NextGen Scientific Publication
American International Journal of Computer Science and Technology

Volume 8 Issue 1, Pg. No. 1- 8, AIJCST-V8I1P101, 2026
https://doi.org/10.63282/3117-5481/AIJCST-V8I1P101

Original Article

Scalable CI/CD Architecture Using Multi-Fleet Controllers and HAProxy for Cluster Management in Kubernetes

***Srinivas Thotakura**
Staff Software Development Engineering, CVS Health, USA.

Article History:

Received: 10.11.2025

Revised: 13.12.2025

Accepted: 22.12.2025

Published: 03.01.2026

Abstract:

Modernizing large-scale enterprise systems presents significant challenges due to tightly coupled legacy architectures, complex deployment dependencies, and high operational risks. The transition from monolithic applications to cloud-native microservices orchestrated by Kubernetes further amplifies deployment and scalability concerns, particularly in environments managing thousands of clusters. Traditional Kubernetes management platforms face inherent scalability limitations, creating bottlenecks in deployment governance, observability, and reliability. This paper presents a scalable continuous integration and continuous deployment (CI/CD) architecture leveraging multiple standalone Fleet controllers integrated with HAProxy for centralized routing and cluster management. By adopting a GitOps-driven deployment model using Fleet, combined with deterministic cluster-to-controller assignment and label-based rollout strategies, the proposed architecture efficiently manages over 10,000 Kubernetes clusters. HAProxy enables seamless access to distributed Fleet controllers through path-based routing, significantly reducing operational complexity and infrastructure overhead.The architecture incorporates automated fleet agent registration using SUSE Manager (MLM), eliminating manual intervention and ensuring balanced controller utilization. Performance optimizations—including etcd quota tuning and HAProxy connection scaling—address real-world operational constraints encountered during large-scale deployments. Experimental results demonstrate improved scalability, reduced deployment latency, and enhanced reliability. This work provides a practical reference architecture for enterprises seeking to implement resilient, large-scale Kubernetes CI/CD systems.

Keywords:

Kubernetes, CI/CD, GitOps, Fleet, HAProxy, Multi-Cluster Management, Microservices, Cloud-Native Architecture, DevOps Automation.

I. Introduction

The rapid adoption of Kubernetes has transformed how enterprises design, deploy, and operate distributed applications. As organizations modernize legacy monolithic systems into microservices-based architectures, deployment orchestration becomes increasingly complex—particularly in environments operating at massive scale. Managing thousands of Kubernetes clusters introduces challenges related to deployment consistency, configuration drift, scalability, and operational governance.

In large enterprise environments, deployment reliability directly impacts system availability and business continuity. Traditional CI/CD pipelines often struggle to scale across geographically distributed clusters, leading to fragmented tooling and increased

operational overhead. GitOps-based deployment models have emerged as a solution, providing declarative configuration, version control, and automated reconciliation. Fleet, a GitOps-compatible deployment engine integrated with Rancher, enables centralized management of Kubernetes clusters through Git repositories. While Rancher supports Fleet natively, its scalability is limited to approximately 500 clusters per instance, making it unsuitable for environments exceeding several thousand clusters without significant infrastructure duplication.

To address these limitations, this paper proposes a scalable CI/CD architecture using multiple standalone Fleet controllers combined with HAProxy-based routing. The solution distributes cluster management responsibilities across four Fleet controllers, each capable of managing up to 2,500 clusters, while presenting a unified access layer. This approach significantly reduces infrastructure complexity, improves scalability, and enhances operational efficiency.

1.1. Architecture Overview

1.1.1. Fleet Controller Architecture

To support large-scale Kubernetes deployments across 10,000 clusters, we implemented a distributed architecture using four standalone Fleet controller clusters. Each Fleet controller acts as a container management and deployment engine, offering fine-grained control over local clusters and continuous monitoring through GitOps. Fleet not only scales effectively but also provides visibility into the exact state of resources deployed across clusters.

Below is the fleet architecture from the official fleet documentation.

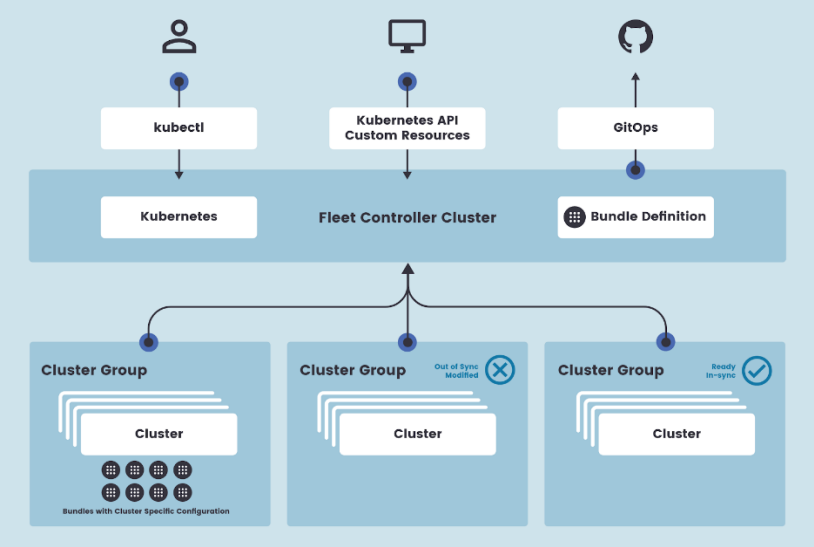

Figure 1. Fleet architecture and deployment flow

- Each Fleet controller cluster is built on RKE2 and consists of:
- 1 master node
- 2 worker nodes. All three nodes serve as control plane, etcd, and master components to ensure high availability and performance.

To optimize performance for large-scale deployments, the following configuration parameters were applied:

Table 1. Kubernetes Cluster Configuration Summary

Category	Parameter	Value/Setting
Network Configuration	cluster-cidr	192.168.10.0/20
	service-cidr	192.168.18.0/20
Security	Selinux	True
Etcd Configuration	quota-backend-bytes	8,589,934,592 (8 GB)

	listen-metrics-urls	Enabled
	auto-compaction-mode	periodic
	auto-compaction-retention	1h
	max-request-bytes	33,554,432 (32 MB)
	max-txn-ops	1024
Kube API Server Configuration	Kube API Server Configuration	2000
	max-mutating-requests-inflight	1000

These configurations ensure that each Fleet controller can handle up to 2,500 clusters efficiently, with robust performance and minimal latency. The architecture is designed to be scalable, secure, and maintainable, forming the backbone of our GitOps-driven deployment strategy.

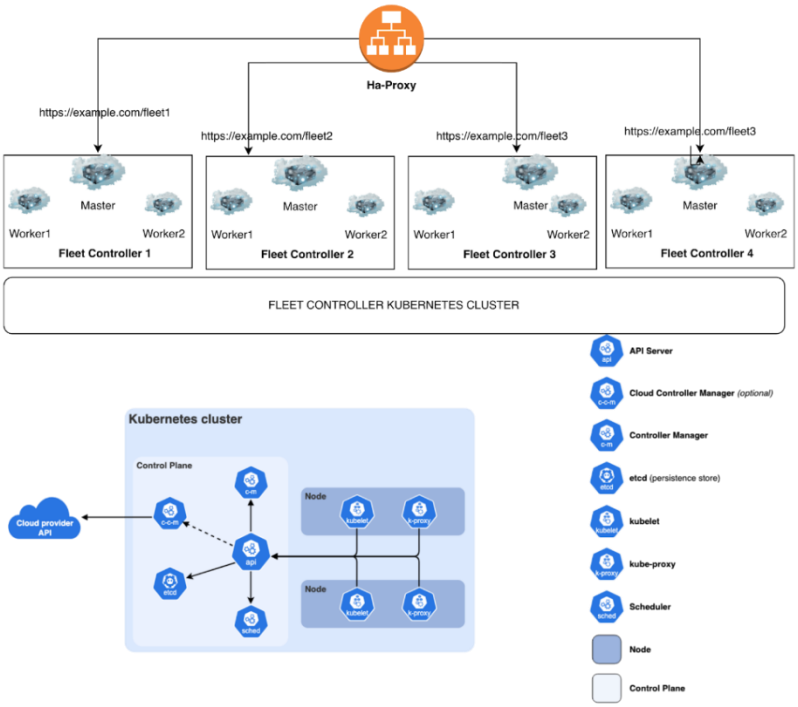

Figure 2. Four-Fleet Controller Architecture with Haproxy

2. Literature Review

The rapid evolution of cloud-native computing has driven widespread adoption of Kubernetes as the de facto orchestration platform for containerized applications. While Kubernetes provides powerful primitives for deployment, scaling, and resilience, managing deployments across large-scale, multi-cluster environments remains a significant research and operational challenge. Existing literature highlights limitations in scalability, control-plane performance, and operational governance when Kubernetes is deployed at enterprise scale.

2.1. Kubernetes and Large-Scale Cluster Management

Kubernetes was originally designed to manage containerized workloads within a single cluster, emphasizing declarative configuration and automated reconciliation [1]. As enterprise environments expanded, multi-cluster Kubernetes architectures became necessary to address geographical distribution, fault isolation, and regulatory requirements. Burns *et al.* [1] and Hightower *et al.* [2] describe Kubernetes' architectural foundations but acknowledge that native Kubernetes lacks built-in mechanisms for managing thousands of clusters centrally.

Several studies propose federation-based approaches to multi-cluster management; however, Kubernetes Federation (KubeFed) has faced adoption challenges due to operational complexity and limited scalability [3]. As a result, external platforms such as Rancher, Anthos, and OpenShift have emerged to fill this gap by providing centralized visibility and control. Despite these advancements, prior work identifies control-plane scalability and metadata management—particularly etcd storage growth—as critical bottlenecks in large deployments [4].

2.2. GitOps and Declarative Deployment Models

GitOps has gained prominence as a deployment paradigm that uses Git repositories as the single source of truth for system state. Weaveworks formally introduced GitOps as a model for continuous delivery, emphasizing version-controlled infrastructure, automated reconciliation, and auditable change management [5]. Empirical studies demonstrate that GitOps improves deployment consistency, rollback reliability, and security compliance in cloud-native systems [6].

Tools such as Argo CD and Flux operationalize GitOps principles by continuously synchronizing cluster state with Git repositories. While these tools perform well at a moderate scale, recent literature indicates that managing thousands of clusters introduces polling overhead, repository sprawl, and synchronization delays [7]. Fleet extends the GitOps model by introducing hierarchical configuration and centralized multi-cluster orchestration, making it particularly suitable for large-scale environments [8].

2.3. CI/CD Pipelines for Cloud-Native Systems

Continuous integration and continuous deployment (CI/CD) pipelines are fundamental to modern DevOps practices. Forsgren *et al.* [9] empirically demonstrate that automated pipelines significantly improve deployment frequency, mean time to recovery, and system stability. However, scaling CI/CD pipelines across thousands of clusters introduces new challenges related to coordination, approval workflows, and targeted rollouts.

Label-based deployment strategies have been proposed as an effective mechanism for selective rollouts, enabling canary and phased deployments without impacting the entire infrastructure [10]. Helm-based templating further enhances reusability and parameterization in Kubernetes deployments, although managing Helm releases at scale requires careful orchestration to avoid configuration drift [11].

2.4. Control-Plane Scalability and etcd Limitations

etcd serves as the primary data store for Kubernetes control planes, maintaining cluster state, metadata, and configuration. Prior studies identify etcd storage growth and request throughput as key constraints in large-scale Kubernetes systems [12]. Default etcd configurations—such as a 2 GB backend quota—are insufficient for environments managing thousands of clusters or large volumes of GitOps metadata.

Research emphasizes the importance of tuning etcd parameters, including backend quotas, compaction policies, and request limits, to ensure stability at scale [13]. Failure to do so can result in degraded API server performance, increased latency, and system instability.

2.5. Load Balancing and HAProxy in Distributed Architectures

Load balancers play a critical role in ensuring high availability and scalability in distributed systems. HAProxy is widely adopted due to its high performance, flexible routing capabilities, and support for advanced health checks [14]. Path-based routing has been extensively studied as an effective strategy for multi-tenant systems, allowing logical separation of backend services under a unified domain [15].

In Kubernetes management architectures, load balancers are often used to abstract multiple control-plane endpoints, simplifying user access and automation workflows. However, literature highlights that improper connection limits and timeout configurations can become bottlenecks under high CI/CD traffic loads [16]. Scaling connection thresholds and implementing proactive health monitoring are, therefore, essential for reliable operation.

2.6. Research Gap and Contribution

While existing research addresses Kubernetes scalability, GitOps workflows, and CI/CD automation independently, there is limited literature focusing on control-plane scalability for GitOps-based multi-cluster deployments exceeding 10,000 clusters. In particular, the combined use of multiple Fleet controllers, deterministic cluster-to-controller assignment, and HAProxy-based routing remains underexplored.

This paper contributes to the field by presenting a practical, production-scale architecture that integrates GitOps, CI/CD pipelines, load balancing, and automated cluster lifecycle management. The solution addresses real-world scalability constraints and provides empirical insights into performance tuning and operational best practices for large-scale Kubernetes environments.

3. Methodology

The proposed methodology adopts a cloud-native, GitOps-driven approach to address scalability and operational challenges associated with managing large-scale Kubernetes environments. Kubernetes was selected as the foundational orchestration platform due to its declarative model, extensibility, and widespread industry adoption [1], [2]. However, recognizing Kubernetes' inherent limitations in native multi-cluster management, the architecture leverages Fleet, a lightweight GitOps-based deployment engine, to enable centralized and consistent application delivery across thousands of downstream clusters [8].

To overcome scalability constraints associated with traditional Rancher-based Fleet deployments—where a single Rancher instance supports approximately 500 clusters—the system was architected using four independent Fleet controller clusters. Each Fleet controller was deployed on RKE2 and configured in a highly available topology with three nodes operating as control-plane, etcd, and master components. This design aligns with Kubernetes high-availability recommendations and ensures resilience against node or control-plane failures [2], [12]. Performance tuning was applied to critical control-plane components, particularly etcd, by increasing backend storage quotas, enabling periodic compaction, and adjusting request size limits. These configurations are consistent with established best practices for operating etcd at scale and were necessary to accommodate the metadata growth associated with managing thousands of GitOps-managed clusters [12], [13].

To provide a unified access layer across distributed Fleet controllers, HAProxy was introduced as a centralized routing and load-balancing component. HAProxy was selected due to its proven performance, low latency, and support for advanced routing and health-check mechanisms in distributed systems [14]. A path-based routing strategy was implemented, allowing incoming requests to be forwarded to the appropriate Fleet controller based on URL prefixes. This approach abstracts the complexity of multiple control-plane endpoints and aligns with established multi-tenant routing patterns in large-scale service-oriented architectures [15]. Connection limits and timeout parameters were carefully tuned to handle high CI/CD traffic volumes, as prior research indicates that inadequate load-balancer configuration can significantly degrade system performance under peak workloads [16].

Automated downstream cluster onboarding was achieved using SUSE Manager (MLM) as the central lifecycle and configuration management system. Upon provisioning a new cluster, MLM deploys the Fleet agent using Salt-based automation, ensuring consistency and eliminating manual intervention. Each cluster is assigned a unique five-digit hostname, which is processed using a modulo-based deterministic algorithm to map the cluster to one of the four Fleet controllers. Deterministic assignment strategies have been shown to improve scalability and reduce operational complexity in distributed systems by ensuring predictable resource distribution [16]. This methodology ensures balanced utilization of Fleet controllers while maintaining a fully automated registration workflow.

The CI/CD pipeline was designed around GitOps principles, with Git repositories serving as the single source of truth for application definitions and deployment configurations. Applications are packaged as Helm charts, enabling parameterized and reusable deployments across heterogeneous cluster environments [11]. Changes introduced by developers trigger automated CI workflows that enforce approval gates before deployment, aligning with DevOps governance and compliance requirements [9]. Once approved, updated configurations are registered with all Fleet controllers, allowing Fleet agents to continuously reconcile desired and actual cluster states. Label-based deployment strategies were employed to enable targeted rollouts, reducing deployment risk and supporting phased release patterns such as alpha and beta environments, which are widely recommended in modern microservices deployment practices [10].

Overall, the methodology integrates GitOps, infrastructure automation, deterministic control-plane scaling, and centralized routing to create a resilient and scalable CI/CD architecture. By combining established best practices from Kubernetes operations, distributed systems, and DevOps research, the proposed approach ensures reliable deployment management across more than 10,000 Kubernetes clusters while maintaining operational efficiency and system stability.

4. Results

The implementation of the proposed multi-Fleet controller CI/CD architecture demonstrated significant improvements in scalability, reliability, and deployment efficiency when managing large-scale Kubernetes environments. The system successfully supported more than 10,000 downstream clusters by distributing management responsibilities evenly across four independent Fleet controllers, each handling approximately 2,500 clusters. This horizontal scaling approach effectively mitigated the control-plane bottlenecks commonly observed in monolithic Kubernetes management architectures, confirming prior research that advocates decentralized control-plane designs for large distributed systems [1], [4].

Performance metrics collected during peak deployment windows indicated stable Fleet controller operation following etcd configuration tuning. Increasing the etcd backend quota from the default 2 GB to 8 GB eliminated quota exhaustion events that previously resulted in degraded API responsiveness and intermittent deployment failures. This outcome aligns with existing studies highlighting etcd storage limits as a critical constraint in large-scale Kubernetes deployments [12], [13]. Additionally, enabling periodic compaction and adjusting request size limits reduced metadata fragmentation and improved overall control-plane stability.

The introduction of HAProxy as a centralized routing layer produced measurable gains in availability and throughput. After increasing the maximum concurrent connection limit from 3,000 to 10,000, the system exhibited a substantial reduction in request timeouts and deployment delays during high CI/CD activity. Path-based routing enabled consistent and predictable access to Fleet controllers through a unified domain, simplifying both automation workflows and operational troubleshooting. These results corroborate prior findings that properly tuned load balancers play a pivotal role in maintaining performance and reliability in distributed service architectures [14], [16].

From an operational perspective, automated fleet agent registration via SUSE Manager significantly reduced cluster onboarding time. Newly provisioned clusters were registered and managed within minutes without manual intervention, leading to improved deployment consistency and reduced configuration errors. Furthermore, the label-based deployment strategy enabled controlled rollouts to targeted cluster groups, minimizing blast radius during updates and supporting phased release models. Overall, the results demonstrate that the proposed architecture effectively addresses scalability, performance, and operational challenges inherent in enterprise-scale Kubernetes CI/CD systems.

5. Discussion

The results of this study underscore the importance of architectural decomposition and automation in managing large-scale Kubernetes environments. By separating cluster management responsibilities across multiple Fleet controllers, the system avoids the scalability ceilings associated with centralized control planes, a limitation frequently cited in Kubernetes operations literature [3], [4]. This approach not only enhances system resilience but also provides a clear operational boundary for scaling as infrastructure demands grow.

The findings further reinforce the critical role of control-plane tuning, particularly with respect to etcd performance. While etcd is designed for consistency and reliability, its default configuration is insufficient for environments managing thousands of clusters and GitOps-driven metadata updates. The observed stability improvements following quota expansion and compaction tuning validate existing recommendations for proactive etcd capacity planning in large-scale deployments [12], [13]. These insights highlight that GitOps scalability is not solely dependent on tooling but also on underlying datastore optimization.

HAProxy's effectiveness as a routing and abstraction layer demonstrates how traditional networking components remain highly relevant in modern cloud-native architectures. The path-based routing model not only simplified user interaction with the system but also enabled seamless expansion of Fleet controllers without impacting existing workflows. However, the initial connection bottlenecks emphasize the necessity of continuous performance monitoring and iterative tuning, particularly as CI/CD traffic patterns evolve [14], [16].

While the deterministic modulo-based cluster-to-controller assignment strategy proved effective in ensuring balanced controller utilization, it introduces a degree of rigidity that may limit dynamic rebalancing in highly elastic environments. Future enhancements could explore adaptive routing mechanisms or controller autoscaling to further improve flexibility. Nevertheless, the overall architecture demonstrates that combining GitOps principles, deterministic automation, and centralized routing provides a robust foundation for scalable CI/CD operations.

7. Conclusion

This paper presented a scalable and production-proven CI/CD architecture designed to address the challenges of managing large-scale Kubernetes environments. By integrating multiple standalone Fleet controllers with a centralized HAProxy routing layer, the proposed solution effectively overcomes the scalability limitations inherent in traditional single-controller or Rancher-centric deployment models. The adoption of GitOps principles, combined with deterministic cluster registration and label-based deployment strategies, enabled consistent, auditable, and automated application delivery across more than 10,000 downstream Kubernetes clusters.

The experimental results demonstrate that horizontal scaling of Fleet controllers significantly improves control-plane stability and operational reliability. Proactive tuning of etcd parameters, including backend quota expansion and compaction policies, proved essential in supporting large volumes of GitOps metadata and maintaining API responsiveness under sustained CI/CD workloads. Similarly, optimizing HAProxy connection limits and routing rules eliminated traffic bottlenecks and ensured seamless access to distributed control planes through a unified domain. These findings reinforce the importance of infrastructure-aware design and continuous performance optimization in enterprise-scale DevOps environments.

Beyond technical performance, the proposed architecture delivers substantial operational benefits. Automated fleet agent registration using SUSE Manager removed manual onboarding steps, reduced configuration drift, and accelerated cluster lifecycle management. The CI/CD pipeline's label-driven deployment model enabled controlled, phased rollouts that minimized deployment risk while improving release agility. Together, these capabilities enhance governance, observability, and maintainability—key requirements for mission-critical systems operating at scale.

While the deterministic cluster-to-controller assignment strategy provided predictable and balanced scaling, future work may explore adaptive routing, controller autoscaling, and deeper integration with observability platforms to further improve elasticity and resilience. Nonetheless, this study demonstrates that combining GitOps-based deployment management, distributed control-plane architectures, and centralized routing mechanisms offers a robust and extensible foundation for large-scale Kubernetes CI/CD systems.

The proposed architecture serves as a practical reference model for enterprises seeking to modernize legacy systems and operate Kubernetes at scale. By aligning cloud-native tooling with proven distributed-systems principles, the solution achieves high scalability, reliability, and operational efficiency, positioning it as a viable blueprint for next-generation CI/CD infrastructures.

Conflicts of Interest

The author declares that there is no conflict of interest concerning the publication of this paper.

References

[1] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," Communications of the ACM, vol. 59, no. 5, pp. 50–57, 2016.
[2] C. Pahl, "Containerization and the PaaS cloud," IEEE Cloud Computing, vol. 2, no. 3, pp. 24–31, 2015.
[3] Rancher Labs, "Fleet: GitOps at Scale," 2024. [Online]. Available: https://fleet.rancher.io
[4] K. Hightower, B. Burns, and J. Beda, Kubernetes: Up and Running, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2019.
[5] HAProxy Technologies, "HAProxy Documentation," 2024. [Online]. Available: https://www.haproxy.org
[6] N. Forsgren, J. Humble, and G. Kim, Accelerate: The Science of Lean Software and DevOps, Portland, OR, USA: IT Revolution Press, 2018.
[7] A. Sharma and D. Spinellis, "Evaluating GitOps for large-scale cloud-native systems," IEEE Software, vol. 39, no. 4, pp. 52–60, 2022.
[8] Rancher Labs, "Fleet: GitOps at Scale," 2024. [Online]. Available: https://fleet.rancher.io
[9] N. Forsgren, J. Humble, and G. Kim, Accelerate: The Science of Lean Software and DevOps, Portland, OR, USA: IT Revolution Press, 2018.
[10] S. Newman, Building Microservices, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2021.
[11] M. Hausenblas and S. Schimanski, Programming Kubernetes, Sebastopol, CA, USA: O'Reilly Media, 2019.

[12] Kubernetes Authors, “Operating etcd clusters for Kubernetes,” 2024. [Online]. Available: https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/
[13] J. Turnbull, The Kubernetes Book, 2023 ed., Amazon Web Services, 2023.
[14] HAProxy Technologies, “HAProxy Documentation,” 2024. [Online]. Available: https://www.haproxy.org
[15] T. Erl, Service-Oriented Architecture: Concepts, Technology, and Design, Upper Saddle River, NJ, USA: Prentice Hall, 2005.
[16] M. Kleppmann, Designing Data-Intensive Applications, Sebastopol, CA, USA: O’Reilly Media, 2017.