Article

Vendor-Agnostic Reconfiguration of Kubernetes Clusters in Cloud Federations

Eddy Truyen[1,‡]*, Hongjie Xie [2,‡] and Wouter Joosen[1,]

[1] imec-DistriNet, KU Leuven; firstname.lastname@kuleuven.be
[2] Dept. of Computer Science, KU Leuven; xiehj0731@gmail.com
* Correspondence: Eddy.Truyen@kuleuven.be; Tel.:+32 16 37 35 85
‡ These authors contributed equally to this work.

Abstract: Kubernetes (K8s) defines standardized APIs for container-based cluster orchestration so it becomes possible for application managers to deploy their applications in a unified manner across different cloud providers. A practical problem is however feature incompatibility between different K8s vendors, who offer commercial K8s products based on the open-source K8s distribution. A large number of documented features in this open-source distribution are optional features that are turned off by default, but can be activated by setting specific combinations of parameters and plug-in components in configuration manifests for the K8s control plane and worker node agents. However, none of these configuration manifests are standardized, giving K8s vendors the freedom to hide the manifests behind a single, more restricted, and proprietary customization interface. Therefore some optional K8s features cannot be activated consistently across K8s vendors and applications that require these features cannot be run on those vendors. In this paper we present a unified, vendor-agnostic feature management approach that bypasses the proprietary customization interface of K8s vendors in order to consistently activate optional K8s features across a federation of clusters hosted by different Kubernetes vendors. We describe vendor-agnostic reconfiguration tactics that are already applied in industry and cover a wide range of optional K8s features. Based on these tactics, we design and implement an autonomic controller for declarative feature compatibility management across a cluster federation. We found that the features configured through our vendor-agnostic approach have no impact on application performance when compared with a cluster where the features are configured using the configuration manifests of the open-source K8s distribution. Moreover, the maximum time to complete reconfiguration of a single feature is within 100 seconds, which is 6 times faster than using proprietary customization interfaces of mainstream K8s vendors such as Google Kubernetes Engine. However, there is a non-negligible disruption to running applications when performing the reconfiguration; this disruption impact does not appear using the proprietary customization methods of the K8s vendors. Therefore, our approach is best applied in the following three use cases: (i) when starting up new K8s clusters, (ii) when optional K8s features of existing clusters must be activated as quickly as possibly and temporary disruption to running applications can be tolerated or (iii) when proprietary customization interfaces do not allow to activate the desired optional feature.

Keywords: Kubernetes; Cluster federation; Feature-oriented configuration management, Vendor lock-in

1. Introduction

With the rapid evolution of Internet applications' business and technical requirements, the cloud computing paradigm is accommodating more and more new concepts. Cloud federation, which interconnects multiple cloud computing environments, is a fast evolving technology in the cloud computing eco-system. It aims to support the following application deployment scenarios[1]:

- Cloud bursting facilitates an application scenario where a portion of workloads is dynamically migrated from an on-premise private cloud to public cloud platforms to cope with peaks and spikes in workload while reducing capital expenditures;

- High availability and geo-distribution requires replicating an application across different availability zones and different regions, respectively. The latter is necessary to offer a the same quality of service level to end-users across the world;
- Security policy and regulation compliance requires that sensitive workloads and data are deployed on-premise while insensitive traffic and workloads can be handled by elastic public clouds.

However, the heterogeneity of the underlying cloud infrastructure like virtual networks, computing instances, and proprietary services of IaaS providers poses a great challenge to the scalability, transferability and interoperability of cloud applications across different cloud platforms. This problem is called vendor lock-in[2]. At present, there are many methods and architectures that tackle such challenges like the Broker Model[2] or Volunteer Federations of providers[3].

The recent trend of using the standardized container orchestration platform Kubernetes (K8s) introduces less complexity to avoid vendor lock-in because Kubernetes hides most heterogeneity of the underlying infrastructure and provide a unifying computing platform[4][5]. More specifically, a K8s cluster can run on different IaaS providers, and with the help of the K8s control plane, the cluster itself can achieve automatic scaling, container management, and other functionality across multiple cloud providers.

Although a single K8s cluster has good support for deployment across different availability zones within a single cloud provider, a single cluster has bottlenecks in scalability [6] and cannot be deployed across multiple geo-regions or different cloud providers. Therefore, cluster federation solutions, which interconnect multiple Kubernetes clusters on different cloud regions or cloud providers, has the natural advantage of improving cluster performance and scalability and can further facilitate geo-distribution as well as management of ultra-large systems.

However, member clusters in a cluster federation need to be configured in the same manner in order to preserve (i) *application behavior consistency*, (ii) *application management consistency* and (iii) *cluster management consistency*: (1) Applications deployed in a cluster federation have to perform consistently on each member cluster to meet commercial and technical requirements and to comply with security policies defined by governance solutions. For example, in order to achieve the same user experience in different regions, an application needs to have the same performance level and use the same security standards for data plane security on each member cluster; (2) Management and deployment of applications must also be performed consistently in order to simplify the process of application management. For instance, only if each member cluster supports exactly the same APIs, application administrators can manage their applications uniformly in a federation setting; this is especially a problem when APIs have been deprecated but they are still highly demanded by using community, e.g. the PodPreset API in the OpenShift community[7]; (3) Management consistency is also needed at the level of the control plane of each cluster. For example, from a security perspective, each cluster needs to have the same security guarantees; e.g., the etcd databases running in the control plane of each cluster may be required to encrypt data at rest. In order to meet these three consistency requirements, it is important that the K8s control plane and K8s agents are configured in the same way across different member clusters.

However, if member clusters belong to different K8s vendors, these three consistency requirements cannot always be met due to differences in default cluster configurations and, more importantly, the *inability to change cluster configurations*. The latter is because K8s vendors may hide configuration manifests that are defined as part of the open-source K8s distribution on GitHub. Such hiding either involves locking configuration parameters and plug-in components in a particular setting or encapsulating the configuration manifests behind a proprietary customization API. Such proprietary API only offers reconfiguration of some configuration settings and increases the complexity of vendor lock-in avoidance.

In this paper, we present a unified and vendor-agnostic cluster reconfiguration approach, so that cluster administrators and application managers can bypass vendor lock-in limitations when reconfiguring clusters and can also change a broader set of configuration settings than offered in the proprietary customization interfaces. In particular, we make the following contributions:

- We propose a feature-oriented approach where K8s clusters can be reconfigured by means of declarative configuration of desired optional features. An optional feature corresponds here to a specific functionality of Kubernetes that is not enabled by default, yet it is clearly described in the open-source documentation of Kubernetes with precise instructions how to enable it. Based on a previous case study of feature incompatibilities between three leading K8s vendors – Azure Kubernetes Service (AKS), Elastic Kubernetes Service (EKS), Google Kubernetes Engine (GKE) – we have identified more than 30 optional features that were stable or highly demanded, but that were locked by at least two vendors in different enabled or disabled states, leading to feature incompatibilities that violate at least one of the aforementioned consistency requirements;
- We account all feature incompatibilities to 3 configuration manifests of the open-source K8s distribution that are partially or completely hidden by proprietary customization interfaces of the three vendors;
- We describe in detail what are the most prevailing vendor-agnostic reconfiguration tactics in industry for bypassing proprietary customization interfaces and changing a broader set of configuration settings. We point out that these tactics are all based on imperative configuration management that suggests one-off installation without further monitoring. This is not in line with the Kubernetes philosophy of declarative configuration management where a separate control loop continuously monitors for differences between desired and actual system configuration states;
- We extend KubeFed, a popular tool for federation of K8s clusters, with an API and autonomic controller for declarative feature compatibility management. As such, cluster administrators and application managers can submit feature configuration manifests to this API to specify what desired features all member clusters in a federation should have. The controller detects missing features in the member clusters. If a cluster does not support one or more desired features, the controller will apply the aforementioned imperative reconfigurations tactics to install them. It will further monitor the member cluster and generate events to report successful or pending installation of the desired features;
- We make an empirical evaluation of the controller and the three vendor-agnostic reconfiguration tactics with respect to (i)the impact of the reconfigured K8s features on the performance of applications, (ii) the disruption of running applications during the reconfiguration process and (iii) the total reconfiguration time.

The remainder of the paper is organized as follows. Section 2 introduces the technical background, mainly focused on Kubernetes and cluster federations. Subsequently, Section 3 investigates related work for managing vendor lock-in avoidance in both cloud computing and container-based clusters. Then, Section 4 makes an analysis of the configuration manifests that cause the overall feature incompatibility problem. The vendor-agnostic reconfiguration tactics for these problematic configuration manifests are elaborated in Section 5. Thereafter, Section 6 covers the design and implementation the autonomic controller that reconfigures cluster features in a federation. Subsequently, Section 7 presents the evaluation of the autonomic co controller and reconfiguration tactics using multiple applications on top of the Google Kubernetes Engine and Kubeadm, a popular Kubernetes installer for on-premise deployments. Finally, Section 8 sets out our conclusions and future research directions.

2. Background

This section will introduce the technical background related to this paper, structured as follow. Section 2.1 reviews the components and architecture of Kubernetes so that readers can have an overview of the functionality and operating mechanism of Kubernetes. Section

2.2 will introduce the architecture and techniques of Kubernetes cluster federations, which is the context where our autonomic controller runs.

2.1. Basics of Kubernetes

Modern distributed cloud computing applications usually need to deploy and manage a large number of microservices encapsulated in containers to achieve the elasticity and scalability of applications. At the same time, microservices and containerization can also facilitate flexibility and efficiency of application development by utilizing more modular designs. To this end, it becomes necessary to use container orchestration systems like Mesos [8] and Kubernetes [5] to efficiently manage large container-based applications. Such systems abstract away and automate complicated container orchestration tasks, including deployment, auto-scaling, resource allocation, etc. [9], which bring advantages of fast application delivery, and reduced operational and resource cost without reducing the quality of the application deployment process.

2.1.1. Container

Before introducing Kubernetes, it is necessary to introduce containers and why containers are widely used in cloud computing.

Containerization is operating system-level virtualization. Unlike the hypervisor, which creates virtual hardware resources, a container only creates a separate isolated space for processes. In Linux, this isolation space is achieved through namespaces [10] and cgroups(Control Groups)[11]. Namespace provides running environment isolation for processes, including network, file system, PID space, etc. Cgroup provides isolation and limits on resources required by processes, including CPU, Memory, Disk I/O, and network usage.

Compared to hardware virtualization, containers are much more lightweight and can package applications and required libaries in a self-container component. Therefore, they can be quickly ported and deployed in different computing environments and provide elasticity needed by cloud applications[12]. As containers run in isolated namespaces, the process in one container typically can't interfere with or monitor processes in other containers or the host OS, providing proper use of Linux-based security access controls such as AppArmor and Seccomp.

2.1.2. Kubernetes Architecture

Kubernetes (K8s), based on Google's internal large-scale container management tool Borg[13], has become the mainstream container orchestration tool in the industry. Kubernetes adopts a declarative configuration management approach[14], where built-in or extended controllers monitor and adjust the actual state of the cluster to a set of desired states specified by the user. To understand the internal operation principle of Kubernetes and better introduce the mechanism of tactics and controllers as explained in Section 5, we first introduce the Master-Slave architecture of Kubernetes (Figure 1) and the functionality of each component.

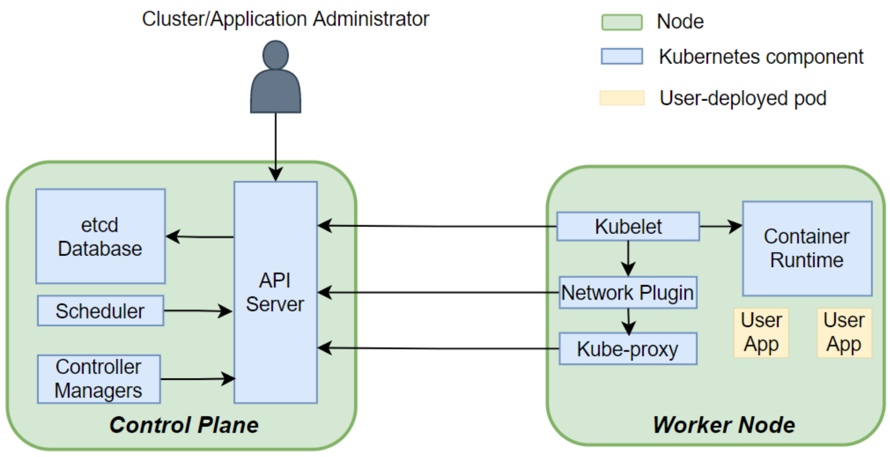


Figure 1. Kubernetes cluster architecture.

A Kubernetes cluster consists of two kinds of nodes: Control Plane and Worker Nodes[15]. The Control Plane node is the brain of the Kubernetes cluster and is responsible for coordinating the workload and resources of the entire cluster. On Control Plane, we usually run four core components: API server, Scheduler, Controller Managers, and etcd database:

- **API server** is the central point of communication among the components of the cluster. It exposes various RESTful APIs, through which worker node agents, controllers, users and applications can create, query or update cluster and application resources. These APIs are an abstraction of the actual resources deployed. Here we introduce the APIs resources that are used in this paper:
  - **Pod** is the atomic unit of deployment in Kubernetes, consisting of one or more tightly coupled containers. A pod can be thought of as a virtual host for containers, and all containers in it share the same Linux network namespace and cgroup parent;
  - **Deployment** represents a set of pod replicas managed by the Deployment controller. We can specify the desired number of replicas and the updating strategy in the Deployment API. The Deployment controller will enforce our specification;
  - **Daemonset** represents a pod that should be deployed on every node of the cluster, and every node should only have one copy of the pod;
  - **Service** is an API resource that specifies a stable network access point behind a set of volatile pods;
  - **Custom Resource Definition (CRD)**[16] represents extended API resources in the API server. We can use this API to introduce new custom APIs;
- **etcd Database** is a key-value database that stores the desired and actual states of all API resource objects. After the API server receives the client's request, it will query or update the corresponding resource states in the etcd database[17];
- **Controller Manager**[18] contains many built-in controllers that implement control loops to manage resources of various built-in API types like Deployments and Services. For example, the Deployment Controller monitors the actual number of replicas of pods in a Deployment and performs actions to make it match the desired number as described by users;
- **Scheduler** is responsible for placing pods on the appropriate worker nodes based on the node states and the requirements of the pods.

Worker nodes are physical or virtual machines where actual workloads run. They are registered in the cluster and thus managed by the control plane. A worker node comprises four main components: Kubelet, Kube-Proxy, Container Runtime, and Network Plugin:

- **Container Runtime** is responsible for container image pulling/pushing to and from a central container registry as well as the creation, execution and resource monitoring of containers;
- **Network Plugin** is responsible for creating a virtual network bridge on each node of the cluster and configuring routing rules on each node to manage the connectivity between containers;
- **Kube-Proxy** is a cluster-wide load balancer that exposes a pool of pods to external clients via a stable Service IP, which is created via a Service API object. It watches the Services and associate Endpoint resource objects on the API server and maintains network and routing rules on the nodes to implement customizable (e.g. using session affinity) round-robin load balancing. The Kube-Proxy runs as a pod in worker nodes;
- **Kubelet** is the essential component on the worker node. It watches the pod resource objects on the API server to detect changes about Pods on its node. It interacts with the container runtime, the network plugin, and other add-ons to ensure that the container running state is consistent with the specification of the pod manifest. For example, if the Kubelet sees that the pods to be created have specific resource limits, it will firstly interact with the container runtime to create a pod-level network namespace with a particular cgroup parent setting;

2.2. Kubernetes Cluster Federation

The native management capabilities of Kubernetes are still at a single-cluster level. Different Kubernetes clusters are independent and do not have a direct connection. A cluster federation can interconnect and package these separate clusters to appear as a single logical cluster to top-level users. In Kubernetes, we have many frameworks to enable cluster federations. The official implementation of multi-cluster management platform evolved from KubeFed V1 to KubeFed V2[19]. Moreover, the recent Open Cluster Management (OCM) framework[20], which delegates the management overhead part to the managed clusters[21], reduces the performance overhead of the host cluster. All of these platforms use a similar Master-Slave architecture. The multi-cluster management framework that this paper is based on is KubeFed V2, maintained and developed by the SIG-Multicluster team. KubeFed provides the essential building block for more complex use cases of federated clusters. KubeFed V2 uses the Custom Resource Definition (CRD) API in combination with the Operator pattern to improve the flexibility and extensibility of KubeFed v1[19]. Figure 2 presents the architecture of the KubeFed cluster federation.

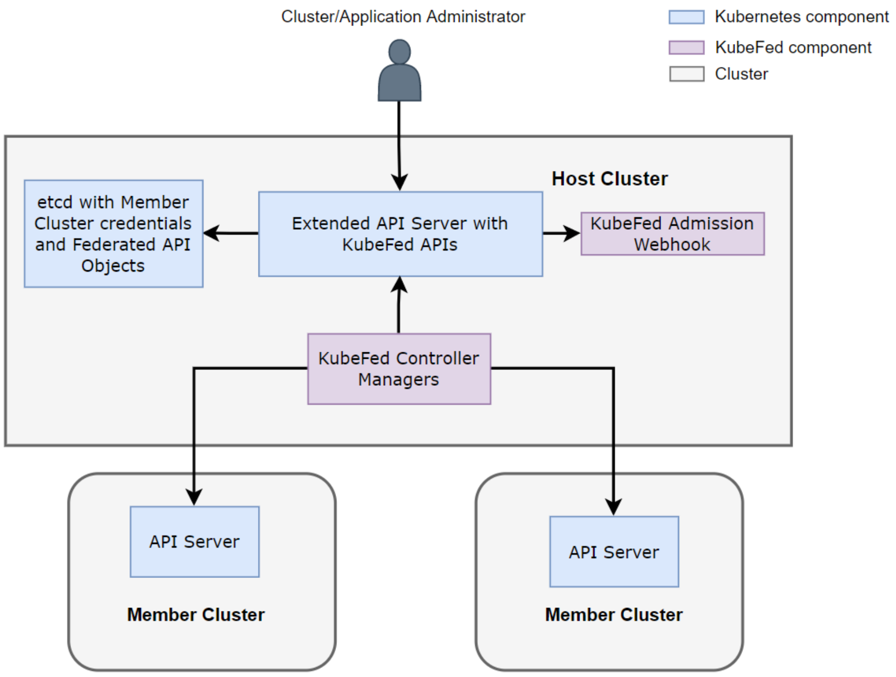

Cluster/Application Administrator
Kubernetes component
KubeFed component
Cluster
Host Cluster
etcd with Member Cluster credentials and Federated API Objects
Extended API Server with KubeFed APIs
KubeFed Admission Webhook
KubeFed Controller Managers
API Server
Member Cluster
API Server
Member Cluster

Figure 2. Cluster Federation based on KubeFed V2.

Using KubeFed, a cluster federation consists of two types of clusters: Host cluster and Member Cluster:

- **Host Cluster** is a Kubernetes cluster where the KubeFed Control Plane resides. It extends the API server with KubeFed APIs and deploys KubeFed controller managers. KubeFed controllers can access the credentials of managed member clusters and communicate with their API servers. Users create and manage federated resources through the KubeFed API, and KubeFed controllers propagate the changes of federated resources to the corresponding member clusters according to the specification provided by the user. The KubeFed admission webhook is responsible for validating the federated resources, such as when we want to create a federated custom resource object, the webhook will only allow it if all member cluster support this custom resource. In addition, a host cluster can also become a member cluster at the same time;
- **Member Clusters** are the place where workloads and resources are actually deployed in the cluster federation. It is no different from normal clusters. It does not know anything about the other member clusters and the presence of the host cluster. Kubefed Controllers in the host cluster are like any other ordinary clients of the member cluster's API server;

KubeFed is not the silver bullet that enables all cluster federation use cases. Kubefed itself considers only the propagation and placement of resources for member clusters[22]. For example, suppose we create a FederatedDeployment resource in the Host Cluster. In that case, KubeFed controllers will create regular deployment resources in the corresponding member clusters according to the configuration of member clusters and federated resource requirements[23]. To implement more complex application scenarios, we must integrate external services and middleware. For example, we use an external DNS service to achieve cross-cluster service access and discovery as this is not available in Kubefed[23]. From the perspective of Kubefed itself, we only have one host cluster making it a single point of failure and performance bottleneck. This problem gave birth research on decentralized host clusters[24].

3. Related work

We distinguish between two broad classes of related work: Section 3.1 discusses state-of-the-art techniques for addressing the transferability of cloud applications across multiple vendors. Then, Section 3.2 introduces current industry approaches to reconfiguration of Kubernetes clusters.

3.1. *Transferability of cloud-native applications in cloud federations*

Transferability is an important prerequisite to ensure elastic deployment of cloud-native applications across different cloud infrastructures and enable cloud federations [34, 35]. When realizing the transferability of cloud-native applications, heterogeneous cloud infrastructure imposes problems of vendor lock-in. That is to say, without Kubernetes, the realization of cloud-native elasticity depends on the APIs the cloud provider. Before the release of Kubernetes, there were several studies aimed at addressing achieving the transferability of cloud-native applications. Buyya et al. [2,34] proposed a broker model where brokers sit among different cloud providers, aggregating and coordinating the services they provide, and exposing provider-independent interfaces to applications. After the release of Kubernetes, Kratzke et al. [4,36,37] are the first to recognize the power of container orchestration platforms for achieving vendor lock-in avoidance with cloud providers. First they argued that the broker-based approach requires that application development artifacts may need to be modified to use the broker interfaces, yielding again a form of vendor lock-in with the broker. This problem of modification of application development artifacts does not occur with container-based platforms as containers offer a portable application component representation. Secondly, they propose an approach that incorporates nodes from multiple cloud providers into one cluster powered by a container orchestration platform. Therefore, applications running on this container orchestration platform can be migrated painlessly regardless of which cloud provider they are on. However, this solution against vendor lock-in does not work for the case where cloud providers are K8s vendors themselves who offer hosted K8s clusters. This yields vendor lock-in due to the feature incompatibilities between different K8s vendors as explained in Section 2.

3.2. *Current feature reconfiguration approaches for K8s in industry*

There are plenty of feature reconfiguration approaches in the industry. Some of them are proprietary to cluster vendors. For example, when building a GKE cluster, GKE provides many customization options for the cluster. As stated before, such proprietary reconfiguration approaches are incompatible with other vendors. On the other hand, there are vendor agnostic reconfiguration tactics that are able to bypass the proprietary customization interfaces when for installing specific plugin components or replacing deprecated, yet popular APIs. For example, the command-line tool of Cilium can install the CNI plugin cilium in almost every Kubernetes vendor[38]. Another example is when the PodPreset API was deprecated, RedHat re-developed this API by means of extending the Kubernetes API with a custom resource definition (CRD) that mimics the PodPreset API object and an admission webhook around the API server that validates and processes requests to create or update PodPreset resources. These reconfiguration methods appear imperative however and therefore are not in line with the declarative configuration management approach of Kubernetes that comes with a control loop that continuously monitors for differences between desired and actual cluster state. The lack of such declarative configuration management approach puts the burden of the monitoring on the shoulders of the cluster administrator, therefore reducing her productivity especially when multiple reconfiguration have been performed in an imperative way. In opposition, this paper presents a fully declarative configuration management approach. In line with the Operator pattern [39], the declarative configuration management approach is implemented by extending the Kubernetes API with a set of custom resource definitions to declare a desired K8s feature, an admission web hook to validate feature reconfiguration requests and a controller that continuously bring the actual state of the member clusters in line with the declarative configuration of desired features.

The aforementioned OCM cluster federation framework[20] has more in common with our approach. It supports a rich policy language for ensuring compliance with various kinds of policies and cluster configurations uniformly across multiple Kubernetes vendors[40]. The framework is also used within hybrid cloud providers such as RedHat OpenShift. At its core, OCM applies the Operator pattern as our approach does but in a decentralized manner where each member cluster takes care of its own policy enforcement and reconfiguration. A more distinct difference is that OCM aims to extend Kubernetes with new functionality and properties that build on top of the Kubernetes API whereas our work studies the ieasibility of the Operator pattern for reconfiguring the internals of Kubernetes.

4. Analysis of the Feature Compatibility Problem

When using cluster federation, software companies often combine a central on-premise cluster with additional clusters from one or more hosted Kubernetes vendors such as EKS [25], AKS [26], and GKE [27]. The clusters provided by these vendors are hosted product types, which free the software companies from a tedious installation and configuration process that is typically the case with installer or distribution K8s products. Moreover, hosted K8s products are the preferred choice for running application in production environments as these hosted K8s cluster products are security hardened, come with service-level agreements (e.g for availability) and extensive certification, and provide support for governance and policy compliance.

The open-source distribution of K8s is highly customizable through various configuration manifests that are composed of configuration files, plug-in components and libraries. However, these manifests are not part of the standardized RESTful APIs of K8s. Therefore vendors can decide to hide these manifests from the customer and lock them in a particular configuration setting. A specific problem with vendors of the hosted product type is that these vendors hide the all components of the K8s control plane; as such a cluster administrator cannot modify the configuration manifest of the control plane components such as the API server. Finally, as the RESTful APIs are organized in different API groups, not all of these API groups must be supported by a K8s vendor in order to be certified by the cloud-native computing foundation.

This brings great challenges to achieving the aforementioned consistency requirements in cluster federations as explained in Section 1. In our previous work [28], we have found that 30 out of 162 documented features of the open-source distribution of K8s v1.13, are not consistently activated or de-activated in the aforementioned leading vendors of the hosted product type, and these feature settings cannot be modified by the cluster administrator due to hidden configuration manifests of the Kubernetes control plane.

The following subsections describe three types of configuration manifests that are causing the difficulties: (i) APIserver configuration, (ii) KubeletConfiguration and (iii) configuration of network plugins.

4.1. API server Configuration

Various plug-in components and configuration parameters can be set via options of the `kube-apiserver` command that starts up the API server [29]. These plug-in components and configuration parameters of the API server can be mapped to the following K8s features that are clearly defined as part of the open-source documentation[5][28]:

1. Admission controllers. These are modular interceptors that wrap the API server. Although these interceptors are defined as part of the open-source distribution of K8s, we have found in our previous work'[28] that 7 out of 28 admission controllers were not consistently set across the studied vendors for K8s version v1.13;
2. The RESTful APIs. The RESTful APIs of the API server are organized into different API groups that can be enabled or disabled. For example for K8s v1.13 we found that EKS does not support the `k8s.metrics.io` API group that is needed for autoscaling of containers. Another problem are the deprecated APIs that are still highly demanded. For example, the PodPreset API has been removed after K8s version v1.19. Hosted K8s products only offer the latest versions and therefore the PodPreset

API is not available anymore. Yet, there is still a high demand for this feature in the OpenShift community[7];

3. Feature gates for beta features. Each K8s version introduces new alpha features that can and should be disabled via feature gates when running clusters in production environments. These alpha features may disappear or be promoted to the stable stage in a successive K8s version, after which the feature gate is removed. However, between the alpha and stable stage, there is the beta stage and beta features are enabled by default in the open-source distribution. However they can be disabled by K8s vendors. In the latest version of K8s there are more than 40 beta features. In our previous work[28], we found differences between the three K8s vendors with respect to 2 beta feature gates. Unfortunately, also different alpha features were enabled;
4. Encryption of secrets stored in etcd is a feature to prevent attackers to read secrets from the etcd database in the clear.

Note, since the control plane and its components are completely hidden in hosted K8s products, it is not possible to access the configuration of the API server in anyway. The only tactic that works is the construction of a replica API and replica admission controller by using a CRD and a dynamic admission web hook that intercepts requests for creation, update and deletion of CRD objects. This tactic is explained in Section 5.1 and is illustrated by means of the PodPreset API.

Moreover, when the control plane is hidden by a K8s vendor, etcd encryption nor feature gates concerning the behavior of the control plane itself can be reconfigured in a vendor-agnostic manner. For those feature gates that concern the Kubelet on worker agents, the KubeletConfiguration manifest on all worker nodes can be changed as is described in the next section.

4.2. KubeletConfiguration Manifest

The Kubelet is the local agent of Kubernetes on every node of the cluster. It is responsible for integrating the container runtime and networking plugin. Moreover, it also configures the container runtime to enforce resource isolation for CPU, memory, ephemeral storage, as well as isolation of file system and network isolation between co-located containers. The `kubelet` command in the reference manual of Kubernetes has gradually evolved from accepting a long list of parameters to taking a single configuration manifest file that supports imperative configuration management. However, all vendors only allow setting some fields of the KubeletConfiguration manifest via a proprietary customization interface[30][31][32], yielding restricted customization without vendor-agnostic reconfiguration support. The affected K8s features include [5][28]:

1. Supported container runtimes. The main container runtimes are containerd and cri-o (docker has been deprecated). Various libraries and settings must be installed on the worker nodes themselves in order to make the selected container runtime work;
2. Supported authentication and authorization schemes of the Kubelet with respect to securing the Kubelet API and authenticating the Kubelet to the API server;
3. Various logging features such as container log rotation;
4. Container image garbage collection;
5. CPU management policies for reserving CPUs to specific Pods.

Section 5.2 presents the vendor-agnostic reconfiguration strategy for full access to the KubeletConfiguration manifest and illustrates it for the CPU management policy.

4.3. Configuration of Network Plugins

In Kubernetes, the container-level network is setup by an external network plugin that must conform with the CNI specification. However, in GKE or AKS, the Kubernetes' initial networking solution, called `kubenet` [33] is used as default, which only considers container connectivity within a node and relies on the cloud provider's infrastructure for cross-node routing. However, to use more advanced network features such as fast network policy enforcement via eBPF, CNI-based network plugins are required. In GKE or AKS, however, CNI plugins can only be activated through proprietary higher-level customization interfaces. Moreover in none of the vendors it is possible to choose from the wide range of existing CNI plugins. Instead, only one vendor-specific CNI plugin can be activated. The following K8s features are affected:

1. External Source Network Address Translation (sNAT) for Pod IPs. IP addresses of Pods cannot be routed to from outside of the cluster. With sNAT enabled, it becomes possible to connect to Pods from outside the cluster via stable external IP addresses;
2. Network policies. The K8s networking model requires that every worker node and pod must be able to connect to any other Pod IP address. Network policies allows expressing distributed firewall rules across the cluster to properly segment different applications from each other. This feature is only supported by some CNI plugins and there exists various implementations that differ in performance overhead and types of network policies;
3. Multiple network interfaces per Pod. This is only supported by the Multus CNI plugin;
4. Re-implementation of the kube-proxy with more efficient load balancing at the level of the Linux kernel and reduction of the number of hops across nodes when forwarding requests to, and returning responses from pods;
5. Encryption of control plane and data plane messages.

Note that when the control plane is hidden, the CNI plugin can only be changed at the worker nodes and not in the control plane. Although most control plane communication occurs via the node network instead of the Pod-level network, some functionalities such as probing of liveness and readiness of Pods requires controllers in the control plane to interact via the Pod network. Section 5.3 presents the vendor-agnostic reconfiguration tactic to replace network plugins while ensuring Pod connectivity with the control plane.

5. Reconfiguration Tactics

This section introduces detailed reconfiguration tactics for the three problematic configuration manifests. We reuse and adapt existing solutions or propose new reconfiguration tactics for these three locked-in configuration manifests.

We consider the primitives of cloud-init scripts nor wrapping vendors' proprietary customization interfaces as valid vendor-agnostic tactics. Although cloud-init [41] is a vendor-agnostic technology supported by almost every cloud provider, the process of rebooting VMs and providing cloud-init scripts as user data depends on providers' API [42][43][44]. The approach of encapsulating proprietary interfaces is still vendor-dependent and cannot be extended to new providers.

5.1. API server configuration

Due to the extensibility of Kubernetes, we can extend the Kubernetes API server with new API resources through custom resource definitions (CRDs) or aggregated API servers. The main difference is that with CRDs, the API server is responsible for handling requests for newly introduced APIs, while with an aggregation server, the API server forwards requests for the new API to a user-implemented API server. Additionally, since K8s v1.16, dynamic admission webhook [45] has been a stable feature in Kubernetes to create custom object admission logic for API servers. When the API server receives a resource object, it will call back the relevant webhook servers deployed externally to the control plane. The mutating webhook server can modify the resource object, and the validating webhook server decides whether to allow the resource object into the cluster. These assets can be used to bring in missing API resources and built-in admission controllers

We illustrate the tactic for the PodPreset feature. As stated in Section 4.1, in order to meet the OpenShift community's demand for this feature, Red Hat Communities of Practice (Redhat-COP) has reactivated the API using a CRD and a mutating admission webhook server[7]. The new CRD defines the exact same API resource format as the original built-in

PodPreset. The mutating admission webhook server listens for newly created pods and matches them with PodPreset objects according to the pod's labels and the label selectors of PodPresets objects. Once matched with a PodPreset object, it injects the running information from the PodPreset object into the pod manifest. To register the webhook server with the API server, a MutatingWebhookConfiguration is created to instruct the API server to pass all newly created pods to the webhook server. Figure 3 describes the interaction between the API server and the webhook server. The webhook server is deployed in a worker node. Whenever the API-server receives a pod creation request, it will send the pod object in an AdmissionReview request to the webhook server. The webhook server injects information into the pod object and returns it in an AdmissionReview response. The returned pod object is used to complete the pod creation process in the API server. In this figure, we suppose there are no other admission steps and the API server sends the pod resource object directly to the scheduler.

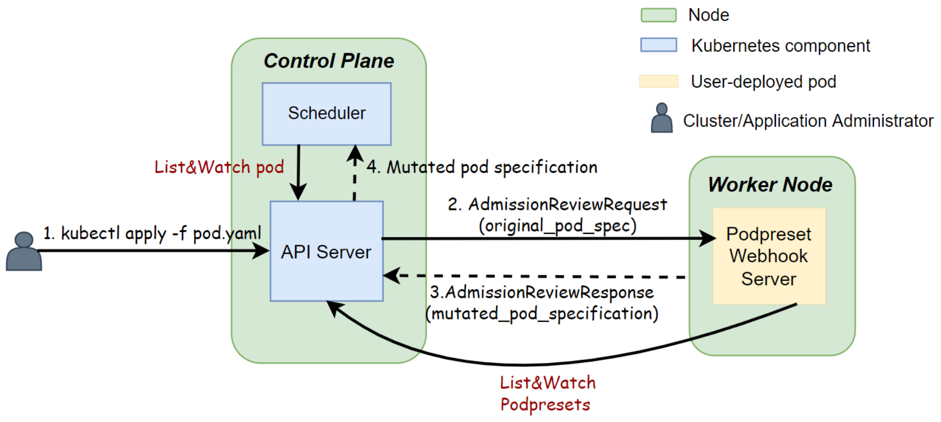

Node
Kubernetes component
User-deployed pod
Cluster/Application Administrator
Control Plane
Scheduler
List&Watch pod
4. Mutated pod specification
1. kubectl apply -f pod.yaml
API Server
2. AdmissionReviewRequest (original_pod_spec)
3.AdmissionReviewResponse (mutated_pod_specification)
Worker Node
Podpreset Webhook Server
List&Watch Podpresets

Figure 3. Redhat-COP PodPresets admission webhook server

5.2. *KubeletConfiguration manifest*

To reconfigure Kubelets with a changed configuration manifest in a vendor-agnostic manner, we take the approach of privileged Daemonsets [46] [47] [30]. A Daemonset is a workload resource that deploys a pod on every node in a cluster. As the name suggests, a Daemonset is typically used to deploy daemons on nodes, such as resource monitoring and logging tools. We will take advantage of this feature of Daemonset to deploy a privileged pod on each node. This pod can run in the same IPC and network Linux namespaces as the host machine by setting the `hostIPC` and `hostNetwork` fields to `true` in the pod manifest. Moreover, the pod can use `nsenter` to execute the scripts that are installed in a mounted host directory. In other words, the pod looks like any other normal process on the operating system, with access to the file system and host network stack, etc.

Therefore, we can let such privileged pods run a script that overrides the Kubelet configuration file in the host file system, modifying the desired configuration fields. For example, the aforementioned CPU Manager policy feature could be enabled by setting the `cpu-manager-policy` field to true. In addition, by setting the `kube-reserved` and `system-reserved` fields of the Kubelet configuration file, we reserve a certain amount of CPU resources for Kubernetes and system components such as Kubelet, container runtime, etc. to prevent node crashes when pods with dedicated cores take all compute resources. Next, the script deletes all Kubelet state files and restarts the Kubelet service with `systemctl`. After the Kubelet restarts, it will load the modified configuration file, thus having the desired configurations..

It's worth noting that Kubelet restarts do not modify the cgroup settings of existing running Pods, so the changed configuration settings related to reserving Kubernetes and system resources will not be applied to them. We have to restart these pods if we want them to be subject to the new configurations. Existing pods will also not stop running during a restart of the Kubelet. But if pods define a readiness or liveness probe[48], these pods might be unready from the clients' point of view, and kube-proxy would not forward traffic to them because the Kubelet is stopped, and no probing can be performed. The Kubelet restart process is fast however, and pods that are not ready can usually be restored to the Ready state in less than ten seconds, so pods will not be migrated to another node during this time.

5.3. CNI Network Plugin

Using the CNI networking plugin, we can bring more sophisticated networking capabilities to our cluster, such as network policies to customize connectivity of pods. However, switching to CNI plugins in clusters of a hosted product type depends on the vendors' proprietary interfaces. At the time of research, Calico was the leading CNI plugin to support network policies and was also employed by all three studied vendors. Therefore this section takes GKE and the Calico network plugin as the example to introduce our vendor-agnostic reconfiguration tactics, of which Figure 4 illustrates the general steps.

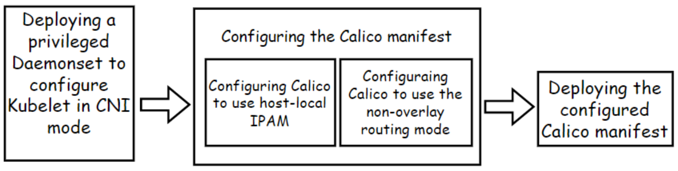

Deploying a privileged Daemonset to configure Kubelet in CNI mode
Configuring the Calico manifest
Configuring Calico to use host-local IPAM
Configuraing Calico to use the non-overlay routing mode
Deploying the configured Calico manifest

Figure 4. Steps for reconfiguring the Calico CNI plugin in GKE

Firstly, similar to the tactics for reconfiguring the Kubelet, a privileged Daemonset reconfigures Kubelet to run in the CNI mode by overriding the Kubelet configuration file. Kubelet discovers and executes CNI Plugins from the `/opt/cni/bin` directory by default. However,in GKE's compute instances, we don't have access to this folder, so we point `cni-bin-dir` field in the Kubelet configuration file to the location of our Calico installation.

Directly installing the default Calico network plugin in a GKE cluster could make the control plane-to-pod communication unavailable. It is because our privileged Daemonset and the Calico network plugin can not be deployed on the control plane. During installation, the plugin will set up a tunnel device on each node to enable an overlay pod network supporting the pod communication across nodes. Such tunnel devices encapsulate a pod IP packet with another IP packet at the level of the cluster VPC network. However, when we directly install the plugin in a GKE cluster, the tunnel devices do not exist on the vendor-managed control plane. In addition, since the Calico network plugin uses the Calico IP Address Management (IPAM) engine to allocate IP addresses to pods dynamically from its managed IP pool, the cluster VPC network will not have routing information for these addresses. For example, Figure 5 illustrates a situation where the API server can not communicate with a webhook server, when the default Calico plugins are installed. The webhook server pod IP address `10.0.0.2` assigned by Calico is not in the pod IP range `10.124.0.1/24` of its node. Moreover, as Calico cannot be installed on the GKE control plane, the packet from the API server is not encapsulated with node-level IP packets. Therefore, the cluster VPC network can not route the packet with the destination IP address `10.0.0.2` properly. However, due to tunnels' encapsulation, pod-to-pod communication across worker nodes is still available. In summary, as the packets from the control plane are not encapsulated, and the network does not have routing information for Calico-managed IP pools, the control plane-to-pod communication is lost. A possible consequence of such disconnection is that the API server can not send requests to deployed admission webhook servers (ex. the PodPresets webhook server, as discussed in Section 5.1) , and thus lost the extensibility of admission logic.

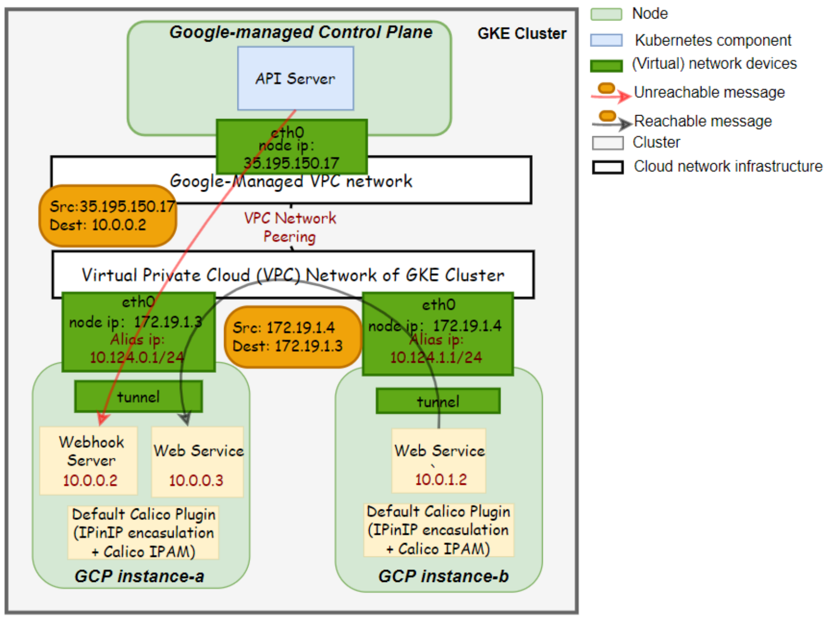


Figure 5. Pod connectivity in a GKE cluster with the default Calico plugin

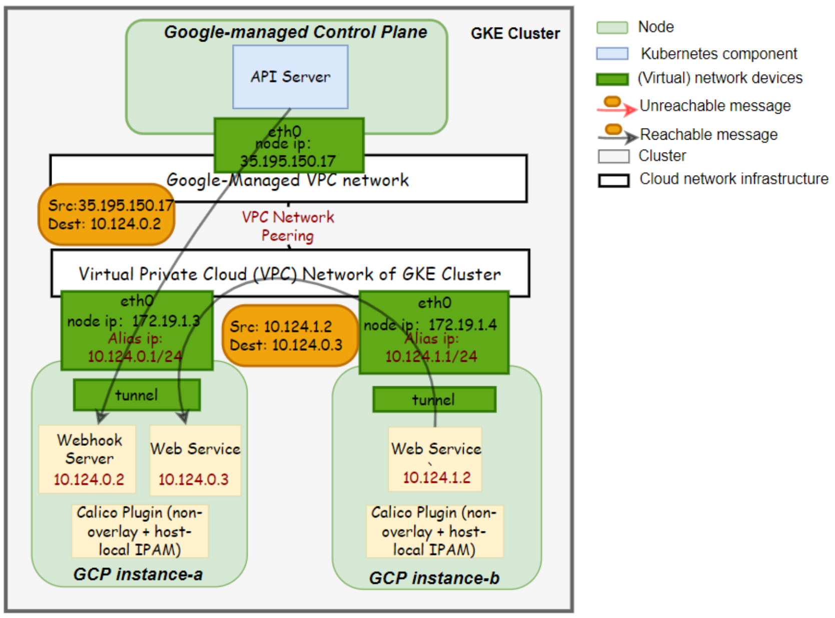


Figure 6. Pod connectivity in a GKE cluster with the configured Calico plugin

To solve this disconnection problem, before deploying the Calico network plugin, we have to modify its manifest, configuring it to run in non-overlay mode and use the host-local IPAM. The Calico network plugin running in the non-overlay mode will not encapsulate IP packets from containers and directly send them to the underlying network stack and then to the cloud network infrastructure. The host-local IPAM assigns IP addresses to pods from the IP range configured in the cluster VPC network. The control plane assigns such IP range to nodes when they join the cluster. Therefore, as the network of GKE is VPC-native by default[49], it will still be aware of pod IP addresses and able to forward traffic from the control plane to pods, as illustrated in Figure 6.

Although the reconfiguration tactic is developed for GKE and the Calico network plugin, it is easily adapted to other vendors and network plugins. For example, as AKS uses Network Address Translation (NAT) in each node to translate the pod IP address to the node IP address[50], and the EKS networking[51] uses similar VPC-native routing mechanism as GKE, all these vendors' network infrastructure can routes pod packets. Plus, many CNI plugins can be configured to run in a non-overlay mode and use the node's pod IP range, such as the native routing mode and host-scope IPAM mode in Cilium. The disadvantage of this tactic is that we cannot take advantage of overlay networking powered by CNI plugins, such as routing pod packets across different VPC subnets.

6. Design and Implementation of the Autonomic Feature Management Controller

This section presents the design and implementation of the controller that reconfigures Kubernetes features in a cluster federation using the vendor-agnostic reconfiguration tactics. To improve the cluster administrators' productivity, the controller adopts the approach of declarative management and enforces a control loop that automatically detects missing features and performs corresponding reconfiguration tactics. Section 6.1 introduces the design of the control loop that the controller implements. Section 6.2 explains architectural and technical details of the controller implementation.

6.1. Design of the Control Loop

Kubernetes uses a declarative management approach to manage cluster resources. The user specifies the desired state of cluster resources through the API server. A control loop monitors the actual state of the resources and compares it with the user-specified desired state. If there is any difference between them, the control loop takes actions to bring the actual state to the desired. The controller in our approach adopts a similar idea to implement the control loop (Figure 7) for feature compatibility management in a Kubefed cluster federation:

- The user specifies the desired features that each member cluster should support through a declarative API;
- The controller monitors actual supported features, and compares them with the desired features for each member cluster;
- If a desired feature is not supported in the cluster, the controller performs the vendor-agnostic feature reconfiguration tactics to activate the missing feature.

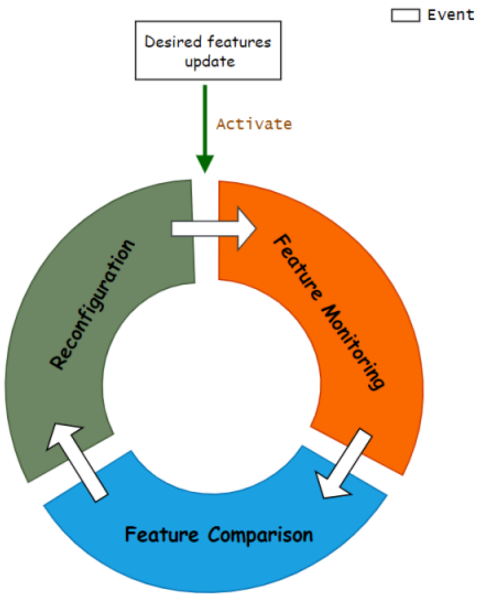


Figure 7. Feature compatibility controller's control loop

The control loop is event-driven, and it blocks after an iteration finishes. When desired features change, the control loop continues with the next iteration to maintain feature compatibility. Regularly, the controller restarts the control loop for every installed feature to see if the feature is still working.

6.2. Controller Implementation

This section describes the implementation of our prototype of the controller. Section 6.2.1 introduces our extended API resource in the Kubefed host cluster to declaratively specify the desired features. Section 6.2.2 presents the architecture of the controller and its implementation details. Section 6.2.3 describe how the controller monitors for missing or wrongly configured features.

6.2.1. Custom Resource: Featureconfig

To enforce the declarative feature management, we extend the Kubefed host cluster API server with a new custom API resource *Featureconfig* through the Kubernetes CRD API. The *Featureconfig* API can be used to describe the intended features that all member clusters should support. As such, the cluster administrators manage feature configurations like any Kubernetes resource, writing a `YAML` file to specify the desired features (ex. Listing 1), and using `kubectl` to apply it in the host cluster. Then, the controller will handle the remaining processes, reconfiguring clusters' features to the desired.

Listing 1: Example YAML file for Featureconfig API

```
apiVersion: feature.kubefed.io/v1alpha1
kind: FeatureConfig
metadata:
  name: CPUManagementPolicy
spec:
  cpumanagementpolicy: static
  podpreset: true
  networkpolicy: true
```

Note that multiple features can be toggled in a single Featureconfig object. When this is the case, the controller will try to install multiple features at once. This is possible if these features must be enabled by the same vendor agnostic tactic.

6.2.2. Controller Architecture

The controller is implemented based on Kubebuilder[52], which provides abstraction over boilerplate code for developing custom controllers, allowing us to focus on the implementation of the control logic. Figure 8 shows the architecture of the controller. Kubebuilder scaffolds the following modules to facilitate the control loop implementation:

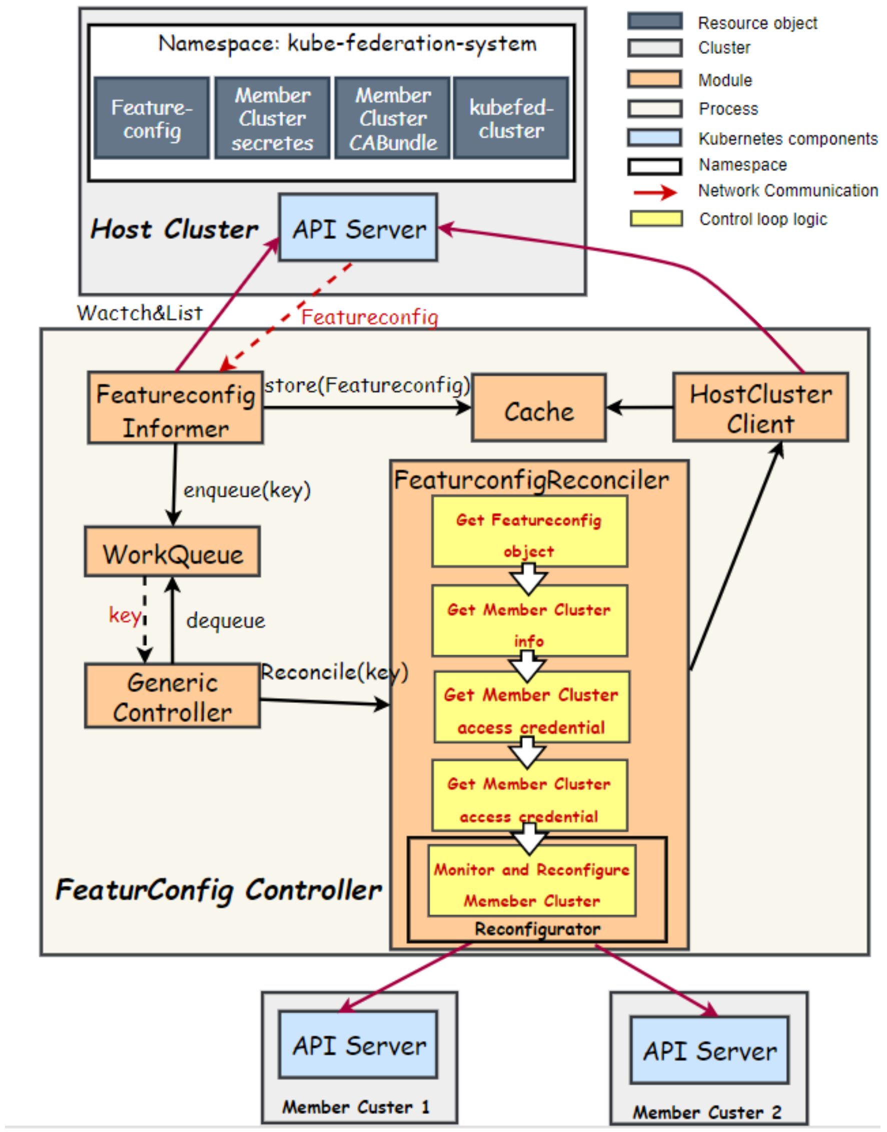


Figure 8. Architecture of the controller

- **Informer** is responsible for watching our custom API resource *Featureconfig* in the host cluster. If a *Featureconfig* object changes, the informer updates the corresponding object in the *Cache*, and enqueues the object's key (name and namespace) as an event into the *WorkQueue*;
- **Cache** is a local store that caches resource objects managed by the controller to reduce the load of the host cluster's API server;
- **WorkQueue** stores events that need to be processed, which are the key (name and namespace) of changed Featureconfig objects;
- **Generic Controller** is responsible for dequeuing an event from the *WorkQueue* and calling the *Reconcile* method in the *FeatureconfigReconciler* with this event;
- **Client** can be used by our reconciler to communicate with the API server of the host cluster. If the object the reconciler wants to fetch is already in the *Cache*, the *Client* will fetch the object from the *Cache* directly instead of sending a request to the API server.

These modules mainly watch for *Featureconfig* objects and drive the execution of the **FeatureConfigReconciler** module which is the principle component added by our work. The control loop logic of this component is implemented in the *Reconcile* method . Algorithm 1 describes the steps of the method. First, the controller fetches the user-specified desired features from the *Cache* using the *Client*, based on the name of a *Featureconfig* object.

Then we get the list of all member clusters including their API server endpoint. These are managed by the Kubefed host cluster as *kubefedclusters* API objects. Then we get the list of reconfigurators for configuring the features. A single reconfigurator can be used for reconfiguring multiple features as different features can be enabled by changes to the same configuration manifest. We than run in parallel for each member cluster a separate control loop in a concurrent actor. The control loop first gets the credentials to access the member cluster's API server. Thereafter, for each relevant feature, the reconfigurator uses these credentials to communicate with the member cluster's API server to monitor whether the desired features are supported and to apply the appropriate reconfigurator.

Algorithm 1: Reconcile method in FeatureconfigReconiler

```
Data: The key of a Featureconfig object
Result: All member clusters support features specified in the Featureconfig object
$Desired\_features := Client.GetFeatureConfig(key);$
$Member\_clusters := Client.GetMemberClusters(KubeFedCluster);$
for $feature \in Desired\_features$ do
    $rcClasses := new\ ReconfiguratorClass[];$
    $rcClass := FeatureconfigReconciler.GetReconfiguratorClass(feature);$
    if $rcClass \notin reconfiguratorClasses$ then
        $rcClasses.append(rcClass);$
    end
end
for $cluster \in Member\_clusters$ do
    $new\ Actor(Cluster := cluster,$
     $Reconfigurators := map(rcClass \in rcClasses => rcClass.instance()$
     $connection\_Credentials := Client.GetClusterCredential(Cluster))\ \{$
    for $rc \in Reconfigurators$ do
        $isReconfigured, err :=$
         $rc.Reconfigure(Desired\_features, connection\_Credentials);$
        if $isReconfigured$ then
            $err := rc.Wait\_for\_up(Desired\_features, connection\_Credentials)$
        end
        if $err$ then
            $report\_err(rc);$
        end
        else
            $report\_success(rc);$
        end
    end
    $\}.Run()$
end
```

We have implemented reconfigurators in a generic way by implementing the *Reconfigurator* interface (Listing 2):

Listing 2: Reconfigurator Interface

```
Reconfigure(desired_features, member_credential) (bool, error)
Wait_for_up(desired_features, member_credential) error
```

- **Reconfigure** is the method where we monitor and apply vendor-agnostic reconfiguration tactics for one or more desired features. It takes the desired features and credentials to communicate with a member cluster. For all desired features that the reconfigurator is able to configure, the reconfigurator will check if any of the desired features are not yet enabled in the cluster. If so, it will apply one of the vendor-agnostic tactic for these missing features and it returns true. If the member cluster supports all desired features, it will apply nothing and therefore return false;

- **Wait_for_up** is the method that waits until the features of the reconfigurator's interest are all successfully activated. This is because the tactics take some time to execute, mostly because of the stages of creating a Daemonset and pulling container images for these Daemonset pods on all worker nodes.

The reason we want the controller to wait for a feature to be activated is that the controller can report the administrator of the cluster federation about success or failure of member cluster reconfiguration.

6.2.3. Feature Monitoring

For each feature we select, we monitor whether the feature is supported by observing the cluster configuration states. To ensure that the controller monitor features in a vendor-agnostic way, we only use the native API endpoints exposed by the API-server to view the cluster configuration states. The relevant configuration states for each selected feature are as follows:

- API server: The API server has an endpoint `/apis` to list all installed API resources. The controller interacts with this endpoint and see if the desired API resource is supported;
- Kubelet: The Kubelet of a worker node exposes the endpoint `/configz` to get its configurations. However, the credential to access Kubelet endpoints is only available in the API server, which is managed by vendors for a cluster of a hosted product type. Therefore, we should use the API server's endpoint `/proxy` to redirect our requests to Kubelet. The Kubelet returns its settings in `JSON` format, and then the controller can inspect the current CPU manager policy;
- Network plugin: We found that although the network plugin mode is one of the Kubelet configurations, it is not returned when we call the Kubelet `configz` endpoints. Therefore, the controller inspects the network plugin mode by directly observing the name of the Daemonset that is installed by the CNI plugin,.

However, observing the cluster configuration states through the endpoints of the API server is not robust enough to verify that a feature is actually correctly implemented. For example, the CNI network plugin may be installed in the cluster, but the Kubelet is misconfigured and still use the original Kubenet plugin. In this case, the controller would mistakenly think the cluster is running with a CNI plugin. To address this issue, we can utilize the end-to-end testing suite provided by the open-source distribution of Kubernetes to run conformance tests to validate if a feature is supported in a cluster. However, the feature coverage of the test suite is insufficient[28]. Improving the coverage of this test suite is beyond the scope of this paper.

7. Evaluation

This section presents the conducted evaluation experiments to answer our research questions. Section 7.1 presents and motivates the research questions. Section 7.2 describes our evaluation environments in terms of used applications and Kubernetes clusters. Section 7.3 presents our experiments and their results and discusses our experimental findings for each research question.

7.1. Revisiting Research Questions

In this section, we present the research questions and elaborate on our motivations to evaluate them:

- ***Question 1***: *What is the performance overhead of reconfigured features during normal operation of cloud-native applications compared to native features supported by Kubernetes?*
  One of the goals of our vendor-agnostic reconfiguration tactics and controller is to reduce the operational cost of multi-cluster management. However, if the reconfigured feature has a significant performance overhead on the normal operation of the cluster or application, this would outweigh the benefits of vendor-agnostic reconfiguration.

So we need reconfigured features to have as little impact on performance as possible in comparison to a Kubernetes cluster where the features are configured as instructed in the official K8s documentation [5].

- ***Question 2**: What is the disruption impact on running applications when re-configuring?*
  For Kubernetes production clusters, we want the controllers to have less impact on running applications or services when executing reconfiguration tactics to ensure the normal operation of the business.
- ***Question 3**: What is the time to reconfigure features in a newly created cluster without any application running?*
  In the context of cloud bursting, K8s clusters must be set up only when there is an imminent workload peak. As such clusters must be launched and configured as quickly as possible

7.2. Evaluation Environment

This section presents our Kubernetes cluster deployment environment and test applications for evaluating our research questions.

7.2.1. Test Applications

When studying the performance overhead and disruption impact of reconfiguration tactics and the controller on clusters and running applications, we select two test applications, a Cassandra database and a SaaS application, representing two classic Kubernetes workloads, Statefulset and Deployment:

- **Cassandra Database**: The Cassandra database is a distributed, high-performance, and highly available NoSQL database widely used in the industry[54]. We evaluate our research questions by measuring the latency of CPU-intensive write operations at various workload levels (i.e number of writes/sec);
- **Configurable SaaS Application**: This SaaS application can be configured to perform various resource-intensive operations such as CPU-intensive, disk- and memory-intensive operations to simulate various real-world applications[55]. Again, here we measure the latency of CPU-intensive operations.

We use K8s-Scalar as our traffic generator to send requests to these applications and measure the latency. K8-Scalar is originally designed to evaluate auto-scaling methods for containerized services in a cluster[56]. Here we take advantage of its traffic generator that can simulate fluctuations in the workload.

7.2.2. Kubernetes Clusters

We utilize two kinds of clusters to run our test applications and conduct experiments to evaluate our research questions.

- **On-premise clusters on Openstack**: The testbed for on-premise clusters is an isolated part of a private OpenStack cloud, version 21.2.4. The OpenStack cloud consists of a master-worker architecture with two controller machines, and droplets on which virtual machines can be scheduled. The droplets have 2 x Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz (14 cores, 28 threads) 256 GB RAM. Each droplet has two 10Gbit network interfaces. The K8s cluster used in the evaluation was deployed using Kubeadm[58]. Kubeadm is an installer product that offers a rich yet proprietary customization interface[58]. Additionally, we use Terraform scripts to automate the testbed creation process. Our on-premise clusters have a control plane node of resource size `c4m4` (4 cores and 4Gb memory) and worker nodes of size `c4m4`;
- **GKE cluster**: We use GKE [27] as our base line to compare the impact of our vendor-agnostic reconfiguration tactics and controller on Kubernetes of a hosted product type. GKE's default cluster uses Kubenet, so all experiments related to switching the Network Plugin are only carried out on GKE. Moreover, like on-premise clusters, GKE can support creating clusters with our selected features through its proprietary

interfaces. For example, we can use `NodeConfig` file to partially modify the Kubelet-Configuration manifest[31], create older clusters to support deprecated APIs, and enable network policy to use the CNI network plugin[59]. Our GKE clusters have a hardware setup with a vendor-managed control plane and two `e2-highcpu-4` (4 cores and 4Gb memory) instances as worker nodes.

Also, since the cluster we need to evaluate is a member cluster in a cluster federation, we need an additional host cluster to manage it. This host cluster only runs the Kubefed control plane and our Featureconfig API, and does not run any other workloads. The cluster is deployed on the OpenStack cloud with only one `c4m4` instance.

7.3. *Experimental Results and Findings*

This section presents the experiments we conduct for each research question, presenting the setups, their results and discussing our findings. In each experiment, three features are reconfigured: the PodPresets API replica, the `static` CPUManagement policy, and the Calico CNI plugin. These correspond respectively with the three Reconfigurator tactics presented in Section 5

7.3.1. Experiment 1: Performance Overhead of Reconfigured Features

In this experiment, we compare the performance of the same application on clusters that natively support our selected features and clusters that have these features enabled using our controller and reconfiguration tactics.

Experimental Setup.

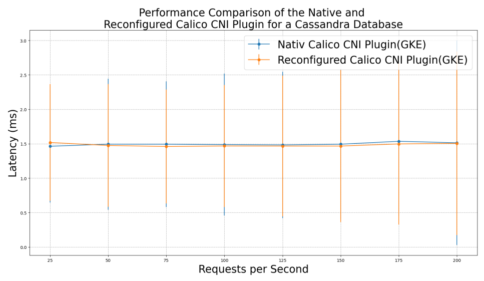
To evaluate this research question, we firstly have to create K8s clusters that support the native versions of our selected features as our baselines. This can be achieved through Kubeadm and GKE proprietary interfaces. Then we use the controller to reconfigure the default GKE and on-premise clusters to obtain clusters supporting our reconfigured features.

Also, our test applications should use the feature we evaluate. For the static CPU manager policy, both are assigned with two dedicated cores. When evaluating the PodPresets feature, we inject environment variables into our test applications. As for the CNI network plugin, we configure network policies stating only the traffic generator K8-Scalar can communicate with the test applications.

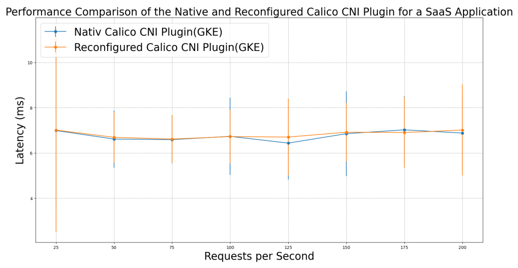
We only deploy a single replica of the test applications on the clusters. To make sure our traffic generator does not become a bottleneck, we deploy it on a different node than the test application, and it generates loads of 25 to 200 requests per second in steps of 25. We run each load 1800 seconds.

As introduced in Section 7.2.2, our GKE clusters have two `e2-highcpu-4` worker nodes and our on-premise clusters have two `c4m4` worker nodes and one `c4m4` control plane node. The Kubernetes version we use is 1.21 except for experiments about PodPresets where we use Kubernetes 1.19 which is the latest version that supports PodPresets API. As stated in Section 7.2.2, we don’t conduct experiments about network plugins in on-premise clusters.

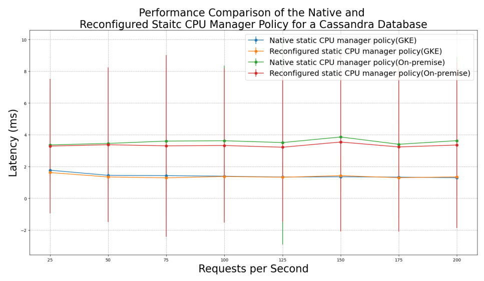
Experimental Results and findings.



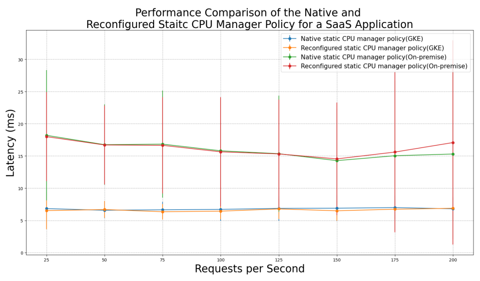
(a)



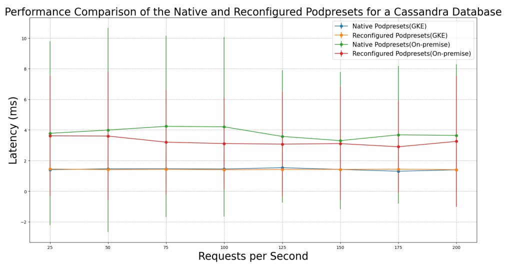
(b)

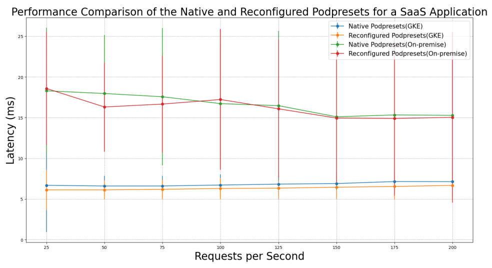


(c)



(d)





(e) (f)

Figure 9. Performance comparison of the reconfigured feature with the native

Figure 9 presents our the results of our experiments for each type of reconfigurator.

For all selected features, we find no significant difference in performance overhead between the reconfigured features and the native versions supported by K8s. For the API server reconfiguration with PodPresets, this is because CRDs and webhook servers only work when a new Pod is being created, and therefore the impact is only visible in case of auto-scaling Pods. Also, the additional webhook server, consumes almost no resources when no pods are being created. For Kubelet reconfiguration with the static CPU manager policy and the CNI plugin reconfiguration with the Calico CNI network plugin, the actual Kubelet and CNI configuration implemented by our reconfigurators do not differ from the configuration files in the base line clusters with natively configured features as prescribed in the documentation.

7.3.2. Experiment 2: Disruption Impact on Running Applications when Re-configuring

The goal of this experiment is to determine the disruption impact of the controller and reconfiguration tactics on running applications.

Experimental Setup.

Likewise, we deploy our test applications and the traffic generator on different nodes to ensure that the traffic generator does not become a performance bottleneck. We use the traffic generator to generate consecutive loads of 100 requests per second. We only run each load for three seconds to evaluate the performance of the test applications at a certain point in time. The controller reconfigures features during these loads.

This experiment is only performed on default GKE clusters but compared with proprietary customization interface of GKE. Our GKE clusters had two `e2-highcpu-4` worker nodes.

As stated in Section 5, the reconfigurators for the Calico network plugin and `static` CPUManager policy requires restarting of the Kubelet agent on all worker nodes. During a restart of the Kubelet, if a pod has readiness and liveness probes set, these probes will not be executed by the stopped Kubelet, so the pod might be observed by the control plane as not ready anymore to serve traffic, and thus the Kube-proxy will not forward requests to it. To explore the actual impact of a Kubelet restart on such pods, we set a readiness probe for the test SaaS application that the Kubelet should execute every second.

Experimental Results and Findings.

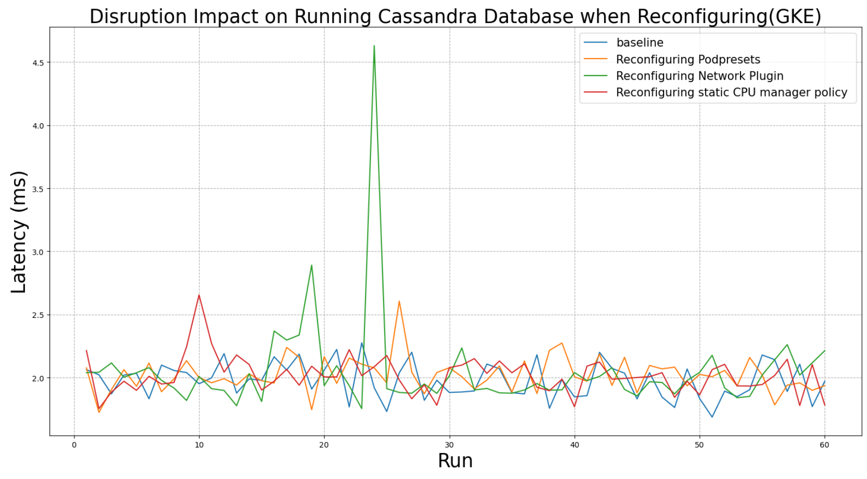


Figure 10. Disruption Impact on Cassandra Database Application

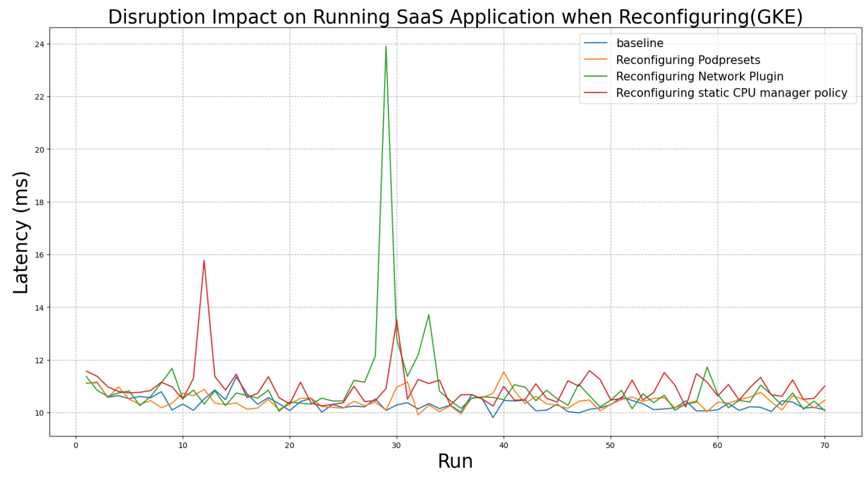


Figure 11. Disruption impact on SaaS application

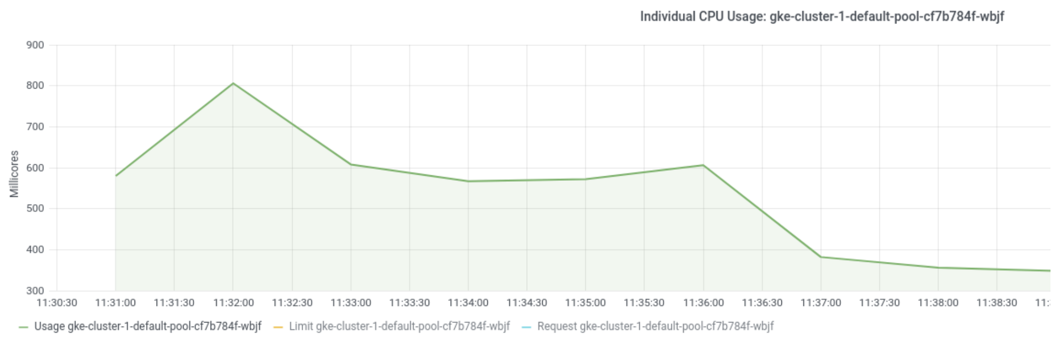


Figure 12. Cassandra node CPU usage when reconfiguring

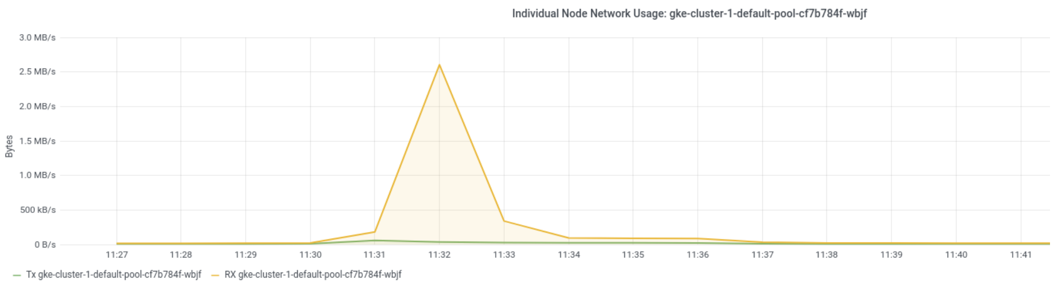


Figure 13. Cassandra node network usage when reconfiguring

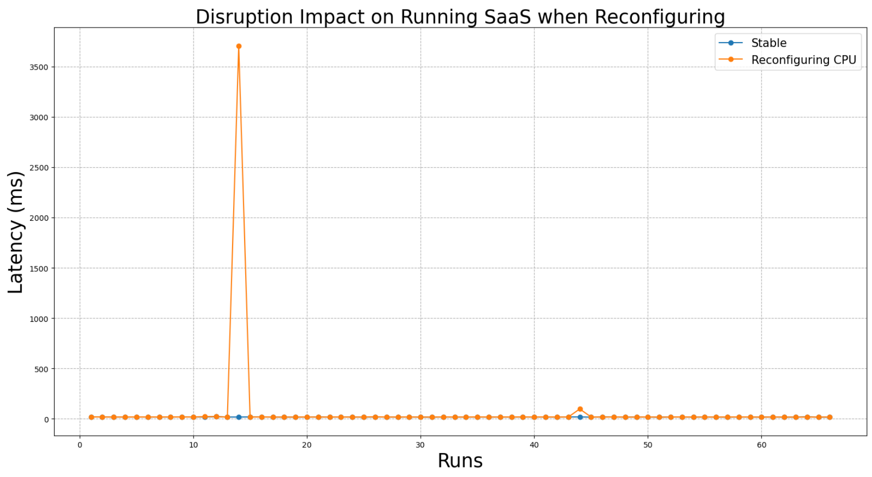


Figure 14. Disruption Impact on SaaS application(During Kubelet restart, no traffic routed to the Application)

Figure 10 and Figure 11 presents the disruption impact on the SaaS application and the Cassandra database when the controller reconfigures each feature.

We found that when the controller reconfigures each feature, for a short period of time there is increased network and CPU usage of nodes. This can be explained by the container image pulling and container creation process for Daemonset pods and dependent plug-in components. This increase is especially noticeable when pulling and installing the Calico network plugin: the node's network usage goes from around 90Kb/s during normal operation to 2.53Mb/s (Figure 13) and the CPU usage goes from about 600 to 800 cpu milliseconds (Figure 12). For the static CPU manager policy, the controller only deploys Daemonset pods to reconfigure the Kubelet settings (Section 5.2). Adding a replica of the PodPreset API has minimal disruption. This is because the container image of the

PodPresets webhook is not deployed to every node as in the case of the Calico network plugin and the CPUManagement policy.

More importantly, we found a sharp increase in latency for those reconfigurators that require restarting the Kubelet. Although the Kubelet restarts very quickly, the traffic generator seems not able to communicate with the test applications during a short period of time (Figure 14). As such, we believe that restarting the Kubelet for all nodes using a privileged Daemonset is a risky strategy that may result in short-term unavailability of cluster services.

When installing the Calico CNI plugin using the proprietary customization interface of GKE, the disruption to the applications is much smaller. This is because GKE creates new worker nodes running in CNI mode and gradually migrates workloads from the existing nodes to the new ones. As a result all Pods are restarted and therefore all use the new CNI plugin. In our solutions, existing Pods are not restarted, but they also do not use the new CNI plugin.

7.3.3. Experiment 3: Time to Reconfigure Features in a Newly Created Cluster without any Application Running

This experiment explores the reconfiguration time for each selected feature on a newly created cluster without any application running. We measure the time from when the controller starts monitoring whether the feature is supported until it deems the feature activated.

Experimental Setup.

We test the reconfiguration time of each selected feature on newly created default GKE and on-premise clusters. As introduced in Section 7.2.2, our GKE clusters have two `e2-highcpu-4` worker nodes and our on-premise clusters have two `c4m4` worker nodes and one `c4m4` control plane node. As discussed in Section 7.2, switching to the CNI plugin is only conducted on GKE. To reduce the randomness of the experimental results, we repeat our experiments five times.

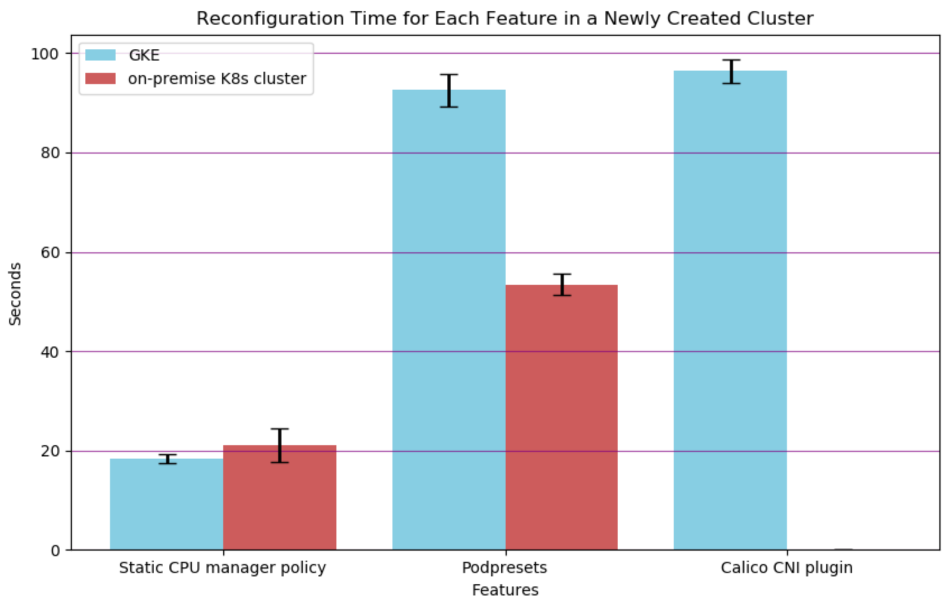


Figure 15. Reconfiguration time for each feature in a newly created cluster

Experimental Results and findings.

Figure 15 presents the average reconfiguration time for each selected feature. We find that if our reconfiguration tactics create more complex API resources on the cluster, the reconfiguration process will take longer. Additionally, the controller takes significantly less time to reconfigure the CNI network plugin than the 10+ minutes it takes to reconfigure via GKE's proprietary interface. As stated above already, this is because GKE creates new worker nodes running in CNI mode and gradually migrates workloads from existing to new nodes.

8. Conclusion

This section sets our conclusion to the paper, discusses the limitation and potential future directions. Section 8.1 summarize our approach to achieve feature compatibility management. Section 8.2 presents current limitation of our work and discusses potential future work.

8.1. Summary of our Feature Compatibility Management Approach

The main objective of this paper is to incept a unified and vendor-agnostic feature compatibility management approach for Kubernetes cluster federations.

Firstly, we have described detailed vendor-agnostic reconfiguration tactics for three problematic configuration manifests: (i) a privileged Daemonset to modify the Kubelet-Configuration manifest, (ii) using a dynamic admission webhook server and a CRD for replacing deprecated APIs of the control plane, (iii) a privileged Daemonset to modify the Kubelet configuration file to make it run in CNI mode and a configured CNI plugin to route packets without overlay mode so control plane-to-pod communication works correctly.

Secondly, we have designed and implemented an autonomic controller that can automatically detect incompatible features and apply the found relevant vendor-agnostic reconfiguration tactics. This controller enforces a control loop that takes the declarative management approach to manage feature compatibility. Users can specify the desired features that all clusters in a federation should support through our extended API *Featureconfig* in the host cluster of a federation. Then the controller takes over the remaining process, monitoring if the desired features are supported for all member clusters and executing the vendor-agnostic feature reconfiguration tactics for the unsupported features. This approach allows cluster administrators to manage feature compatibility of member clusters like managing any Kubernetes resources, reducing error-prone imperative operations, and improving productivity.

Thirdly, we have evaluated our approach according to the research questions proposed in Section 1. We have found that the performance overhead of applications that run on a cluster with our reconfigured features is close to a cluster that is configured using the official documentation. When reconfiguring incompatible features, our approach brings a disruption impact on running applications, especially when reconfiguring CNI network plugins. By analyzing the resource usage of nodes, we hypothesize the reason for this could be that the image pulling and container creation process occupy a significant amount of network and computing resources on host machines. Additionally, while reconfiguring the Calico CNI plugin and KubeletConfiguration, the procedure of restarting the Kubelet may cause pods with readiness or liveness probes to become inaccessible through their exposed services for a while. Finally, we have evaluated the reconfiguration time for three relevant features and found all features could be reconfigured within 100 seconds. It is worth noting that the controller takes significantly less time to reconfigure the CNI network plugin than GKE's proprietary interface that restarts all worker nodes and gradually migrates Pods by restarting them on the new nodes. Therefore we conclude that our approach is best applied in one of the three following use cases: (i) when starting up consistently configured K8s clusters across different vendors, (ii) when optional K8s features of existing clusters must be activated as quickly as possibly and temporary disruption to running applications on these clusters can be tolerated or (iii) when proprietary customization interfaces do not allow to activate the desired optional feature.

8.2. Limitations and Future Work

This section introduces the limitations of this work and possible future directions.

Feature monitoring.

Currently, our controller monitors the configuration states of each member cluster by querying their API server. However, this approach is not reliable enough in determining whether the desired features are supported. The configuration states queried from the API server might not match the actual configurations. For example, though the CNI network plugins are installed on the cluster, the cluster might still use the Kubenet network plugin for pod networking. However, from the view of the controller, it can not detect such mismatches. The open-source distribution of K8s has an end-to-end testing suite that can run conformance tests on clusters to determine if a feature is actually supported. But currently, the coverage of optional features in the testing suite is very low.

Security.

When cluster administrators use the controller and reconfiguration tactics, special attention needs to be paid to the privileged Daemonset. Such Daemonsets manage privileged pods residing in the same mount, IPC and network Linux namespace as the host machines. If attackers have access to such pods, they can launch attacks like installing malware in the host operating system, tampering with other pods on the same machine, switching the Kubelet to a malicious one, etc. Therefore, it is important to have some mechanisms to prevent such pods from becoming the security loophole of the cluster. There are plenty of countermeasures. First, we can set disallow any ingress traffic to the privileges pods. Secondly, the privileges of the Daemonset pods could be restricted to only those it really needs to implement the reconfiguration tactic. Thirdly, the same applies for the role-based access control policies with respect to access to the API server of the control plane. Fourthly, the DaemonSet and all its Pods must be removed after the reconfiguration has succesfully ended or after a timeout.

Scalability and Reliability.

The current prototype of the autonomic controller has not been evaluated from scalability or reliability perspective. Scalability requirements entails reconfiguring large number of features in a large cluster federations without exponential growth of reconfiguration time. From a reliability perspective, it is not clear what is the probability that the vendor-agnostic reconfigurations lead to broken control plane components or Kubelets that cannot be repaired anymore by rollback or forward rolling. **Author Contributions:** Conceptualization,

E.T; methodology, E.T.; software, H.X.; validation, H.X.; investigation, H.X.; resources, E.T and W.J.; data curation, H.X amd E.T.; writing—original draft preparation, H.X.; writing—review and editing, E.T.; visualization, H.J.; supervision, E.T and W.J.; project administration, W.J.; funding acquisition, W.J. All authors have read and agreed to the published version of the manuscript.

Funding: This research is partially funded by the Research Fund KU Leuven

Data Availability Statement: Not applicable

Conflicts of Interest: The authors declare no conflict of interest

References

1. Moreno-Vozmediano, R; Montero, R.S; Llorente, I. M. IaaS cloud architecture: from virtualized datacenters to federated cloud infrastructures. *Computer vol. 45 no. 12* **2012**, 65—72.
2. Buyya, R.; Ranjan R.; Calheiros R.N. Intercloud: utility-oriented federation of cloud computing environments for scaling of application services. In Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing, 2010, 13-–31.
3. Grozev, N.; Buyya, R. Inter-cloud architectures and application brokering: taxonomy and survey. *Software: Practice and Experience, vol. 44, no. 3* **2014**, 369–390.

4. Kratzke, N. About the complexity to transfer cloud applications at runtime and how container platforms can contribute?. In Proceedings of the International Conference on Cloud Computing and Services Science, 2017, 19—45.
5. Kubernetes. https://kubernetes.io/ (accessed on 19 Dec 2022).
6. Considerations for Large Clusters. Available online: https://kubernetes.io/docs/setup/best-practices/cluster-large/ (accessed on 28 Nov 2022).
7. A PodPreset Based Webhook Admission Controller. https://cloud.redhat.com/blog/a-podpreset-based-webhook-admission-controller (accessed on 28 Nov 2022).
8. Apache Mesos. https://mesos.apache.org/ (accessed on 19 Dec 2022).
9. Truyen, E.; Van Landuyt, D., Preuveneers, D.; Lagaisse, B.; Joosen, W. A comprehensive feature comparison study of open-source container orchestration frameworks. *Applied Sciences vol. 9, no. 5, 931* **2019**.
10. Linux Programmer's Manual - Namespaces. http://man7.org/linux/man-pages/man7/namespaces.7.html (accessed on 19 Dec 2022).
11. Linux Programmer's Manual - Cgroups. https://man7.org/linux/man-pages/man7/cgroups.7.html (accessed on 19 Dec 2022).
12. Bernstein; D. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing, vol. 1, no.3* **2014**, 81–84.
13. Verma, A.; Pedrosa, L.; Korupolu, M.; Oppenheimer, D.; Tune E.; Wilkes, J. Large-scale cluster management at google with borg. In Proceedings of the Tenth European Conference on Computer Systems, 2015, 1–17.
14. Declarative Management of Kubernetes Objects using Configuration Files. https://kubernetes.io/docs/tasks/manage-kubernetes-objects/declarative-config/ (accessed on 19 Dec 2022).
15. Kubernetes Components. https://kubernetes.io/docs/concepts/overview/components/ (accessed on 19 Dec 2022).
16. Custom Resources. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/ (accessed on 19 Dec 2022).
17. etcd: A Distributed, Reliable Key-Value Store for the most Critical Data of a Distributed System. https://etcd.io/ (accessed on 19 Dec 2022).
18. Kube-controller-manager. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/ (accessed on 19 Dec 2022).
19. Kubernetes Federation Evolution. https://kubernetes.io/blog/2018/12/12/kubernetes-federation-evolution/ (accessed on 19 Dec 2022).
20. Open Cluster Management. https://open-cluster-management.io/ (accessed on 19 Dec 2022).
21. Open Cluster Management: Architecture. https://open-cluster-management.io/concepts/architecture/ (accessed on 19 Dec 2022).
22. Kubernetes Cluster Federation. https://github.com/kubernetes-sigs/kubefed (accessed on 19 Dec 2022).
23. Kubefed: User guide. https://github.com/kubernetes-sigs/kubefed/blob/master/docs/userguide.md. (accessed on 19 Dec 2022).
24. Larsson, L.; Gustafsson, H.; Klein, C.; and Elmroth, E. Decentralized kubernetes federation control plane. In Proceedings of the IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC 2020), 2020, 354–359.
25. Amazon Elastic Kubernetes Service. https://aws.amazon.com/eks/ (accessed on 1 Dec 2022).
26. Azure Kubernetes Service. https://azure.microsoft.com/en-us/services/kubernetes-service/ (accessed on 1 Dec 2022).
27. Google Kubernetes Engine. https://cloud.google.com/kubernetes-engine (accessed on 1 Dec 2022).
28. Truyen E.; Kratzke N.; Van Landuyt D.; Lagaisse B.; Joosen W. Managing feature compatibility in Kubernetes: vendor comparison and analysis. *IEEE Access, vol. 8* **2020**, 228420—228439.
29. Kube-apiserver. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/ (accessed on 1 Dec 2022).
30. Customize Node Configuration for Azure Kubernetes Service (AKS) Node Pools. https://docs.microsoft.com/en-us/azure/aks/custom-node-configuration/ (accessed on 19 Dec 2022).
31. Customizing Node System Configuration. https://cloud.google.com/kubernetes-engine/docs/how-to/node-system-config (accessed on 19 Dec 2022).
32. Customizing Kubelet Configuration. https://eksctl.io/usage/customizing-the-kubelet/ (accessed on 9 Dec 2022).

33. Use Kubenet Networking with your own IP Address Ranges in Azure Kubernetes Service (AKS). https://learn.microsoft.com/en-us/azure/aks/configure-kubenet/ (accessed on 1 Dec 2022).
34. Kratzke, N.; Peinl, R. Clouns - a cloud-native application reference model for enterprise architects. In Proceedings of IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW), 2016, 1—10.
35. Herbst, N. R.; Kounev, S.; Reussner, R. Elasticity in cloud computing: what it is, and what it is not. In Proceedings of 10th International Conference on Autonomic Computing (ICAC 13), 2013, pp. 23–27.
36. Abdo, J. B.; Demerjian, J.; Chaouchi, H.; Barbar, K.; Pujolle G. Broker-based cross-cloud federation manager. In Proceedings of 8th International Conference for Internet Technology and Secured Transactions (ICITST-2013), 2013, 244—251.
37. Kratzke, N. Smuggling multi-cloud support into cloud-native applications using elastic container platforms. 2017. In Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017), 2017, pp. 57-70.
38. Cilium: Quick Installation. https://docs.cilium.io/en/stable/gettingstarted/k8s-install-default/ (accessed on 9 Dec 2022).
39. Operator Pattern. https://kubernetes.io/docs/concepts/extend-kubernetes/operator/ (accessed on 19 Dec 2022).
40. Policy Collection. https://github.com/stolostron/policy-collection/blob/main/stable/ (accessed on 28 Nov 2022).
41. cloud-init Documentation. https://cloudinit.readthedocs.io/en/latest/ (accessed on 19 Dec 2022).
42. Creating and Configuring Instances: Configuring an Instance. https://cloud.google.com/container-optimized-os/docs/how-to/create-configure-instance#configuring_an_instance/ (accessed on 19 Dec 2022).
43. Amazon, Launch Template Support: Amazon EC2 User Data." https://docs.aws.amazon.com/eks/latest/userguide/launch-templates#amazon_ec2_user_data (accessed on 19 Dec 2022).
44. cloud-init Support for Virtual Machines in Azure. https://docs.microsoft.com/en-us/azure/virtual-machines/linux/using-cloud-init/ (accessed on 19 Dec 2022).
45. Dynamic Admission Control. https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/ (accessed on 19 Dec 2022).
46. Automatically Bootstrapping GKE nodes with Daemonsets. https://cloud.google.com/solutions/automatically-bootstrapping-gke-nodes-with-daemonsets (accessed on 19 Dec 2022).
47. Initialize your AKS nodes with Daemonsets. https://medium.com/@patnaikshekhar/initialize-your-aks-nodes-with-daemonsets-679fa81fd20e (accessed on 19 Dec 2022).
48. Configure Liveness, Readiness and Startup Probes. https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/ (accessed on 19 Dec 2022).
49. VPC-native Clusters. https://cloud.google.com/kubernetes-engine/docs/concepts/alias-ips (accessed on 19 Dec 2022).
50. Use Kubenet Networking with your own IP Address Ranges in Azure Kubernetes Service (AKS). https://docs.microsoft.com/en-us/azure/aks/configure-kubenet/ (accessed on 19 Dec 2022).
51. Amazon EKS Networking. https://docs.aws.amazon.com/eks/latest/userguide/eks-networking.html (accessed on 19 Dec 2022).
52. The Kubebuilder Book. https://book.kubebuilder.io/ (accessed on 19 Dec 2022).
53. Add SSH Keys to VMs. https://cloud.google.com/compute/docs/connect/add-ssh-keys#gcloud (accessed on 19 Dec 2022).
54. Apache Cassandra. https://cassandra.apache.org/_/index.html (accessed on 19 Dec 2022).
55. Truyen, E.; Jacobs, A.; Verreydt, S.; Beni, E. H.; Lagaisse, B.; Joosen W. Feasibility of container orchestration for adaptive performance isolation in multi-tenant SaaS applications. In Proceedings of the 35th Annual ACM Symposium on Applied Computing, 2020, 162–169.
56. Delnat, W.; Truyen, E.; Rafique, A.; Van Landuyt, D.; Joosen, W. K8-scalar: a workbench to compare autoscalers for container-orchestrated database clusters. In Proceedings of the 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, 2018, 33–39.
57. Open Source Cloud Computing Platform - OpenStack. https://www.openstack.org/ (accessed on 19 Dec 2022).
58. Kubeadm. https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm/ (accessed on 19 Dec 2022).

59. GKE: Creating a Network Policy. https://cloud.google.com/kubernetes-engine/docs/how-to/network-policy (accessed on 19 Dec 2022).
60. XDP - Express Data Path.https://prototype-kernel.readthedocs.io/en/latest/networking/XDP/ (accessed on 19 Dec 2022).
61. eBPF. https://ebpf.io/ (accessed on 19 Dec 2022).