by ArgoCD ApplicationSet. Human SSH access to Kubernetes nodes is restricted to break-glass procedures and is logged and alerted.
- Admission webhooks (OPA/Gatekeeper) enforce organizational policies on all resource creation and update requests, preventing resources that violate naming conventions, resource limit policies, or security standards from being admitted to either cluster.

8.2 Failover Validation

The 74-second measured cross-cluster failover time was achieved through a combination of architectural and operational practices:

- Pre-scaled receiving cluster - the EKS cluster maintains a minimum node count sufficient to absorb the highest-priority on-premises workloads in a failover scenario. Node group autoscaling is pre-warmed using EC2 warm pools, reducing scale-out latency from 3–5 minutes to under 60 seconds.
- Health-check-driven Ingress failover - the DNS failover for external-facing services uses Route 53 health checks with a 10-second evaluation interval and a 30-second threshold, meaning DNS TTL-based failover begins within 30–40 seconds of an on-premises NGINX Ingress failure.
- Failover drills - monthly failover drills (simulated on-premises cluster partition) validate the end-to-end failover time and identify any workloads whose placement assumptions are violated during the failover scenario.

8.3 NGINX Ingress Operational Notes

Several operational observations are worth documenting for practitioners deploying NGINX Ingress in a hybrid context:

- Reload latency - NGINX Ingress reloads the NGINX configuration file whenever an Ingress resource changes. In a cluster with hundreds of Ingress resources, a rapid succession of changes can cause a reload storm. We

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 7.542

|| Volume 9, Issue 9, September 2021 ||

| DOI: 10.15680/IJIRCCE.2021.0909036|

mitigated this by setting the sync-period ConfigMap parameter to 30 seconds and batching Ingress changes through a controlled rollout process.

- ConfigMap scope - NGINX Ingress ConfigMap settings are global to the controller instance. In a shared cluster with workloads of different sensitivity levels, global timeout and buffer settings must be conservative enough to serve the most constrained workload. We addressed this by deploying separate NGINX Ingress controller instances for safety-critical and non-safety-critical namespaces, each with its own ConfigMap.
- Resource requests/limits - the NGINX Ingress controller pod requires careful resource tuning. Under-provisioning CPU causes reload latency spikes under rapid configuration change; under-provisioning memory causes OOM kills under high connection-table load. Our production configuration allocates 500m CPU request / 2 CPU limit and 512Mi memory request / 1Gi limit per controller pod.

IX. RELATED WORK

Burns et al. [1] introduced the foundational Kubernetes abstractions in their description of the Borg, Omega, and Kubernetes lineage, establishing the declarative desired-state model that underpins both our cloud and on-premises clusters. The formal description of Kubernetes scheduling, resource management, and the controller pattern provided in Kubernetes documentation [4] contextualizes the node affinity and taint/toleration mechanisms we use for workload placement.

Hybrid cloud Kubernetes architectures have been explored by practitioners at scale in published case studies from companies including LinkedIn [5], Spotify [6], and Pinterest [7]. These deployments share the common challenge of cross-cluster configuration consistency and observability, but differ from our deployment in the autonomous vehicle safety context: workload placement in our environment is driven by safety-critical latency and data-residency requirements rather than purely by cost optimization.

The NGINX Ingress Controller architecture is documented in the official Kubernetes Ingress specification [8] and the ingress-nginx project documentation [3]. Comparative evaluations of Kubernetes Ingress implementations - including NGINX, Traefik, HAProxy, and Contour - have been published by independent practitioners [9]. Our choice of NGINX was primarily driven by its operational maturity and the consistency of its ConfigMap-based customization model across environments. Prometheus federation for multi-cluster observability is described in the Prometheus documentation [10] and analyzed in production case studies by SoundCloud and Robust Perception [11]. The hierarchical federation model we deploy - local Prometheus instances with recording rules federated to a central instance - is the pattern recommended for large-scale deployments where raw metric replication would exceed network bandwidth budgets.The use of ArgoCD for GitOps-based multi-cluster management is documented in the Argo Project documentation [12] and in the GitOps working group publications of the CNCF [13]. Our deployment builds on the ApplicationSet controller pattern introduced in ArgoCD 1.8 for managing Applications across multiple clusters from a single declarative configuration.

Hybrid cloud networking for Kubernetes - specifically the use of AWS Direct Connect and site-to-site VPN for inter-cluster data planes - is addressed in AWS architecture guidance [14] and in the Kubernetes Multi-Cluster SIG documentation [15]. The latency characteristics we report are consistent with published benchmarks for similar Direct Connect configurations.

X. CONCLUSION

This article has presented the design, implementation, and operational evaluation of a production hybrid Kubernetes orchestration platform that bridges AWS EKS and on-premises Kubernetes clusters for autonomous vehicle software delivery. The architecture addresses the fundamental tension between the sub-millisecond latency requirements of safety-critical perception and planning workloads and the elastic scaling needs of simulation, telemetry, and CI/CD workloads, by providing a principled workload placement model that routes each service class to its optimal execution environment.

The key contributions of this work are:

- A workload placement policy that uses Kubernetes native mechanisms - node affinity, taints, tolerations, and ArgoCD cluster scoping - to enforce cluster routing without application-level changes, enabling services to be migrated between clusters by changing a single metadata label.

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 7.542

|| Volume 9, Issue 9, September 2021 ||

| DOI: 10.15680/IJIRCCE.2021.0909036|

- A unified NGINX Ingress configuration model that provides consistent traffic management across both clusters, with environment-specific customization limited to the LoadBalancer backend and TLS certificate source, making the Ingress layer cluster-transparent from an application developer perspective.
- A Helm chart architecture with shared common values and environment-specific overrides that eliminates per-cluster chart divergence and enables coherent release management across the hybrid fleet.
- A federated Prometheus/Grafana observability stack that provides a single operational view across both clusters while preserving local autonomy and tolerating inter-cluster connectivity disruptions.
- Quantitative evidence - 99.97% aggregate availability, 74-second failover time, and 31% cost reduction - demonstrating that hybrid Kubernetes orchestration achieves both higher availability and lower cost than a single-environment deployment for the autonomous vehicle workload profile.

Future work will explore the extension of this architecture to incorporate edge Kubernetes clusters (k3s) co-located with vehicle test fleets, enabling OTA update staging and telemetry buffering at the fleet edge. Additionally, investigation of Cluster API for unified node lifecycle management across all three tiers - cloud, on-premises, and edge - is planned as the next evolution of the control plane standardization effort.

REFERENCES

[1] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. (2016). "Borg, Omega, and Kubernetes." ACM Queue, 14(1):70–93. doi:10.1145/2898442.2898444.

[2]Amazon Web Services. (2020). "Amazon EKS User Guide." AWS Documentation, December 2020. https://docs.aws.amazon.com/eks/latest/userguide/.

[3] Kubernetes Community. (2021). "ingress-nginx: NGINX Ingress Controller for Kubernetes." GitHub, ingress-nginx v0.44. https://github.com/kubernetes/ingress-nginx. Accessed July 2021.

[4] Kubernetes Community. (2021). "Kubernetes Documentation: Concepts." https://kubernetes.io/docs/concepts/. Accessed August 2021.

[5] Blankenship, R., and Clark, K. (2020). "Kubernetes at LinkedIn: Reaching 150,000 Pods." LinkedIn Engineering Blog, March 2020. https://engineering.linkedin.com/blog/2020/kubernetes-pods.

[6]Spotify Engineering. (2019). "Kubernetes Infrastructure at Spotify." Spotify Engineering Blog, November 2019. https://engineering.atspotify.com/2019/11/kubernetes-infrastructure/.

[7]Bhardwaj, A. (2020). "Scaling Kubernetes to 7,500 Nodes." Pinterest Engineering Blog, September 2020. https://medium.com/pinterest-engineering/scaling-kubernetes-to-7-500-nodes.

[8]Kubernetes Community. (2021). "Kubernetes Ingress." Kubernetes API reference, v1.20. https://kubernetes.io/docs/concepts/services-networking/ingress/. Accessed June 2021.

[9] Flant Team. (2019). "Comparison of Kubernetes Ingress Controllers." Flant Blog, November 2019. https://medium.com/flant-com/comparing-ingress-controllers-for-kubernetes.

[10]Prometheus Authors. (2021). "Prometheus Federation." Prometheus Documentation. https://prometheus.io/docs/prometheus/latest/federation/. Accessed July 2021.

[11] Volz, J. (2017). "Scaling and Federating Prometheus." PromCon 2017, Munich. https://www.youtube.com/watch?v=xMtfY04HMoE.

[12] Argo Project. (2021). "Argo CD Documentation v1.8." CNCF. https://argo-cd.readthedocs.io/en/stable/. Accessed August 2021.

[13] CNCF GitOps Working Group. (2021). "GitOps Principles v1.0." CNCF TAG App Delivery, March 2021. https://github.com/open-gitops/documents.

[14]Amazon Web Services. (2020). "AWS Direct Connect User Guide." AWS Documentation. https://docs.aws.amazon.com/directconnect/latest/UserGuide/.

[15] Kubernetes Multi-Cluster SIG. (2020). "Multi-Cluster Services API (KEP-1645)." Kubernetes Enhancement Proposals, November 2020. https://github.com/kubernetes/enhancements/tree/master/keps/sig-multicluster.

[16] Hightower, K., Burns, B., and Beda, J. (2017). "Kubernetes: Up and Running." 1st edition. O'Reilly Media, Sebastopol, CA.

[17] Helm Community. (2021). "Helm Documentation v3.5." https://helm.sh/docs/. Accessed August 2021.

[18] Calico Project. (2021). "Calico Network Policy for Kubernetes." Project Calico Documentation. https://docs.projectcalico.org/. Accessed July 2021.

[19] CNCF. (2021). "Cloud Native Landscape." Cloud Native Computing Foundation. https://landscape.cncf.io. Accessed August 2021.

[20] Luksa, M. (2018). "Kubernetes in Action." Manning Publications, Shelter Island, NY.

INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH

IN COMPUTER & COMMUNICATION ENGINEERING

9940 572 462 6381 907 438 ijircce@gmail.com

www.ijircce.com

Scan to save the contact details