Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

Federation of Local K8s Clusters through a Central Cluster in the Cloud

Kemal Sedat Ceyhan

Supervisor: Dr. Stefan Weber

August 2022

A Dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science (Future Networked Systems)

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Signed: Date:

Abstract

Kubernetes is the leading open-source tool responsible for automating the deployment and management of containerized applications. One of the most important advantages of Kubernetes is "high availability" of applications. It increases software scalability and availability. Kubernetes can scale up and down applications and supporting infrastructure resources in response to changing organizational demands, enabling dynamic management of company resources. Kubernetes also enables flexibility in multi-cloud scenarios by ensuring that applications may run in either a public or private cloud.

The safest way to deploy a Kubernetes application across different regions for high-availability and disaster recovery is to create multiple-clusters in different regions. This would allow the same application to serve multiple geographical regions, improving availability & performance, disaster recovery and scaling application beyond a single cluster's limits. Multi-cluster deployment solves a range of difficulties; however, it increases the complexity of operation and maintenance as well. We will need to deploy resources and handle the clusters separately. These are the issues that Kubernetes Cluster Federation attempts to address for Multi-Cluster Kubernetes via a centralized administration interface. Kubernetes Cluster Federation is a mechanism for managing the configuration of several Kubernetes clusters using a centralized "federation control manager." Although a popular tool, there are almost no resources out there creating a Cluster Federation setup between a Central Cloud cluster acting as the host cluster and on-premises clusters running on a local machine acting as the member clusters.

This research proposes a POC where we will use a Cluster Federation tool called KubeFed to create a solution in which the functionalities and resources of a central cluster in the Cloud are automatically propagated to a local cluster on an independent machine in a separate location. These local clusters can live on small computing machines all over the world and get configurations and resources from a central cluster. End-users can make requests to local machines in nearby branches and receive the same services as those supplied by the central cluster situated further away. This way, we can propagate resources from a central cluster to local machines all around the world and decrease the end-user's latency and improve performance.

Acknowledgement

Thank you to my parents who always believed in me and supported me along the way. I really appreciate all your help and I sincerely believe that their emotional support hast been a big part of this thesis.

I also would like to take this chance to thank Dr. Stefan Weber for all his help and guidance. I am grateful for all the ideas and input you provided for this thesis. I especially appreciate your work ethic and how you always spared your time for my research. I also want to thank Professor Stephen Barrett for his valuable input during our demo session.

Contents

1 Introduction **1**

1.1 Goal of the Thesis 4

1.2 Structure 5

2 Background **6**

2.1 Kubernetes and its History 6

2.2 The Evolution in App Deployment 7

2.3 Kubernetes Benefits 8

2.4 Kubernetes Architecture 9

2.4.1 Control Plane Components 10

- Kube-apiserver 10
- Scheduler 11
- Controller Manager 12
- Etcd 12

2.4.2 Node Components 13

- Container Runtime 13
- Kubelet 13
- Kube-proxy 13

2.5 Kubernetes Objects 14

2.5.1 Namespace 15

2.5.2 Pod 15

2.5.3 ReplicaSet 15

2.5.4 Deployment 15

2.5.5 Service 16

2.5.6 ConfigMap 18

2.5.7 Secrets 18

2.5.8 Labels & Selectors 18

- NodeSelector and Labelling Nodes 19
- Affinity and Anti-Affinity 19

2.5.9 Custom Resource Definitions (CRD) 20

2.6 Summary 20

3 Kubernetes Multi-Cluster 21
3.1 Multi-Cluster Approach in K8s 21
3.1.1 Multi-Cluster Kubernetes Advantages 22
3.1.2 Multi-Cluster Kubernetes Architecture 23
3.2 Kubernetes Cluster Federation 25
3.2.1 Cluster Federation Description and Benefits 25
3.2.2 Cluster Federation Caveats 27
3.3 Other Alternatives 27
3.3.1 Cluster API 28
3.3.2 Other Tools Offered by Cloud Providers 28
3.4 Summary 30

4 Cluster Federation Projects 31
4.1 KubeFed Project (Version 2) 31
4.1.1 KubeFed Detailed Overview 31
4.1.2 KubeFed Caveats 36
4.2 ClusterMesh Cilium 37
4.3 Network Service Mesh 38
4.4 Submariner 38
4.5 Consul 39
4.6 Shipper 39
4.7 Admiralty 40
4.8 Summary 40

5 Design and Implementation 42
5.1 Proposed Solution 43
5.2 Requirements and Fulfillements 44
5.2.1 Creating a Local Minikube Cluster 44
5.2.2 Connectivity between Minikube and GKE 45
5.2.3 Exposing Minikube API Server 48
5.2.4 Installing KubeFed on Central GKE Cluster 52
5.2.5 Deploying Resources on GKE Host-Cluster 54
5.2.6 Exposing “fed-app-service” of Local Minikube 59
5.3 Summary 62

6 Evaluation 63
6.1 Latency 63
6.2 Resiliency 66
6.2.1 Disconnection of ngrok Agents 66
6.2.2 Disconnection of the Central GKE Cluster 67

6.3 Security 67
6.4 Summary...68

7 Future Work 69
7.1 Automating the Solution 69
7.2 Resiliency in KubeFed..69
7.3 Resource Allocation in KubeFed 70
7.4 Disconnection of API Server...71
7.5 Summary 71

8 Conclusion 72

A1 Appendix 76
A1.1 Creating the Minikube and its Certificates 76
A1.2 Exposing the Minikube..77
A1.3 Google Cloud Console: Merging Config Files...................................77
A1.4 Installing KubeFed on the gke-host-cluster 78
A1.5 Application Deployment on gke-host-cluster 79
A1.6 Exposing “fed-app-service” of Minikube 82

List of Figures

1.1 Kubernetes Multi-Cluster Federation 2
2.1 Application Deployment Process Evolution 7
2.2 Kubernetes Benefits 9
2.3 Kubernetes Cluster Architecture 10
2.4 Control Plane API workflow 11
2.5 Control Plane Scheduler workflow 12
2.6 Example Kubernetes Deployment 16
2.7 Example Kubernetes Default Service 17
2.8 Placing workloads on a particular node through Labels & Selectors 19
2.9 Placing workloads on particular Nodes through Node Affinity 20
3.1 Different Types of Multi-Cluster K8s Architectures 24
3.2 Federation of two clusters from different regions through a Manager 26
3.3 Cluster API Components 28
3.4 Conformant Kubernetes Clusters we can add to Anthos Management 29
4.1 List of Federated Clusters Observed from Host Cluster 32
4.2 KubeFed components automatically deployed as Kubernetes Pods 33
4.3 Sample KubeFed Template Diagram 34
4.4 Conversion of Deployment resource into Federated Deployment 34
4.5 Main KubeFed Components 36
5.1 Setting up Minikube on the Local Machine 44
5.2 Central GKE cluster located in Australia Region 45
5.3 KubeConfig files on the Local Machine with and without Certificates 46
5.4 Local Environment’s “kubeconfig” file uploaded to Cloud Shell 46
5.5 Merged Kubeconfig files in GCP 47

5.6 GCP having access to our local Minikube as a cluster context 48
5.7 Unable to access the resources of local Minikube from GCP 48
5.8 Exposing port=8001 for Minikube API server 49
5.9 Ngrok URL forwarding requests to the Local API server of Minikube 49
5.10 Minikube API server changed to ngrok URL in kubeconfig_merged 50
5.11 Successfully accessing to the resources of local Minikube from GCP 51
5.12 Creation of a test-namespace in Minikube cluster from GCP 51
5.13 test-namespace successfully created in the local Minikube Cluster 51
5.14 Installing KubeFed in GCP 52
5.15 kubefed-controller-manager pods deployed to kube-federation-system NS 53
5.16 Minikube joining the Federation and becoming a Member-Cluster 53
5.17 Confirmation of Minikube's Federation 53
5.18 List of Federated Clusters 54
5.19 FederatedNamespace object created in the GKE host-cluster 55
5.20 Namespace called "federated-ns" automatically created in Local Minikube 55
5.21 Google-Hello FederatedDeployment created in the GKE Host-Cluster 56
5.22 Deployment pods automatically created in the GKE host-cluster 56
5.23 Deployment pods automatically created in our Local Minikube Cluster 57
5.24 Google-Hello FederatedService created in the GKE Host-Cluster 57
5.25 Service object automatically created in the GKE host-cluster 58
5.26 Load-Balancer created for "fed-app-service" Service of GKE cluster 58
5.27 Reaching the "fed-app" deployment running in GKE host-cluster (AU) 59
5.28 Service object automatically created in the local Minikube cluster 59
5.29 Forwarding requests from Service Port to Container Port 60
5.30 Reaching the "fed-app" deployment running locally in Minikube (EU) 60
5.31 Ngrok URL forwarding requests to the Local Service of Minikube 61
5.32 Reaching the "fed-app" deployment running in Local Minikube Externally 61
6.1 Latency of the central GKE cluster hosted in AU 64
6.2 Latency of the Minikube cluster hosted in Dublin 64
6.3 Latency of the central GKE Cluster hosted in AU from London 65
6.4 Latency of the Local Minikube Cluster hosted in EU from London 65
6.5 Changed ngrok Endpoint for the Minikube API Server 66

Chapter 1

Introduction

Since last decade, we have witnessed the increased use of **virtualization** technologies allowing us to run countless virtual machines (VMs) on a single physical server. Virtualization greatly improved the isolation of the running applications and provided a good amount of security. This abstraction of resources resulted in the production of highly scalable, cost-efficient, and innovative applications [62]. Virtualization allows you to run several operating systems on a single physical server's hardware. However, virtualization requires a great number of resources (e.g., CPU and RAM) as each VM contains an operating system and a virtual copy of all the hardware that an operating system needs. Moreover, moving VMs between different services (e.g., cloud providers) is not a trivial task. This led to the emergence of **containerization**. A container is a lightweight software unit that contains all the necessary code, libraries and dependencies, allowing their corresponding application to run seamlessly from one computing setting to another [1]. What makes a container lightweight compared to a VM is that, the container shares the host operating system's Kernel with other containers where the operating system's shared item is read-only [2]. Rather than having multiple operating systems running in each VM, with containers, we deal with only one operating system, facilitating the scalability and ease of deployment of our applications [63, 64]. Understandably, the increased use of containerization resulted in the emergence of container orchestration platforms responsible for coordinating multiple containers running on multiple nodes. Kubernetes has been the leading technology for the orchestration of containerized applications. When a developer aims to create a highly available distributed system, one of the first things that come up to our minds is using Kubernetes. The rise of microservices caused an increased usage of container technologies because the containers offered the perfect host for small, independent applications such as microservices. In today's world, applications consist of hundreds of containers; managing such a number of containers across multiple environments using scripts and/or self-made tools can be quite challenging and sometimes impossible. This produced the need for having container orchestration technologies such as Kubernetes. Kubernetes offers high availability, scalability and disaster recovery through automatic application deployment, load-balancing and resource allocation. It manages a cluster of nodes that includes hundreds of containers. It is the leading orchestration tool for containerized applications deployed in either virtual or physical devices [3, 65]. We will talk more about the main Kubernetes objects and its architecture in detail in the next chapter.

A Kubernetes cluster is a collection of nodes that execute containerized apps. As we mentioned before, containerization of an application consists of packaging it with its libraries and/or dependencies together with all the required services [4]. This process facilitates the deployment and “moving around” process of applications. Containers may run across different computers and environments thanks to Kubernetes clusters. A cluster can include nodes from any location or environment (e.g., on-premises together with cloud, virtual, physical, etc.) With the popularity of the container technology and Kubernetes among companies, it is not unusual for businesses to operate many clusters instead of a single cluster. The use of a single cluster may present certain problems: single point of failure, inability to control the users' access to certain resources within the cluster, applications competing with each other for the use of shared resources, etc. This resulted in the use of ‘multi-cluster’ approach in Kubernetes [5]. By deploying multiple clusters, organizations aim to improve their applications’ overall availability, scalability and failure and business isolation [6]. However, since the applications are deployed to different clusters’ nodes separately, they need to be managed separately as well. All the configurations and application deployments need to be repeated on each cluster even if the applications are exactly the same. This resulted in the emergence of “Cluster Federation” concept.

The core assumption of Kubernetes Cluster Federation is that a unique source of application configuration (e.g., deployment, service, etc.) is applied to a central location known as the Host Cluster, and that the configuration is automatically propagated to all clusters in the environment according to specified criteria. This provides the application with a unified view of all Kubernetes clusters as a single cluster. In an ideal world, there should be no need to configure or deploy the application numerous times in separate clusters, or to retain an application's state per cluster. The app developer does all the work in the host cluster and all the configurations are automatically reflected in the child or member clusters [7]. Below is a figure demonstrating the process of cluster federation:

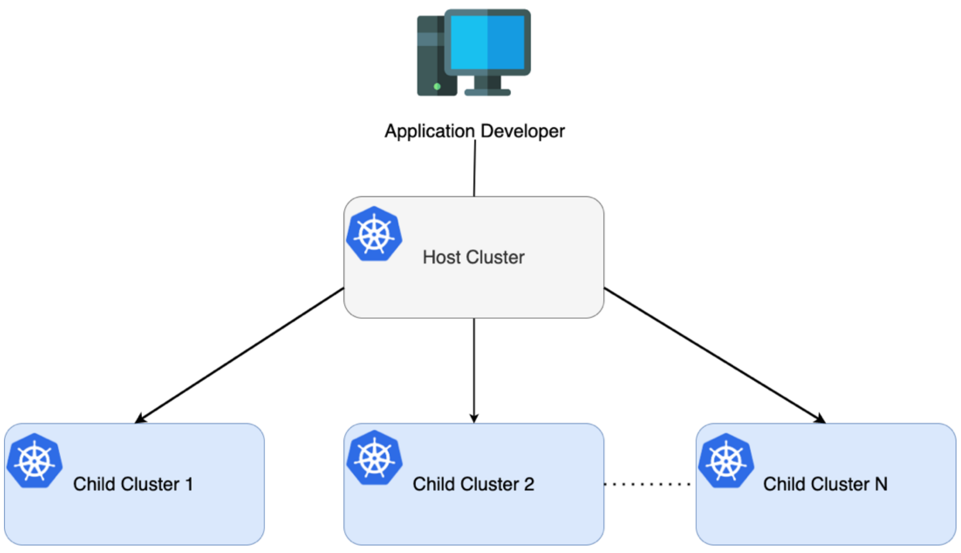


Figure 1.1: Kubernetes Multi-Cluster Federation

The Cluster Federation is achieved by a host cluster that contains controller managers responsible for the propagation of Kubernetes resources among child clusters. After that, the deployed resources are federated through the host cluster. The application developer can change the configuration of certain resources of child clusters with the help of the host cluster's controller managers. Kubernetes Cluster Federation facilitates the process of application deployment by creating a 'single cluster' environment while enjoying the benefits that come from multi-cluster setting: We create multiple clusters, each having multiple nodes and applications running in these nodes just like in a regular "multi-cluster" setting. However, in cluster federation, we make one of the many clusters the "Host Cluster" where the others are "Member Clusters". We create the configurations and deploy the applications on the Host Cluster, and all the resources are automatically propagated to the member clusters. Thus, we enjoy the ease of management that comes from a single cluster setup, while having the resiliency, high availability, and disaster recovery features of a multi cluster environment.

Most of the Kubernetes Cluster Federation or Management tools are still in development and the ones offered by big cloud providers and used in production do not fully provide the freedom to manage different types of clusters other than what their platform offers. For instance, Google Anthos [8] is not a Cluster Federation tool that has the feature of automatically propagating resources across multiple member clusters from a single Host Cluster, but it is a production-ready solution for multi-cluster management within GCP or across different cloud providers. Anthos lets us manage workloads running on third-party cloud environments such as Oracle, Azure, AWS, as well as on-premises Kubernetes clusters. It enables developers to monitor and manage the configurations and policies of clusters running in different environments through a single platform called Anthos. However, it has a limited range of environments that it supports, and it is a multi-cluster management tool rather than a federation tool [9]. The solutions offered by other cloud providers are similar.

The open-source Cluster Federation tools (e.g., KubeFed) that are mostly still in development phase do allow the federation of multiple Kubernetes clusters across different regions deployed in different cloud providers. For instance, using the most popular open-source cluster federation tool called KubeFed, the host cluster can run in GKE and federate deployment resources across multiple Azure clusters. Although there are a few examples creating such a setup out there, the number of such examples and the instructions to simulate the setup are quite limited with no tangible demonstration. However, there are almost no resources out there creating a Cluster Federation setup between a Central Cloud cluster acting as the host cluster and on-premises clusters running on a local machine acting as the member clusters. In such a scenario, the federated on-premises cluster running on a local machine could be a small branch of a company and the central cloud cluster could be the host cluster federating resources from a central cloud (e.g., GCP, AWS, Azure) to the local machines at branches in different regions. The goal of our thesis will be to create the POC simulating this scenario.

1.1 Goal of the Thesis

Imagine we have a central cluster with several applications running on different nodes in a Cloud. These applications running on a central cloud can have any type of functionalities such as banking, healthcare, gaming, etc. And we want these same functionalities to also reside in several small branches **outside the central cloud platform**. These branches can be local machines located in regions different from the central cloud's cluster to improve latency and availability. We can think of these local machines as machines with low-medium processing units. As mentioned in the previous section, such a scenario can be achieved through Cluster Federation, however, there are no tangible examples or instructions for creating this setup. Through cluster federation tools, we can automatically propagate the central cloud cluster's resources, and therefore functionalities, to the clusters located in local machines that are independent from the Cloud Platform. This way, all the resources in the central cloud cluster can also reside in local branches in different regions and be accessible by the users in those regions. On top of providing the same functionalities of the central cloud to the local machines on the branches, this would also decrease the latency for the end-users. An example scenario with this setup may include lots of small local machines with low computing power for IoT devices. The clusters running in these local machines may be federated by a 'host-cluster' residing in a central cloud and the local clusters can automatically execute the applications deployed in the host-cluster of the central cloud. This way, the local machines can execute the computing processes at the edge network for the IoT devices located all over the world and decrease latency. Moreover, the applications in the clusters of the local machines can be easily reconfigured and propagated by the host-cluster running in the central cloud.

Hence, this thesis designs a Proof of Concept to create a simulation of the scenario described above. We will use a Minikube cluster in our local environment using our own machine in Dublin, Europe. This will represent a 'local branch' in Europe. The central cluster that will serve as the 'host-cluster' for the Cluster Federation will reside in a Cloud Provider. This host-cluster will be located in a region different from Europe and will be responsible for propagating resources to the Minikube cluster in our local machine. All the resources and functionalities created in the central cloud cluster will be automatically propagated to Minikube. Users near this local branch in Europe can use the applications running in the central cloud cluster by simply accessing the Minikube cluster residing in a local machine near them. This will require establishing a connection between the Minikube cluster running on a local machine and the central cloud. For our solution, we will use the KubeFed tool, which is the leading open-source Cluster Federation technology.

1.2 Structure of the Thesis

The next sections of this thesis are as follows. Chapter 2 provides a comprehensive overview on Kubernetes, the main technology this thesis is based on. We briefly talk about Kubernetes background and then explain its overall architecture and list its main components. In Chapter 3, we talk about the concept of “Multi-Cluster Kubernetes” together with its advantages and disadvantages. We then connect it to Cluster Federation and provide a brief overview about how Cluster Federation fits into the world of multi-cluster Kubernetes. In Chapter 4, we mention some of the popular Cluster Federation tools with focus on KubeFed and how it works since it is one of the main tools we will use in our design. Chapter 5 provides our design & implementation for the proposed POC in this thesis. In Chapter 6, we will evaluate our POC solution by providing its advantages and disadvantages through measurements. In Chapter 7, we will talk about how our project fits into the real-world and what could be improved if we had enough time and resources. Finally, we will conclude our thesis in Chapter 8.

Chapter 2

Background

This chapter provides more detail on Kubernetes, its main components and its underlying architecture. This will help us for the later sections on Cluster Federation and POC design & implementation. The first section will introduce a brief history and evolution of Kubernetes [10]. The second section will describe the underlying architecture of Kubernetes in detail. Finally, we will describe the main Kubernetes objects that are widely used by almost all Kubernetes developer.

2.1 Kubernetes and its History

Before talking about Kubernetes, we need to introduce Google's **Borg** system first, which is the precursor of Kubernetes. Google defines its Borg system as a cluster manager responsible for running hundreds of thousands of jobs, from many thousands of various applications, across a number of clusters each with up to tens of thousands of machines [66]. This was a very small project that initially included around 5 people in 2004.

Later in 2013, Google came up with **Omega**, an intelligent scheduler for thousands of clusters. What made Omega unique was its new cluster scheduling mechanism that would be the foundation of Kubernetes. Omega highly utilized parallelism, shared-state, and lock-free optimistic concurrency control to use each cluster's full potential [67]. Google used Omega for both short-term batch type jobs and for their infrastructure services (e.g., BigTable).

During the mid 2014, Google introduced **Kubernetes**, an open-source variation of their long-used Borg. One of the big differences is that Borg was written in C++, and Kubernetes in Golang. Later in 2015, the collaboration between Google and the Linux foundation resulted in a rapid use and popularity of Kubernetes. Currently, Kubernetes is managed by the Cloud Native Computing Foundation (CNCF) [61]. It is being used by almost all big company that deals with container orchestration today [68, 65]. Kubernetes is the leading open-source tool responsible for automating the deployment and management of containerized applications. We mentioned in the introduction that containers help packaging applications with their respective libraries and dependencies. The containerization helps the application become a single runnable unit that can be executed in any platform without any extensive configuration. This immensely simplifies the mass execution of applications since the developers won't have to deal with any installation of application requirements. Now, we have hundreds of thousands of easily executable containers. However, this will introduce a new problem: how to manage all these containers and make sure that we have no downtime (e.g., containers smoothly taking each other's place when some go down, etc.) This is where Kubernetes comes in! In the next section, we will talk about the evolution of application deployment throughout the last 20 years to demonstrate the advantages that Kubernetes provides.

2.2 The Evolution in App Deployment

In the last two decades, the application deployment process has changed significantly. Below is an illustration of these changes [11]. Official Kubernetes site has given three categories to illustrate the evolution of application deployment processes: traditional deployment, virtualized deployment and container deployment [11].

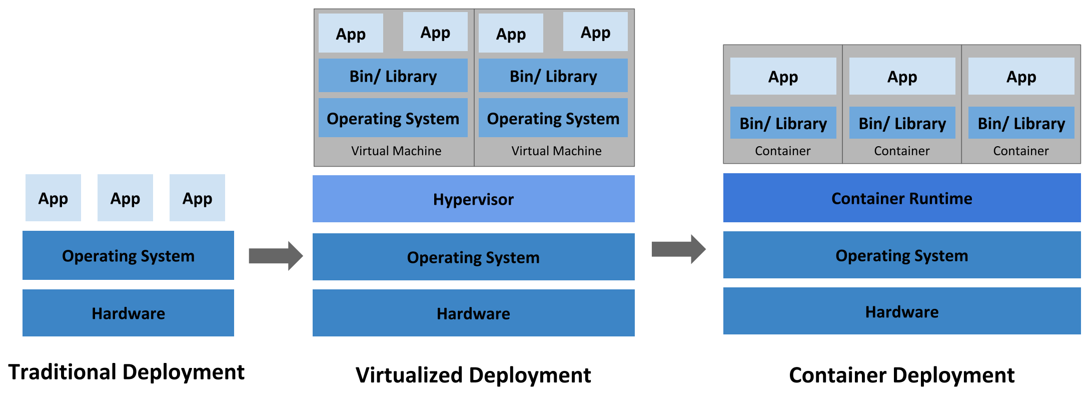


Figure 2.1: Application Deployment Process Evolution [11]

Traditional deployment refers to running applications on physical machines. An application developer would build applications and execute them on either a single or different physical machines. The obvious problem in this scenario is the lack of isolation between different applications running on the same physical server. There may be multiple applications running on the same machine and one of them may be using more resources than the others, leading the other applications to perform poorly. This is a typical example of resource competition between applications. In traditional deployment, application developers would have hard time creating boundaries between applications to allocate each with their own proper resources. To deal with this issue, it is possible to run each application on a different physical machine, totally isolating them from each other for a better resource allocation. However, this may not always be feasible as this approach is neither financially viable nor scalable. Dealing with multiple servers for only small deployments would yield unnecessary cost. It is also highly probable that applications deployed on different physical servers be allocated more resources than they need.

To deal with the above-mentioned issues, we were introduced with the **virtualization**. Virtualized deployment allowed developers to run multiple Virtual Machines on a single physical machine's CPU [11]. By running applications on different VMs, we enjoy an increased isolation between apps in a single physical server without the need to deploy them in multiple physical machines. With the increased isolation in a single machine, VMs offered a high level of security and a better resource utilization. Since the applications are virtually isolated, they couldn't easily access each other's data or resources. The power to add, remove or update many virtually isolated applications in a single physical machine improved scalability and dramatically reduced hardware costs. However, the isolation of applications came with a significant overhead. Since each virtual machine is a full machine that has all the components, it also includes its own OS on top of the virtualized hardware [11].

To deal with the 'heavy-weight' nature of virtual machines, a new solution came to the rescue: **containerization**. Containers are similar to the virtual machines, but they are 'light-weight'. This is because containers share the Operating System among the applications unlike VMs that have their own Operating Systems each. Like VMs, containers also have their own CPU share, memory, filesystem, etc., but they can be easily moved around across different cloud or OS distributions [11]. This property is what makes containers unique today. Furthermore, due to their "light-weight" nature, containers are so much faster than VMs. Starting, stopping, and updating containers require much less work than Virtual Machines. The power of containerization is what makes Kubernetes popular today.

2.3 Kubernetes Benefits

One of the most important benefits that Kubernetes brings is "high availability" of applications. It provides increased software scalability and availability. Kubernetes can scale up and down the applications and supporting infrastructure resources according to the organization's changing demands, enabling the dynamic management of company resources [12, 11]. This helps organizations save a lot on their ecosystem management with automated and smart relocation of resources based on workloads. Moreover, it provides perfect disaster recovery by checking the lifecycle of resources constantly; it saves the latest state information and keeps the applications running from the latest recorded state.

Kubernetes also provides flexibility in multi-cloud environments by making sure that the applications successfully operate in any public or private environment. Kubernetes also simplifies the smooth migration of applications from on-premises environment to public or private clouds. Essentially, Kubernetes helps facilitate the development, deployment and release processes of applications. Below is a summary of Kubernetes benefits:

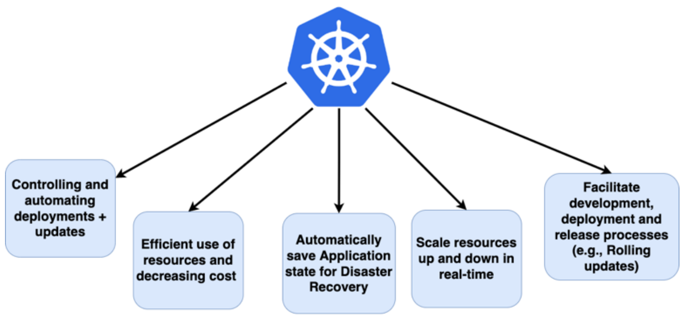

Controlling and automating deployments + updates
Efficient use of resources and decreasing cost
Automatically save Application state for Disaster Recovery
Scale resources up and down in real-time
Facilitate development, deployment and release processes (e.g., Rolling updates)

Figure 2.2: Kubernetes Benefits

2.4 Kubernetes Architecture

Kubernetes consists of three main components: cluster, node and pod. Kubernetes **cluster** is basically a set of worker machines (e.g., **nodes**) responsible for executing containerized applications. A **node** [13] is a simple server, either a physical or virtual machine consisting of computing hardware. It is basically a representation of a single machine in a Kubernetes cluster. A node is either a datacenter physical machine, or a virtual machine hosted on a cloud provider such as GCP, AWS or Azure. We can view a node as a machine that consists of a set of RAM and CPU resources to be used for running applications.

A Kubernetes **cluster** [14] is simply put an "intelligent node pool". When you deploy applications to the cluster, it manages work distribution to specific nodes smartly for the application developer. If any nodes are added to or withdrawn from the cluster, the cluster will reallocate work as needed. It shouldn't matter to the application developer what particular machines are responsible for executing the code. We can think of a Kubernetes Cluster as a hivemind that takes all the responsibility of dealing with individual machines (e.g., nodes) for application deployment.

We have previously talked about what a **container** [15] is and how it packages an application for an easy execution and moving around. Instead of executing containers directly, Kubernetes encapsulates one or more containers into a higher-level structure known as a **pod** [16]. Containers residing in the same pod share the same local network and resources. Pods provide intercommunication between containers as if they were on the same computer, while being isolated from others. They may hold multiple containers and be easily scaled up and down based on the load. Moreover, pods are the smallest units of a Kubernetes cluster. A pod is an abstraction over a container. They create a running environment, or a layer on top of the container so that an application developer won't need to directly work with containers. They are ephemeral units residing in nodes and meant to be deleted and recreated multiple times during the lifetime of a Kubernetes cluster.

A Kubernetes cluster contains two types of nodes: **master nodes** (also called control plane) and **worker nodes** [17]. Master nodes are responsible for providing the communication between the application developer and the cluster itself and scheduling the workload among the nodes. The worker nodes are responsible for executing applications containing the business logic. Thus, they generally require more resources (e.g., memory, CPU, etc.) than the master nodes. Below is a diagram showing the overall architecture of a Kubernetes cluster. It shows all the necessary components required in both the control plane and the worker nodes. In the next section, we will give a brief description for all these components [17].

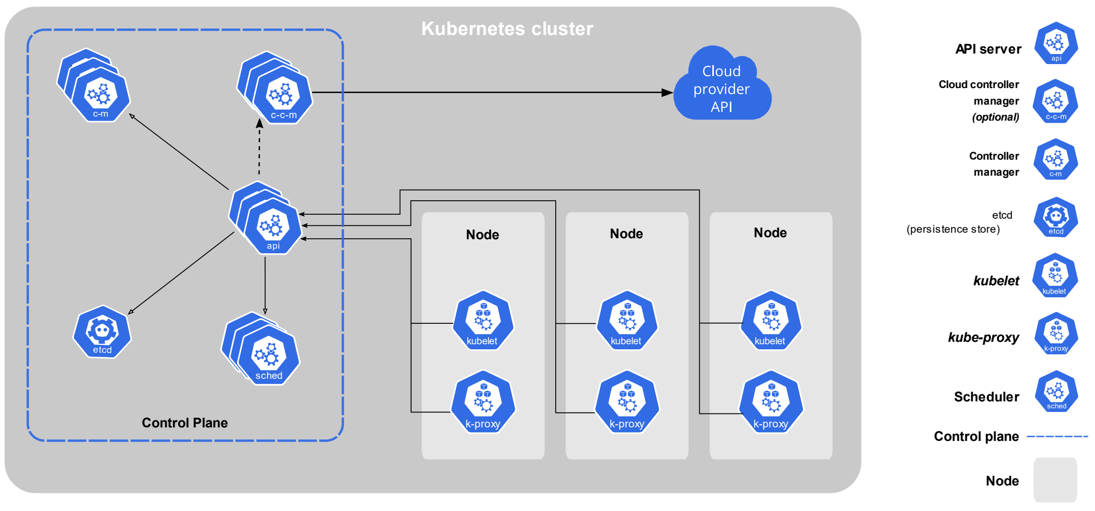


Figure 2.3: Kubernetes Cluster Architecture [17]

2.4.1 Control Plane Components

The control plane is the brain of a Kubernetes cluster responsible for making global decisions about the cluster [17]. For instance, when a pod is unhealthy, the control plane deletes and recreates a new pod. In general, the control plane components run on the same machine (master node) and this machine does not run the user created containers in it. Rather, the control plane is responsible for managing the applications running on the worker nodes. There are four processes that run on every master node that control the cluster state and the worker nodes.

Kube-apiserver

Control plane's API server [17] is responsible for exposing the Kubernetes API acting as the front-end for the application developers. When a user wants to deploy a new application in a Kubernetes cluster, they interact with the API server using some client (e.g., Kubernetes Dashboard, command line tool like kubectl, etc.) The API server can be thought as a cluster gateway which gets the initial request of any updates into or queries from the cluster. It also acts as a gatekeeper for authentication, making sure that only the authenticated and authorized requests get through the cluster. Whenever the user wants to schedule new pods, deploy new applications, or any other components, the user must talk to the API server on the master node. Then, the API server validates the request, after which it forwards the request to other control plane processes in order to schedule the pod or create a particular component that the user requested. The same process applies when the application developer wishes to query the status of deployments or cluster health, etc. The user makes the request, and the API server returns a response.

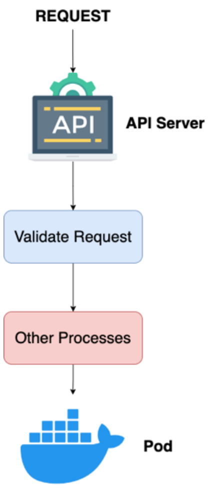


Figure 2.4: Control Plane API workflow

Scheduler

Now, the user sends a request to the API server to schedule a new pod. After the API server validates the user's request, it will be handed over to the scheduler in order to start that application pod in one of the worker nodes. Instead of just randomly assigning it to any node, the scheduler has an intelligent way of deciding on which specific worker node, the next pod or component will be scheduled. The scheduler checks how much resources (e.g., CPU, RAM) the application to be scheduled will need, and then the scheduler will go through the worker nodes and see the available resources on each one of them. For instance, if one of the worker nodes is the least busy or has the most resources available, the scheduler will schedule the new pod on that node [17, 18].

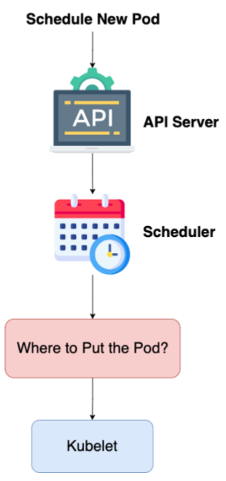


Figure 2.5: Control Plane Scheduler workflow

Controller Manager

What happens when pods die on any node? There must be a way to detect that the pods died and then reschedule those pods to recover the cluster state as soon as possible. What Kubernetes Controller Manager [17, 19] does is to detect those state changes like crashing of pods and help the cluster reach its desired state once again. To achieve this, Controller Manager makes a request to the Scheduler to reschedule those dead pods, and the process described in the Scheduler section happens again: The Scheduler decides where to put the new pod based on the worker nodes' resource calculation and sends the request to the "kubelet" of the appropriate worker node to restart the new pod.

etcd

This is the last control plane process that can be considered as the cluster brain. This is because every change in the cluster gets saved or updated into the key-value store of etcd. All the processes described above in the controller manager and scheduler sections happen thanks to the data stored in the etcd [20]. For instance, the Scheduler knows what resources are available on each worker node through etcd's data. Using that information, scheduler schedules pods on worker nodes that have the most available resources. Cluster Manager knows that a cluster state changed through etcd as well. Finally, when we make a request to the Control Plane's API server to query the cluster health, the API server's response is based on the stored data in etcd. However, we need to keep in mind that application data (e.g., database data) is not stored in etcd. etcd stores only cluster state information which is used for the master processes to communicate with the worker processes and vice versa.

2.4.2 Node Components

The Kubernetes worker nodes run the pods that are responsible for the application workloads. These components run on every node, maintain running pods and offer Kubernetes runtime environment. There are three main node processes as described below.

Container Runtime

The container runtime [21] is the software that oversees the container execution. Kubernetes supports multiple container runtimes, including Docker, containerd, CRI-O, or any implementation of the Kubernetes Container Runtime Interface (CRI) [22]. The most common is the Docker runtime, and we will be using Docker throughout our work.

kubelet

As we have previously mentioned in the Control Plane section, kubelet [23] runs on each cluster node and is responsible for running containers in pods. Application developer makes a request to the Control Plane API server to run, stop or restart a pod, and the Scheduler decides on which node a new pod will be scheduled. The process that actually starts that pod with a container is the kubelet. The kubelet gets the request from the scheduler and executes that request on the node. kubelet gets the pod configurations from the API server and communicates with the container runtime to run them on the nodes.

kube-proxy

kube-proxy is basically a network proxy running on each node, responsible for maintaining network rules on nodes [17]. kube-proxy implements essential functionalities for Kubernetes services, which we will discuss in the next section. By updating rules in IP tables, kube-proxy helps the communication from and to the pods in the cluster.

2.5 Kubernetes Objects

Kubernetes defines various types of objects, and these objects form the building blocks of Kubernetes [24]. The Kubernetes API discussed in the previous section is used for querying and managing these objects (e.g., create, delete, update, get, etc.) There are quite many Kubernetes objects, however, in most cases we work with only a handful of them. Kubernetes Objects, in most cases, have the following fields as can be seen from the example configuration files (Figure 2.6 and 2.7).

- **apiVersion**: This is the specific version of the Kubernetes object's schema
- **kind**: This is a string val. that represents the Kubernetes object's associated REST resource
- **ObjectMeta**: The Kubernetes object's metadata that includes 'name', 'labels', 'annotations', etc.
- **ResourceSpec**: Description of the Kubernetes object's desired state as defined by the application deployer
- **ResourceStatus**: The current state of the resource

The application developer can execute basic CRUD operations on these Kubernetes resources such as:

- **CREATE**: creating the Kubernetes resources in the storage backend. The creation happens with the desired state
- **READ**: There are three main operations that come with the READ:
  - **GET**: Fetch a specific resource using their name
  - **LIST**: Fetch all the Kubernetes objects of a specific type within a specified namespace. The application deployer can restrict the fetched results to match a selector query
  - **WATCH**: Serves to stream the results for an object(s) as it gets updated
- **UPDATE**: There are two main operations that come with the UPDATE:
  - **Replace**: Serves to replace the existing spec with the provided one
  - **Patch**: Apply a change to a specified field
- **DELETE**: Delete a resource from the cluster. Sometimes the child objects of the resource need to be removed before we can delete the parent resource

In the next section, we will briefly introduce the core Kubernetes objects that are relevant to the thesis.

2.5.1 Namespace

In a Kubernetes cluster, we generally organize resources using namespaces [25]. We can think of a namespace as a virtual cluster inside a Kubernetes cluster where we create virtual subgroups of resources. The other objects that we will talk about in this section are all deployed within a particular namespace of a Kubernetes cluster. Kubernetes cluster provides 4 namespaces out of the box: default, kube-node-lease, kube-public, kube-system. For instance, the kube-system namespace in a cluster includes components that are the system processes. Upon the creation of a cluster, default namespace is the namespace in which all the Kubernetes objects (e.g., deployments, pods, services) are created if no other namespace is specified. The question we need to ask next is: why do we need namespaces? They help the isolation of resources within a single cluster. Different groups of developers with different teams/projects in the same cluster can share resources without interfering with each other's work. For instance, there may be three different custom created namespaces for development, testing, and production work in a single cluster with each having specific resource quota.

2.5.2 Pod

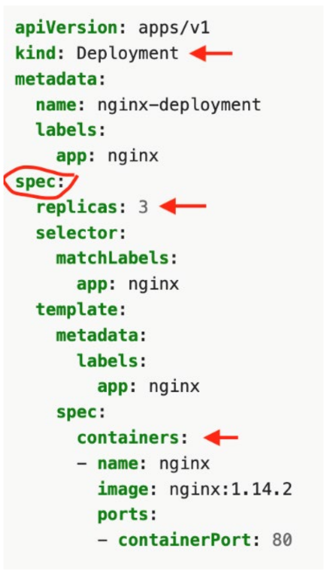
Pods [16] are the smallest processing units in a Kubernetes cluster. They are a collection of a single or multiple containers sharing the same resources (e.g., network, storage, etc.) that are scheduled together on the same pod. They are basically wrappers around one or multiple containers. Pods are ephemeral and meant to be destroyed and recreated during the lifetime of a Kubernetes cluster. Each pod in the cluster is assigned an IP address that's used by all the containers within that particular pod. Other pods in the cluster use each other's IP addresses for communication. A Kubernetes developer does not directly create or manage pods. Instead, they handle 'deployments', which we will talk about soon.

2.5.3 ReplicaSet

ReplicatSet is responsible for maintaining a stable set of replica pods running at any given time [26]. It ensures that a desired number of pods are always running in the cluster. For instance, if a pod in the ReplicaSet is deleted, the ReplicaSet notices that the number of running replica pods read from the ResourceStatus do not match with the desired number of replicas read from the ResourceSpec, and the ReplicaSet creates more pods to match the number of desired state. An application developer generally does not work with ReplicaSets directly and lets the 'Deployment' object handle the replicas.

2.5.4 Deployment

Kubernetes Deployment object [27] is responsible for the creation, update, and deletion of pods. In practice, the application deployer would not be creating pods or ReplicaSets but creating deployments where they specify how many replicas they need to run. Users can scale up and down the number of replicas of pods they need. A deployment automatically creates a ReplicaSet, which then creates the desired number of pods specified in the ‘ResourceSpec’. As mentioned before, a Pod is a layer of abstraction on top of a container(s); Deployment is another layer of abstraction on top of a pod(s). Below is an example nginx deployment from the official Kubernetes website [28]. This deployment file creates a ReplicaSet that includes three nginx pods:



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Figure 2.6: Example Kubernetes Deployment [28]

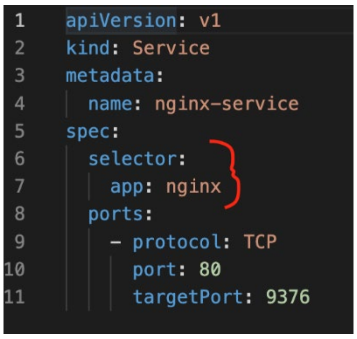
- In this example deployment file, a deployment with the name “nginx-deployment” is created under the **“.metadata.name**” section.
- Under the “**.spec.replicas”** section, we can see that a total of three replica pods are created.
- The selector section with the “**app: nginx**” key-value helps the Deployment know which Pods to manage.
- The template section has the Pod labels under **“.metadata.labels**” field (app: nginx). This matches with the key-value in the selector section.
- The spec field has a “containers” **(.spec.containers**) sub-field. We can see that the Pods only have a single container with the ‘nginx’ Docker image.

2.5.5 Service

As mentioned in the ‘Pod’ section, Kubernetes pods are ephemeral and are meant to be destroyed and recreated during the lifetime of a Kubernetes cluster. Pods have their unique IP addresses that other cluster objects use for communication. However, every time a pod gets destroyed and recreated, they are assigned a new IP address. Using that newly assigned IP address for communication may overcomplicate things in the cluster since it is very common for pods to be recreated and assigned new IPs. This is where Service come to the rescue. A Service [29] is a static IP address that can be attached to each pod to expose them. The important point is that the lifecycles of Pods and Services are not connected. Even if the Pod dies, the Service, and its static IP address will remain intact, and the users can access the pods using the same Service static IP. There are several service types providing different access scopes to the pods [69]. The service types are:

- **ClusterIP**: If no Service type is provided in the configuration file, the Service gives a ClusterIP to the pods. The ClusterIP essentially corresponds to an internal IP address, and pods with ClusterIP can only be accessed by other resources in the same cluster and not by external objects.
- **NodePort**: The pods with NodePort service type are accessible from outside the cluster through a static NodePort in the range of 30,000 – 32,767. The Service is essentially exposed through a static port of each Node's IP. The access to the NodePort service happens through a request in the form of '<NodeIP>:<NodePort>'.
- **LoadBalancer**: This type is the most popular one. It exposes the Kubernetes Service externally through a cloud provider's LB solution (e.g., GCP, AWS, Azure). The LoadBalancer automatically redirects the external requests to the corresponding service.

Below is an example service that we connect to the 'nginx' deployment from the previous section. Notice that no Service type is provided. This corresponds to the 'ClusterIP service type.



```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Figure 2.7: Example Kubernetes Default Service

Remember that our previous nginx deployment had "app:nginx" label. Notice in our Service configuration that in the selector field we have the same "app:nginx" label. This means that our service exposes any Pod in the cluster with the "app:nginx" label. To summarize, we have a Service with the name "nginx-service", redirecting all requests that come from TCP port 80 to port 9376 for any Pod that has the label "app:nginx".

2.5.6 ConfigMap

A ConfigMap [30] is a key-value dictionary of configuration settings. They allow application developers to separate configuration data/files (e.g., environment specific configurations) from containers to make applications portable. They may contain config. information such as strings, hostnames, URLs as key-value pairs for certain Kubernetes objects. The configuration settings and which Kubernetes objects they are applied to are determined through 'Labels & Selectors', which is described in the next section. We need to keep in mind that ConfigMaps are not encrypted and thus not recommended for storing sensitive data. Another Kubernetes object called "Secrets" are used for storing confidential information.

2.5.7 Secrets

Secrets are similar to ConfigMaps, but store small amount of confidential data such as passwords, ssh keys, OAuth tokens, etc.

2.5.8 Labels & Selectors

When we create Kubernetes objects, we sometimes work with Configuration files where we organize and create associations between certain objects. **Labels** are essentially key-value assignments attached to Kubernetes objects and used for grouping them, and **Selectors** serve to choose groups of objects with the same label [31]. Figure 2.6 is a Kubernetes Deployment object where the pods are labelled as "**app: nginx**". Figure 2.7 is a Kubernetes Service object that has the selector for all the Kubernetes objects having the label "**app: nginx**" as can be seen in lines 6 and 7. This suggests that the Service object (nginx-service) is attached to all pods with the label "**app: nginx**". Labels and Selectors are crucial parts of Kubernetes configuration files.

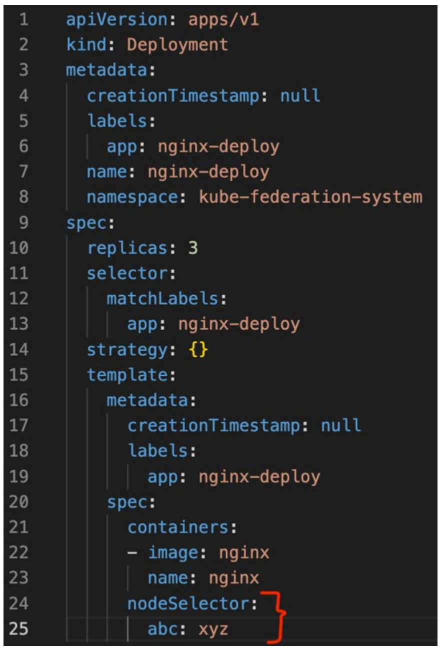
In our design & implementation section, we will assign workloads from a central cloud cluster to a local cluster in a branch (e.g., our local computer). In particular, we will assign workloads to a particular server (e.g., node) in our local cluster from a central cloud. To achieve this, we can limit the deployment of certain pods so that they can only run on specific set of node(s) using label selectors [32]. In general, Kubernetes doesn't expect such configuration as it has a scheduler automatically placing pods across different nodes based on the existing loads. However, we sometimes want to control which nodes we wish the pods to run at because these nodes may be located closer to a particular end user or we may want to co-locate multiple pods on the same exact node that are extensively communicating with each other [32]. In Kubernetes, we allocate workloads (e.g., pods) to particular node(s) in three main ways: **(i)** nodeSelector using node labels, **(ii)** Affinity and anti-affinity, **(iii)** nodeName field. The one that we will use in our design & implementation is the Kubernetes preferred way of selecting nodes: nodeSelector and node labels.

2.5.8.1 NodeSelector and Labelling Nodes

In general, when we wish to label nodes, we do it manually through a command below:

```
kubectl label node local-cluster-node-m01 abc=xyz
```

Here, we are labelling the node "**local-cluster-node-m01**" with the label of **"abc=xyz".** Remember that the labels are just key-value pairs. Using this label with a NodeSelector will place the workloads (e.g., pods) on any node with the **"abc=xyz"** label, in our case, just the "local-cluster-node-m01" node:



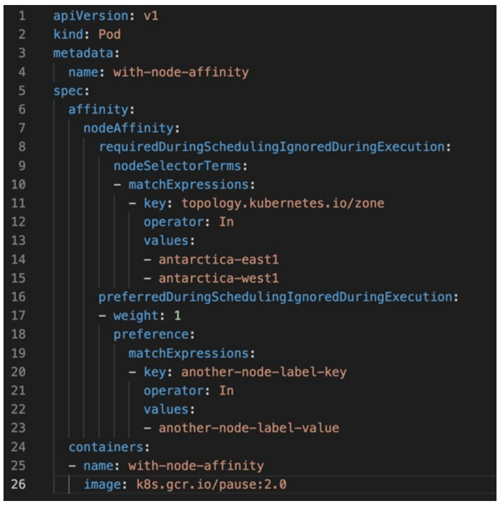
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: nginx-deploy
  name: nginx-deploy
  namespace: kube-federation-system
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx-deploy
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: nginx-deploy
    spec:
      containers:
      - image: nginx
        name: nginx
      nodeSelector:
        abc: xyz
```

Figure 2.8: Placing workloads on a particular node through Labels & Selectors

In Figure 2.8, notice that below the containers field, we have a '**nodeSelector'** field where we give the labels of nodes on which we want the pods to be deployed. In this example, we are deploying three Nginx pod replicas onto all nodes with "abc=xyz" label. We may want such a scenario if for instance the Nginx pod replicas need to communicate with each other extensively.

2.5.8.2 Affinity and Anti-Affinity

Node affinity is similar to the NodeSelector method, but it allows to give more specific conditions. While NodeSelector allows us to allocate pods to nodes using a simple 'key-value' pair label; Node Affinity allows us to give a logical set of conditions for the selection of nodes. Below is an example configuration file using Node Affinity from the official Kubernetes website [32]:



```yaml
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: topology.kubernetes.io/zone
            operator: In
            values:
            - antarctica-east1
            - antarctica-west1
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: another-node-label-key
            operator: In
            values:
            - another-node-label-value
  containers:
  - name: with-node-affinity
    image: k8s.gcr.io/pause:2.0
```

Figure 2.9: Placing workloads on particular Nodes through Node Affinity [32]

In Figure 2.9, in order for the pods to be deployed in the desired nodes, the nodes must have a label with the key= topology.kubernetes.io/zone and the value must be either of `antarctica-east1` or `antarctica-west1` . The node also preferably has a label with the key=another-node-label-key and the value of another-node-label-value [32].

2.5.9 Custom Resource Definitions (CRD)

A Kubernetes resource is an endpoint in Kubernetes API that allows you to store an API object of any kind. For instance, Service resources serve to store a collection of the Service objects. A Custom Resource helps create your own API objects and define your own kind such as Pods, Deployments, ConfigMaps, ReplicaSets, etc. [33 , 34]. Custom Resource Definitions or CRDs allow application developers to extend existing Kubernetes capabilities by adding new kinds of API objects useful for the application. CRDs are what we use to create Custom Resources and they allow these Custom Resources to be used as Kubernetes resources.

2.6 Summary

In this chapter, we gave an in-depth background of Kubernetes, why it is used, its components and main architecture. The information provided in this chapter will help us in the design & implementation section of our thesis. We learnt that Kubernetes is the leading tool for container orchestration that helps us automate the process of deployment, updates, development, and release of an application. So far, we have only focused on a single cluster setup in Kubernetes. In the next section, we will talk about "Multi-cluster Kubernetes" and touch upon the "Cluster Federation" concept. These two concepts are the two building blocks of our thesis.

Chapter 3

Kubernetes Multi-Cluster

In this chapter, we will introduce the Multi-Cluster Kubernetes approach, talk about its various use-cases and implementations, and summarize the benefits that come with it. After we draw a clear picture of the Multi-Cluster Kubernetes, we will introduce the concept of “Cluster Federation”, which is the foundation of this thesis. In the next chapter, we will talk about existing Kubernetes Federation Projects and particularly focus on KubeFed project, which is the open-source project with which this thesis aims to build a working solution on a cross-platform context (e.g., host cluster in cloud with a member cluster running on a local Minikube representing a branch of an application).

3.1 Multi-Cluster Approach in K8s

As mentioned in the previous chapter, a Kubernetes application is deployed in a cluster that includes multiple master and worker nodes responsible for hosting application pods with containers. In simple cases, a single Kubernetes cluster may be sufficient for a company’s business needs, however, it has multiple drawbacks [35]. **Multi-Cluster Kubernetes** is simply a strategy of creating an environment in which we use more than one Kubernetes cluster for application deployment. These clusters might reside on the same physical host, on various hosts in the same data center, or even in different clouds in different countries.

A **single cluster approach** is generally more cost-effective and facilitates administration of the Kubernetes resources, however, there are certain drawbacks to this strategy. As the name suggests, a single cluster may result in a single point of failure for the deployed applications. Some examples of such failures could be configuration errors, infrastructure outages, control planes going down, etc. [36]. Secondly, a single cluster most probably contains multiple running applications sharing the same resources (e.g., networking, OS, hardware, etc.) [36]. Although, Kubernetes namespaces may help to virtually isolate different applications running on the same cluster, this may not be enough to fully prevent the unwanted interactions between different applications, hence increasing the possibility of security risks and resource competition. Another similar example would be a scenario in which a company uses a single cluster to deploy all their applications. Different teams may be assigned to different applications, but they might still access other teams’ applications in the same cluster. This would obviously exacerbate possible attack surfaces. Moreover, a Kubernetes cluster can grow to host at most 5,000 nodes, 150,000 pods and 300,000 containers [36]. This amount of resource may not be necessary in most cases, however, it is still worth mentioning. The drawbacks mentioned in this paragraph pushed organizations to adopt a **Multi-Cluster Kubernetes approach**, so they can deploy their applications across multiple Kubernetes clusters for a higher availability and a better isolation. In the next section, we will give a more detailed overview of the benefits that come from Multi-Cluster strategy.

It should also be noted that, although we can add multiple nodes to a single cluster to offer some sort of high availability, a single cluster currently cannot have nodes from different regions. More specifically, all the nodes in a single cluster must be from the same region, however, they can span across different zones within a region. The safest way to deploy a Kubernetes application across different regions for high-availability and disaster recovery is to create multiple-clusters in different regions. This would allow the same application to serve multiple geographical regions [37]. We will implement such solution in the design & implementation section where one of the clusters will be our local Minikube cluster and the central cluster will reside in GCP Australian region.

3.1.1 Multi-Cluster Kubernetes Advantages

By deploying Kubernetes applications across multiple clusters, multi-cluster K8s aims to tackle the limitations mentioned in the previous section. Below are some of its advantages [38]:

- **Higher Availability and Performance**: Application developers can deploy different clusters across different geographical locations, providing geo-redundancy [36]. Same applications may be deployed in multiple clusters in different regions for high availability and low latency. It may be more practical to have different clusters in each location, while being able to centrally manage them. This is not a feature that a single cluster can offer since all the nodes in the cluster must be from the same region. Although, multi-cluster deployment allows us to deploy clusters across different regions and possibly cloud providers, if we cannot easily manage these clusters from completely different environments through a single controlling point, then our lives wouldn't get any easier with multi-cluster deployment either. The central management of different clusters is also a foreshadowing to the 'Cluster Federation' concept that this thesis revolves around. A multi-cluster deployment across different regions and cloud providers without a single federation control manager might quickly become a nightmare.

- **Tenant Isolation**: As suggested in the introduction, it is challenging to work with multiple Kubernetes environments to handle development, testing, and production settings in the same cluster. Different teams can be assigned to different projects (e.g., production or testing, etc.) in the same cluster, and the isolation between the applications running in different projects within the same cluster can be somewhat provided through namespaces. However, the security mechanism of Kubernetes makes it challenging to segregate different environments from one another. A problematic application running in one namespace can still affect applications running in different namespaces but still sharing the same hardware, etc. Kubernetes multi-cluster setups ease the process of isolating users and projects by cluster.

- **Failover**: With a multi-cluster system, application developers make sure that workloads do not encounter downtime caused by a fault within a single cluster by effortlessly relocating them to different clusters [35].
- **Avoiding Vendor Lock-In**: Through a multi-cluster setup, application developers can deploy their clusters on different cloud providers of their choices, fully utilizing different capabilities and pricings offered by multiple cloud providers. For instance, in our design section, we will have one of the clusters in our 'local branch' running in Minikube.
- **Scaling Applications beyond a Single Cluster's Limits**: As mentioned in the introduction, there is a limit to the number of nodes, pods and containers deployed on a single cluster. By deploying applications in multiple clusters, you can increase the computing power even further. This would work best with a Multi-Cloud Kubernetes environment.
- **Edge Computing and IoT**: With multi-cluster approach, we can deploy local clusters responsible for processing data at the edge and send the output to regional clusters. These regional clusters may also perform certain operations and return the output to a central cluster. In an IoT setting, the local clusters at the edge may include smaller compute devices acting as "leaf nodes" for the cluster.

3.1.2 Multi-Cluster Kubernetes Architecture

As we see in the work of Saba Feroz Memon from Aalto University [36] and the descriptions made in several company products [39, 37], there are several use-cases for the Multi-Cluster Kubernetes setup, requiring different types of architectures as can be seen in Figure 3.1. Below we will summarize some of these use-cases:

- **Replicating Clusters**: In this setup, one of the clusters is the 'main' cluster (or host cluster in Cluster Federation terms) responsible for hosting the main applications; the other clusters are replicas that have the copies of the host cluster's applications. This approach provides high-availability and disaster recovery by allowing the same application to run in different geographical locations. The clients' requests can be routed to the geographically nearest cluster to decrease latency. This setup will be a part of our design & implementation section through "cluster federation".
- **Cluster per Deployment Unit**: In this setup, we have separate clusters for each deployment unit of an application (e.g., development, testing, production). Imagine we have 3 different applications with each having production, development, and testing environments. In this scenario, we would have three separate clusters each containing all 3 applications, but only one of the three deployment units (dev, testing, or production). In this setup, an entire development, testing, or production team can work on their own cluster for all three applications. There is an isolation of deployment units rather than the isolation of applications. Refer to Figure 3.1 for a clearer picture.
- **Cluster per Application**: In this setup, rather than having a cluster per deployment unit, we have a cluster per application. Again, imagine we have 3 applications. In this case, each cluster would contain only the deployment units of a single application. A Single Cluster would contain all the development, testing, and production deployment units of

a single application. There is an isolation of applications rather than the isolation of deployment units. This method allows for the adaptation of each cluster based on the application they are hosting. Refer to Figure 3.1 for a clearer picture.

- **Cluster per both Application and Deployment Unit**: This strategy is an amalgam of the previous two setups. Imagine we have 3 different applications and 3 different deployment units (e.g., development, testing, production). In this scenario, we would have a separate cluster per application per deployment unit, giving us a total of 9 separate clusters. This allows for a total isolation, reducing the effect of an event on production unit.

  •**Cluster per Application Features**: This strategy deploys a cluster per different features of a single application. This helps developers to work on specific features of a mission-critical application separately, hence isolating between an application's different components for a more focused and secure feature development.

Below are representations of above architectures influenced by the work of Saba Feroz Memon [36]:

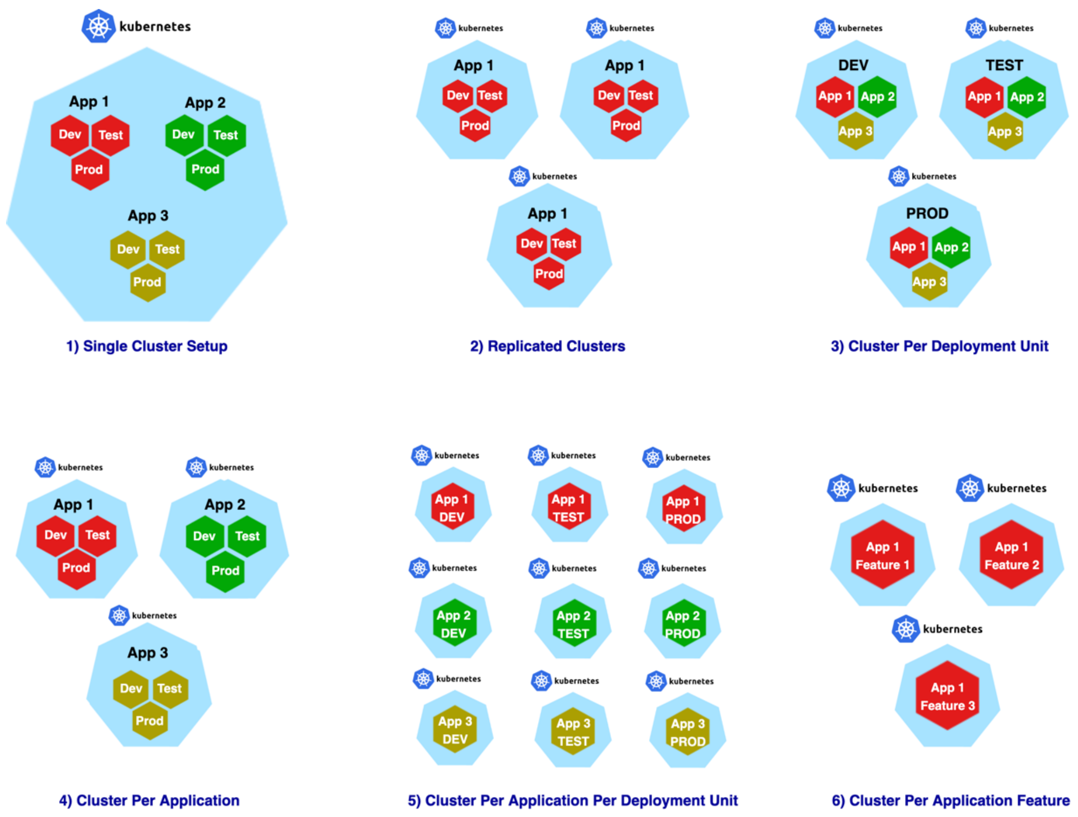

1) Single Cluster Setup
2) Replicated Clusters
3) Cluster Per Deployment Unit
4) Cluster Per Application
5) Cluster Per Application Per Deployment Unit
6) Cluster Per Application Feature
kubernetes
App 1
App 2
App 3
Dev
Test
Prod
DEV
TEST
PROD
Feature 1
Feature 2
Feature 3

Figure 3.1: Different Types of Multi-Cluster K8s Architectures

3.2 Kubernetes Cluster Federation

In the previous section, we have defined what the 'Multi-Cluster Kubernetes' means and its various use-cases. We have described its benefits and why most companies are following the multi-cluster strategy. We emphasized that Multi-Cluster Kubernetes is generally used for providing High Availability, Low Latency, Failure and Business Isolation and avoiding Vendor Lock-in. We mentioned that this isolation could be partially provided in a Single-Cluster scenario by creating different namespaces and deploying the resources separately. However, we also pointed out that this does not offer a secure isolation compared to deploying applications on different clusters. We've also talked about how a single cluster cannot have nodes from different regions. Thus, deploying a Kubernetes application across multiple clusters helps the application span multiple geographical regions for high availability and disaster recovery.

Multi-cluster deployment solves a range of difficulties, as shown by the aforementioned examples. However, it increases the complexity of operation and maintenance as well. In a single cluster scenario, application deployment and upgrades are much less complicated as the developer can easily modify or update the YAML of the cluster. Changing application configuration and watching it propagate in a single cluster is much more straightforward than a multi-cluster setting. If we are dealing with multiple clusters, we can modify the YAML files one by one for each cluster, but the trick is to make sure that the application load status is the same across different clusters. This would be even more challenging if the clusters are from completely different regions and/or cloud providers. In these scenarios, it would be quite difficult to manage all these clusters from different environments. How can we implement service discovery or load-balancing among different clusters that we are managing? These are the challenges that **Kubernetes Cluster Federation** aims to solve for Multi-Cluster Kubernetes through a single point of management [40, 41]

3.2.1 Cluster Federation Description and Benefits

Kubernetes Cluster Federation is a strategy to manage the configuration of multiple Kubernetes clusters through a single point called "federation control manager" [70]. Deployers have the option to choose which clusters the manager will coordinate and what their configurations will look like through a single group of APIs. This process helps the application deployer to manage multi-cluster deployments more smoothly and transparently [40]. Through a single point, the application deployer can deploy applications across multiple clusters and watch the deployment status of all the federated resources. Most Cluster Federation tools allow the deployer to select whether certain clusters have a particular configuration or all of them share the exact same resources, a concept which we will visit in the next chapter (e.g., KubeFed). Two important components of Cluster Federation are:

- **Syncing Resources Across Clusters**: This is the fundamental challenge that application deployers encounter when they deploy applications across multiple clusters (especially across different regions and cloud providers). Through Cluster Federation, deployments on multiple clusters possibly running on different cloud providers can automatically sync.
- **Intercluster Discovery**: Ability to automatically configure DNS servers and Load-Balancers with backends, discovering all member clusters possibly running on different cloud providers.

Below is a sample Cluster Federation setup with two federated member clusters, each from different regions.

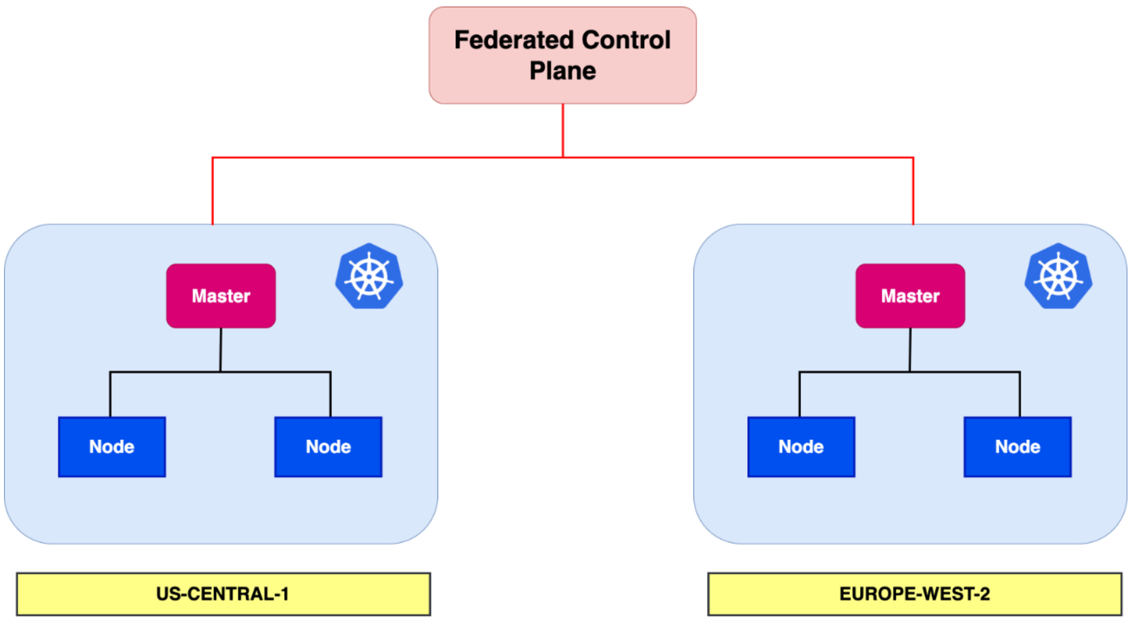


Figure 3.2: Federation of two clusters from different regions through a Manager

Imagine that the organization has two separate clusters, one in us-central1 and the other in europe-west-2. They may both run on the same cloud provider or completely different environments. Through a single Federation Manager, the same configuration can be easily applied to both clusters in different regions. It facilitates the coordination of the application configuration and deployment across multiple clusters. Cluster Federation helps the coordination of clusters from completely different regions and/or cloud providers and helps responsiveness and resiliency of the applications on federated clusters [36, 71]. The application's configuration is applied to the Federation Manager, and it schedules the deployment with the exact same configuration across the clusters. If the cluster in us-central1 fails, the same application will still be available in europe-west-2 or vice versa. However, keep in mind that if the Federated Control Plane gets down, the already deployed applications will continue to work, but we won't be able to federate them or check the applications' desired states from a single point.

Overall, **Kubernetes Cluster Federation** provides 'ease of configuration to the desired state'. From a single group of APIs, the deployer can make sure that a ReplicaSet or Deployment is configured to execute a desired number of pods across multiple clusters, etc. Federation also complements the high-availability feature of multi-cluster deployment with ease of management feature since Federation facilitates the coordination of clusters running the same application across different geographical regions and cloud providers [42].

3.2.2 Cluster Federation Caveats

However, nothing comes with only advantages. There are many good reasons to use Kubernetes Cluster Federation as described in the previous sections, but there are also several things to consider [42]:

- **Higher Network Bandwidth and Associated Costs**: The Federation Manager monitors all the member clusters to make sure that they stick to the desired current state. In most cases, we will be running our clusters in multiple regions or possibly different cloud providers, which can substantially increase the networking costs. In many cases, conventional approaches to Cluster Federation (e.g., native cluster management found in most public cloud platforms) may be much less costly.

- **Limited Cross-Cluster Isolation**: As previously described, a single Federation Manager is responsible for the configuration and deployment of applications across multiple clusters. Thus, if a bug appears in the control plane, it has the potential to affect all the clusters.

- **Lack of Maturity in Current Federation Projects**: Most of the Cluster Federation projects are recent and do not support all resources. Most of the projects are active in development process as we will discover in the next chapter. Moreover, Cluster Federation tools generally do not provide their own security mechanisms, thus most security solutions must be custom integrated. Another missing feature of Cluster Federation tools that we will focus on in our design & implementation section is their lack of support for creating a hybrid-environment with on-premises and cloud settings (e.g., central cluster in the cloud and the member cluster federated on a local machine). We will create such an environment for our implementation in the upcoming chapters.

3.3 Other Alternatives

Kubernetes Cluster Federation is not the only means to manage multiple Kubernetes clusters. There are two other well established strategies that companies choose to use: **the Cluster API**, currently a Kubernetes sub-project, and **Google Anthos**, a Kubernetes solution offered by Google [9]. These are not particularly used for Cluster Federation, however, they are widely used for Cluster Management.

3.3.1 Cluster API

Cluster API [43] is a Kubernetes Open-Source project that aims to help cluster operators simplify cluster lifecycle management. It hopes to answer the question of: “what if we were to use Kubernetes itself to manage Kubernetes?”. As we know, Kubernetes provides declarative APIs for managing workloads. What if we were able to create declarative APIs for managing cluster lifecycle, and that’s what the Cluster API does. Cluster API provides Kubernetes style APIs and patterns (declarative APIs) that allow a cluster operator/administrator to declaratively define what a cluster should look like and then have Cluster API reconcile that declarative definition. Kubeadm is a crucial tool used by Cluster API to reuse and integrate existing ecosystem components rather than recreating them from scratch. Cluster API mostly serves to scale up and down existing workload clusters, unlike KubeFed tool which aims to federate configurations across member clusters. Cluster API aims to facilitate the creation and replication of node infrastructure across clusters whereas KubeFed aims to help with the replication of Kubernetes resources such as deployments and services across clusters. Cluster API has two types of clusters: a Management Cluster and a Workload Cluster. The Management Cluster is responsible for managing the lifecycle of the Workload Clusters just like how a single cluster is responsible for managing the lifecycle of its nodes. Through Cluster API, we can check the lifecycles of nodes in multiple workload clusters from a single management point.

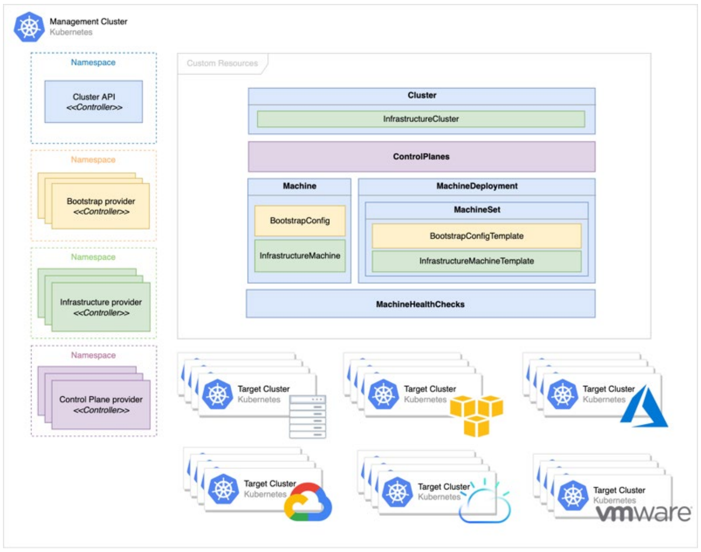
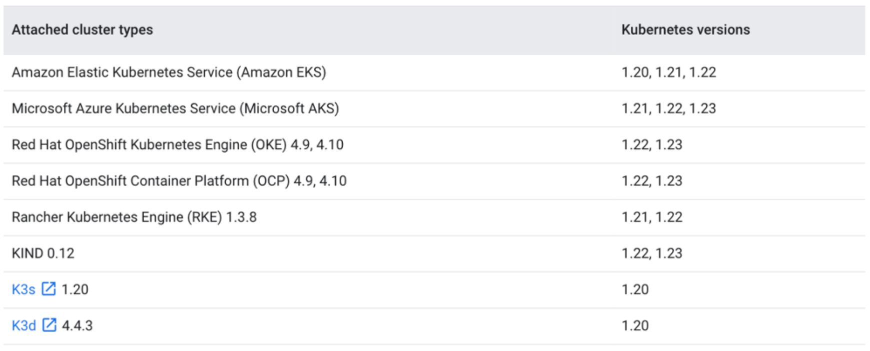


Figure 3.3: Cluster API Components

3.3.2 Other Tools Offered by Cloud Providers

There are some other Cluster Management tools provided by some of the biggest Cloud Providers. Some of these tools only work within their own cloud provider and not across different providers, but most of them offer cross-platform features. Some of these tools are

Google Anthos, **Red Hat's OpenShift**, **AWS outposts** and **Azure Arc**. Among them, Google Anthos is the most flexible and mature cluster management tool, however, it is not a Cluster Federation tool. It helps application deployers view all the managed clusters from a single point called Anthos. Most of the Cluster families (e.g., EKS, GKE, VMware, OpenShift) are supported by Google Anthos. Google Anthos allows us to attach agreed upon third party clusters to our fleet and lets us view our existing third-party clusters along with our Anthos clusters. From a single point of management (e.g., UI with a Dashboard), Anthos provides a centralized configuration control with Anthos Config Management and Microservice architecture management through Anthos Service Mesh [44]. Below are the conformant Kubernetes clusters that we can add to our management using Anthos:



Attached cluster types	Kubernetes versions
Amazon Elastic Kubernetes Service (Amazon EKS)	1.20, 1.21, 1.22
Microsoft Azure Kubernetes Service (Microsoft AKS)	1.21, 1.22, 1.23
Red Hat OpenShift Kubernetes Engine (OKE) 4.9, 4.10	1.22, 1.23
Red Hat OpenShift Container Platform (OCP) 4.9, 4.10	1.22, 1.23
Rancher Kubernetes Engine (RKE) 1.3.8	1.21, 1.22
KIND 0.12	1.22, 1.23
K3s 1.20	1.20
K3d 4.4.3	1.20

Figure 3.4: Conformant Kubernetes Clusters we can add to Anthos Management

As mentioned before, Google Anthos is not a Federation tool where we select one of the clusters as the host-cluster and the others as member clusters. Resources are not automatically propagated to a member cluster through a host-cluster. Rather, Anthos is an enterprise solution that helps us configure all our registered clusters without needing to leave its platform. Anthos Config Management offers many advantages since it seamlessly synchronizes configurations and policies across various clusters. Below are some of the important features and advantages that Google Anthos brings:

- **Simplified Management of Clusters**: From a simple platform, we are able to manage (e.g., deploy configurations, policies, etc.) across clusters from different environments.

- **Consistent Configurations and Policy Management**: Anthos provides an auditable version control system to manage and keep track of all the configurations of our clusters.

- **Scalable across Environments**: Centralizing the configuration and management of multiple clusters from different environments using a single platform, Anthos provides automated, scalable and resilient ways to manage complex systems.

- **Secure and compliant**: User can define a single set of policies and can be assured that the security policies will be consistently applied across all the environments registered to Anthos. Moreover, Anthos Config Management continuously monitors the states of the clusters and make sure that the policies correspond to the application deployer's desired policies.

Google Anthos is a perfect tool to manage any type of cluster in Figure 3.4 through a single platform. We can also see that Anthos is a strong production ready tool compared to KubeFed, which is still in development. However, Anthos doesn’t perform “Cluster Federation” in the sense that regular federation projects like KubeFed does, and it doesn’t support all cluster types and Kubernetes versions. For simple cluster Federation solutions, KubeFed may have more advantages since it is only designed for resource federation, and it is environment agnostic. Moreover, it is much easier to set up than Anthos. However, if we need a more robust solution that considers all security issues and requires a platform to watch all the configurations of multiple clusters from different environments, Anthos would be a much better choice. We need to consider the needs of our solution and choose an appropriate tool accordingly.

3.4 Summary

In this chapter, we have talked about the concepts that this thesis will focus on: Kubernetes Multi-Cluster deployment and Kubernetes Cluster Federation. We defined what Multi-Cluster deployment refers to and some of its benefits such as: high availability and performance, tenant isolation, fighting failover, avoiding vendor lock-in, scaling applications beyond a single cluster’s limits, etc. We then provided different architectures for Multi-Cluster Kubernetes based on different use-cases. We also mentioned certain drawbacks of multi-cluster deployment such as complexity of operation and maintenance compared to a single cluster scenario. We then introduced the main concept of our thesis: Kubernetes Cluster Federation and how it provides ease of management for multi-cluster settings through a single set of API points. However, most of the cluster federation tools are still in development process and have certain important caveats that need to be considered as well. In the next section, we will introduce certain Federation Projects, putting our emphasis on a particular one called “KubeFed”.

Chapter 4

Cluster Federation Projects

In the previous chapter, we have introduced the concept of Multi-Cluster Kubernetes and how it is used in production. We provided the most commonly used multi-cluster architectures and their benefits compared to a single cluster setting. We have also briefly mentioned the limitations and drawbacks of multi-cluster Kubernetes and how Kubernetes Cluster Federation may help mitigate such drawbacks. We gave a sample diagram showing how Cluster Federation generally works in Kubernetes and also provided its benefits together with its limitations. We have also offered a small section on Cluster Federation alternatives provided by the major cloud providers. In this chapter, we will talk more in detail about the current cluster federation tools, particularly emphasizing on the KubeFed project, which is the heart and soul of our thesis's design and implementation.

4.1 KubeFed Project (Version 2)

KubeFed is the official Kubernetes Cluster Federation implementation built by a Kubernetes Special Interest Group (SIG) [45]. KubeFed enables users to coordinate the configurations of several Kubernetes clusters from inside a Host cluster through a single set of APIs. In most cases, KubeFed and the Kubernetes Cluster Federation concept are used interchangeably. In the next section, we will provide a detailed overview of KubeFed and its many benefits.

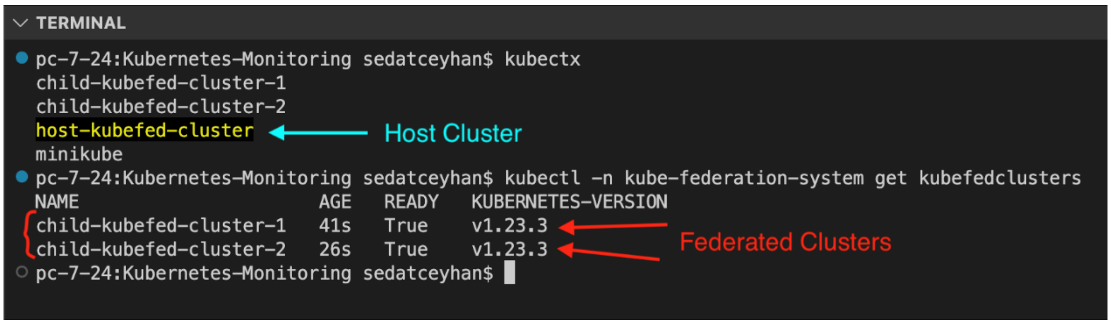
4.1.1 KubeFed Detailed Overview

KubeFed helps us synchronize resources from a central 'Host Cluster' to the 'Federated Clusters' that can be hosted in any regions and/or cloud providers, a property that a vanilla multi-cluster Kubernetes deployment lacks. This also corresponds to the term frequently used in the project: "propagation". It corresponds to distributing resources from the host cluster to all the federated member clusters. A single Configuration File (YAML) is created and executed in the Host cluster, which propagates to all, or selected member clusters based on the configuration of the YAML file. This means that the KubeFed Host Cluster allows the app deployer to deploy applications across multiple clusters through a single set of API points.

There are two types of clusters in KubeFed:

•**Host Cluster**: Cluster that is responsible for exposing the KubeFed API and run the Federation Control Plane to federate applications across member clusters. A host cluster can also be a member of its own. We schedule federated resources in the Host Cluster and see these resources automatically deployed as regular resources in the Member Clusters.

•**Member Cluster**: Cluster that is registered through the KubeFed API. Member clusters are federated by the application deployer through the Host Cluster. When we schedule federated resources in the Host Cluster, the resources are automatically deployed in the Member Clusters.



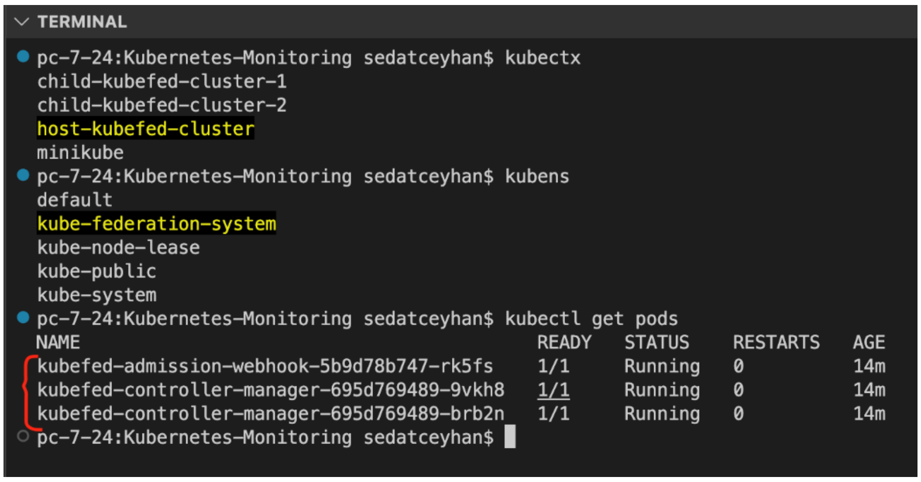
```
TERMINAL
pc-7-24:Kubernetes-Monitoring sedatceyhan$ kubectx
child-kubefed-cluster-1
child-kubefed-cluster-2
host-kubefed-cluster
minikube
pc-7-24:Kubernetes-Monitoring sedatceyhan$ kubectl -n kube-federation-system get kubefedclusters
NAME                      AGE   READY   KUBERNETES-VERSION
child-kubefed-cluster-1   41s   True    v1.23.3
child-kubefed-cluster-2   26s   True    v1.23.3
pc-7-24:Kubernetes-Monitoring sedatceyhan$
```

Figure 4.1: List of Federated Clusters Observed from Host Cluster

The Host Cluster includes two crucial components called **KubeFed API** and **KubeFed Controller Manager**. These are the building blocks for the KubeFed. When KubeFed is installed to the Host-Cluster, these two components are automatically deployed as pods in a namespace of our preference. Refer to Figure 4.1 where all KubeFed controller manager and API servers are deployed as pods in host-cluster's "kube-federation-system" namespace that we created:

•**KubeFed API Server**: The KubeFed API Server is exposed by the host cluster. The KubeFed Controller Manager uses the API server to communicate with other clusters. This makes the KubeFed API server quite similar to the Kubernetes API server in that, the former helps the KubeFed controller manager manage and coordinate the member clusters; the latter helps the Kubernetes control manager to query, create, delete Kubernetes resources [36].

•**KubeFed Controller Manager**: KubeFed Controller Manager is very similar to the Kubernetes Controller Manager in that Kubernetes Controller Manager constantly monitors the state of pods, deployments, nodes within a single cluster whereas the KubeFed Controller manager monitors the state of all the federated clusters to preserve the desired state of the member clusters. In order to synchronize the state of all the Federated resources with those deployed in the member clusters, the KubeFed Controller Manager uses a push reconciler strategy [36].



```
TERMINAL
pc-7-24:Kubernetes-Monitoring sedatceyhan$ kubectx
child-kubefed-cluster-1
child-kubefed-cluster-2
host-kubefed-cluster
minikube
pc-7-24:Kubernetes-Monitoring sedatceyhan$ kubens
default
kube-federation-system
kube-node-lease
kube-public
kube-system
pc-7-24:Kubernetes-Monitoring sedatceyhan$ kubectl get pods
NAME                                          READY   STATUS    RESTARTS   AGE
kubefed-admission-webhook-5b9d78b747-rk5fs    1/1     Running   0          14m
kubefed-controller-manager-695d769489-9vkh8   1/1     Running   0          14m
kubefed-controller-manager-695d769489-brb2n   1/1     Running   0          14m
pc-7-24:Kubernetes-Monitoring sedatceyhan$
```

Figure 4.2: KubeFed components automatically deployed as Kubernetes Pods

Moreover, KubeFed uses two kinds of configuration information as mentioned in [45]:

•**Cluster Configuration**: The member clusters that we wish KubeFed to target [45]. This basically includes the register information required for the KubeFed Control Plane to add member clusters.

•**Type Configuration**: To define the types of APIs to be handled by KubeFed [45]. Each Type Configuration is a Custom Resource Definition (CRD) object that includes three configuration components: **templates**, **placement**, **overrides**. Below we will describe these components as they are important for our design & implementation.

Coming back to the concept of propagation, corresponding to the mechanism for distributing resources to Federated Member clusters, there are three main components of resource propagation from a host cluster to the member clusters. These components reside in a configuration file (YAML) of a federated resource. The configuration file containing these components is applied within the Host-Cluster as a FederatedDeployment, FederatedService, etc., and they are deployed as regular Deployments or Services in the member clusters:

•**Templates**: Similar to the basic Kubernetes configuration files in YAML, the template describes basic information of the resource to be federated among the member clusters (e.g., Deployment, Service, DaemonSet) [45]. The template may include the image of the deployed container, env. variables and the number of instances [46]. Template needs to include all the information necessary to create the deployment.

•**Placement**: The ‘Placement’ section of the configuration file includes the federated member clusters that will contain the deployed resources. Not every federated cluster needs to include the deployed federated resources. The application developer may choose to opt out certain member clusters [46].

•**Overrides**: Override section of the configuration file is responsible for overriding the features of the resources in the original template to match certain specific criteria of a member cluster. The overridden conditions can be the number of replicas, secret-keys, etc.

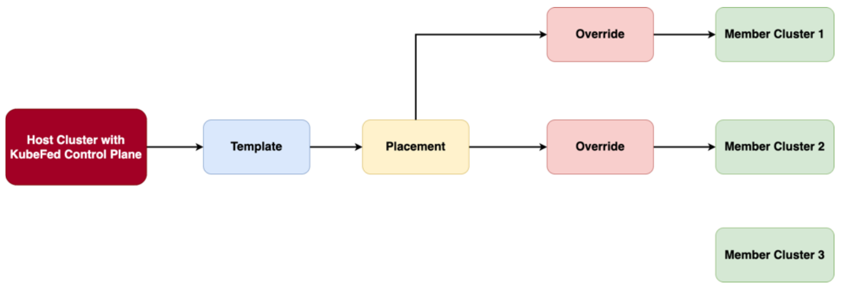
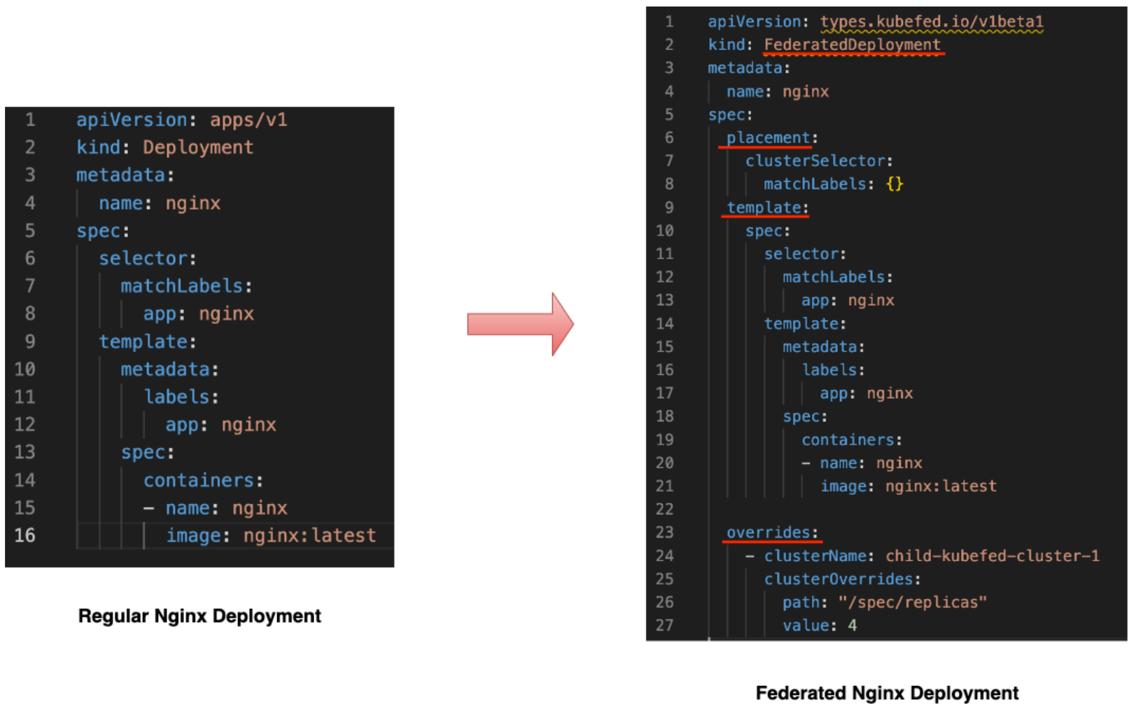


Figure 4.3: Sample KubeFed Template Diagram

Below is a comparison between a regular Nginx deployment and the Federated version of the same Nginx deployment:



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
```

Regular Nginx Deployment

```yaml
apiVersion: types.kubefed.io/v1beta1
kind: FederatedDeployment
metadata:
  name: nginx
spec:
  placement:
    clusterSelector:
      matchLabels: {}
  template:
    spec:
      selector:
        matchLabels:
          app: nginx
      template:
        metadata:
          labels:
            app: nginx
        spec:
          containers:
          - name: nginx
            image: nginx:latest

  overrides:
    - clusterName: child-kubefed-cluster-1
      clusterOverrides:
        path: "/spec/replicas"
        value: 4
```

Federated Nginx Deployment

Figure 4.4: Conversion of Deployment resource into Federated Deployment

The first thing that we do in KubeFed is to create a Federated Namespace resource in the Host Cluster. This Federated Namespace is just a regular namespace that will be created in all the member clusters, and whenever the host cluster creates federated resources in this namespace, they will be deployed in the member clusters as well. The right-hand side of the Figure 4.4 is an example federated resource that we create in the federated namespace of the host cluster, which will force the deployment of this resource among the same federated namespace of the member clusters. Below, we will give more detail on how this all works.

In order to create **regular Nginx Deployments** in member clusters (the deployment configuration on the left-hand side of Figure 4.4), the host cluster will deploy a **FederatedDeployment** configuration as can be seen on the right hand-side of Figure 4.4. This FederatedDeployment of Nginx, in fact, corresponds to a regular Nginx deployment on the left-hand side. Moreover, this Federated Nginx Deployment is deployed in the FederatedNamespace of the Host Cluster (a namespace that appears in both the host cluster and all the member clusters where all the federated resources are created). What we see deployed on the member clusters is the regular Nginx deployment on the left-hand side of Figure 4.4. The FederatedDeployment configuration file is for the host cluster to give certain specifics to the deployment. We deploy the FederatedDeployment on the right-hand side of figure 4.4 in the host cluster, and this will automatically create regular Deployments on the left-hand side of figure 4.4 in the member clusters. Unless the host cluster is also federating itself (namely the host cluster is a member cluster of itself as well), the deployments will not be created in the host cluster; the FederatedDeployment would only propagate deployments to the member clusters. Notice the underlined sections: **placement**, **template**, and **overrides**. A few things are happening in this configuration file:

- **Placement**: Under the placement field, we are selecting all the member clusters for the propagation of this particular Nginx deployment. By leaving the “matchLabels” section empty, we are choosing all the member clusters.

- **Template**: The template section is just the information required for the Nginx deployment. Notice that this section (L.10 – L.21) is identical to the regular Nginx deployment on the left-hand side (L.5 – L.16).

- **Overrides**: Remember that this section corresponds to overriding certain requirements for the deployment. In this case, we are only changing the deployment for the “child-kubefed-cluster-1” member cluster by overriding its number of Nginx pod replicas to 4.

The final three components are the building blocks for higher-level APIs:

- **Status**: Describes the status of the resources that are distributed across the federated member clusters.

- **Policy**: Describes the subset of member clusters that can include the federated resources.

- **Scheduling**: Describes how the workloads are distributed across the federated member clusters.

The diagram below provides a big picture of how all the KubeFed components function together [45]:

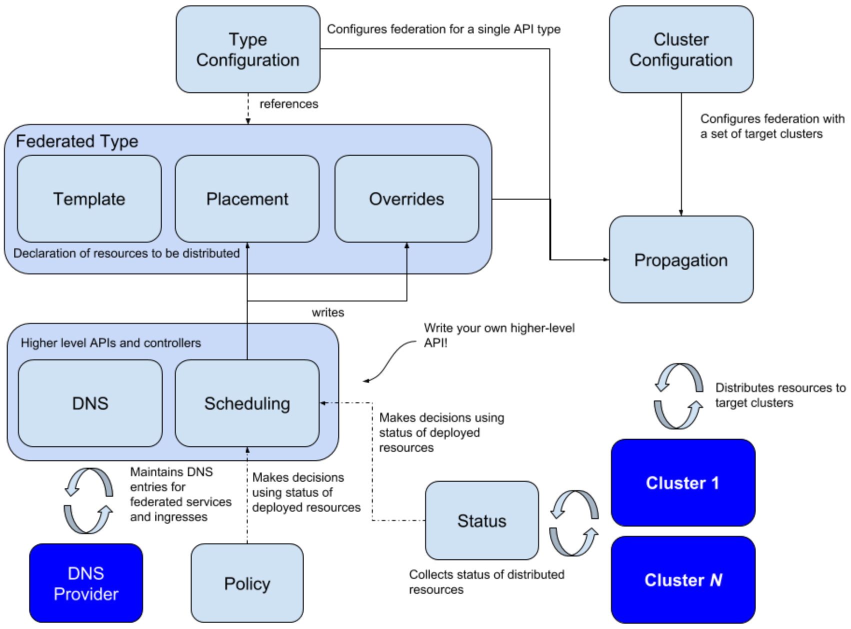

Type Configuration
Configures federation for a single API type
Cluster Configuration
references
Configures federation with a set of target clusters
Federated Type
Template
Placement
Overrides
Declaration of resources to be distributed
Propagation
writes
Higher level APIs and controllers
Write your own higher-level API!
DNS
Scheduling
Distributes resources to target clusters
Makes decisions using status of deployed resources
Maintains DNS entries for federated services and ingresses
Makes decisions using status of deployed resources
Cluster 1
Status
Cluster N
Collects status of distributed resources
DNS Provider
Policy

Figure 4.5: Main KubeFed Components [45]

The advantages and caveats mentioned in the Cluster Federation chapter applies to KubeFed as well. Refer to the previous chapter for the general benefits and drawbacks of KubeFed. In the next section, we will talk about some of the disadvantages specific to KubeFed that we encountered during our own experiments.

4.1.2 KubeFed Caveats

We have previously mentioned that the host cluster in KubeFed monitors all the member clusters to make sure that they are all in the desired states. Running these multiple clusters in different cloud providers or regions increase the networking costs dramatically. Thus, increasing the number of clusters in different regions would also increase the cost of communication between the host cluster and member clusters [47].

Another interesting limitation in KubeFed is its lack of automatic resource allocation among member clusters. As of now, KubeFed only allows the selection of member clusters manually to deploy the federated resources (e.g., placement section of the federated configuration files). We manually give our selection of member clusters for particular resources, and they are deployed in the member clusters. KubeFed does not really support automated policy-based scheduling. The host cluster basically propagates resources to the member clusters without checking in advance whether the target clusters actually have the necessary resources (e.g., storage, ram, etc.) left to host these federated resources. This is a big drawback that causes inefficient use of resources, thus KubeFed is not ready to scale to handle thousands of clusters expected in most multi-cluster use-cases (e.g., fog computing).

We conducted a small test on GCP using free credits. We created a host cluster with KubeFed and two member clusters each with a master node together with two worker nodes. One of the member clusters is created with nodes with higher CPU and RAM compared to the other cluster. We observed that the cluster with low resources was over-allocated with respect to its nodes' total CPU, and the cluster with higher resources was under-allocated. This happened because KubeFed does not check the cluster's available resources before deploying pods. It does not compare the member clusters' resource status before distributing pods among them. The scheduling and distribution of pods among clusters are purely manual.

Another issue with KubeFed that is discussed in detail in the work of Saba Feroz Memon from Aalto University [36] is KubeFed's single point of failure. We previously discuess how the federation of KubeFed works. We have a host cluster federating resources across the member clusters. All the Federation control plane (e.g., Federation Controller Manager pods and API webhooks in Figure 4.2) are deployed in the host cluster. If the host cluster fails, the federated resources that have already been deployed in the member clusters will continue exist with no issue, but they will no longer be federated since all the federation controller manager pods lived in the failed host cluster. This presents a single point of failure in KubeFed.

In the next sections, we will talk about some other Federation Projects that are not as popular as KubeFed. Moreover, some of these projects are no longer in use.

4.2 ClusterMesh Cilium

Cilium is a networking tool that offers network connectivity and load balancing between applications running on Kubernetes containers; ClusterMesh is an extension of Cilium that offers pod IP routing, service discovery, network policy enforcement and other networking plugins between multiple clusters [48]. ClusterMesh is essentially the multi-cluster implementation of Cilium. Clusters in a ClusterMesh have all their own etcd servers responsible for keeping track of cluster states and talking with other clusters through their proxies. The official documentation of ClusterMesh Cilium [49] states that the greatest benefit offered by ClusterMesh is high availability of Clusters. The deployed services on one cluster can be replicated onto other clusters belonging to different regions. When one of the clusters fails, the requests can be forwarded to the other replica. The other benefit stated in the official document is that multiple tenants can be provided shared services through ClusterMesh. Finally, ClusterMesh can facilitate the migration of workloads from one Cloud provider to another by splitting stateless and stateful services across different clusters. However, ClusterMesh is not as flexible as KubeFed since it requires Cilium-managed etcd for the creation of certificates and management of compaction. Moreover, ClusterMesh needs Cilium as its "Container Network Interface" and it is more challenging to set up as opposed to other Container Network Interfaces [50].

4.3 Network Service Mesh

There are certain limitations to Kubernetes networking. K8s networking cannot provide advanced L2/L3 network features and cannot meet some of the dynamic reuqirements of pods; most importantly, it lacks support for cross-cluster, multi-cloud and hybrid-cloud connectivity. These are not issues encountered by an every- day Kubernetes users but may as well become limitations for big projects. Kubernetes networking features can't quite meet the needs of Network Function Virtualization. Network Service Mesh (NSM) is a Cloud Native Foundation project that offers sophisticated L2/L3 networking capabilities for Kubernetes applications. Network Service Mesh does not interact with the Kubernetes Container Networking Interface; rather, it is a completely independent mechanism that comprises of many components that may be deployed inside or outside of a Kubernetes cluster. It is a multi-cloud/hybrid-cloud network solution that is native to the cloud. It is a network service providing networking functionalities (e.g., routers, firewalls, VPN gateways, etc.) providing features at L2/L3 layers. Network Service Mesh offers L2/L3 connectivity (e.g., VPN, firewall) on applications running in Kubernetes clusters [51]. Most of the advanced network features that Network Service Mesh offers are not provided by default Kubernetes Container Networking Interface plugins. It essentially aims to offer holistic networking features to Kubernetes clusters. The problem is that Network Service Mesh does not provide federation or orchestration features out of the box. It is mostly restricted to networking capabilities only.

4.4 Submariner

Kubernetes deployments include network virtualization allowing containers running on multiple nodes within the same cluster to communicate with each other. However, containers running in different Kubernetes clusters require additional resources (e.g., NodePorts, Ingress controllers). Submariner is a Cloud Computing Foundation Sandbox project aiming to connect overlay networks of different Kubernetes clusters. It essentially provides a secure and fast connectivity between different Kubernetes clusters through network routes and tunnels. This allows network communications among applications running on different clusters without us needing to create NodePorts, ingress controllers or load balancers [52]. It creates the necessary tunnels and routes needed to enable containers from different clusters to communicate directly. The most important benefits of Submariner are its disaster recovery, high availability and fault tolerance through the inter-communication feature between pods/services running on multiple clusters across on-premises and cloud environments. Submariner's architecture consists of several components: a Broker, Gateway Engine, Service Discovery, Globalnet Controller, Network Plugin Syncer and Route Agent [52].

Again, Submariner is more of a holistic networking service for Kubernetes rather than a seamless Cluster Federation tool. Moreover, it is not quite easy to set up Submariner as it may not be compatible with all Kubernetes plug-ins, requiring additional configurations.

4.5 Consul

Consul is a service mesh aiming to provide communications between microservices [36, 53]. The main features that Consul offers are service discovery, health checks for cluster health and encrypted traffic through TLS and support for multiple data centers. It uses its own key-value store for storing cluster data. Similar to KubeFed, Consul also offers Federation features through Consul Federation [54]. It can be seen that Consul Federation has more support and offers a more resilient Federation solution compared to KubeFed; however, it requires more configurations to have a working set up. In KubeFed, we install all the requirements on the host-cluster and do not involve the member clusters in the initial setup at all. On the other hand, with Consul, we have to install the tools and dependencies on each "secondary cluster" in the federation as well. In addition to the primary cluster's (e.g., host cluster in KubeFed) config file, every secondary cluster (e.g., member cluster in KubeFed) requires a separate Helm Config file as well since they have different settings. One can observe that Consul requires a more intricate setup process, and the updates in clusters may not be automatic. However, we should note that Consul provides a better documentation than KubeFed. It also offers a UI to observe the federated clusters in one dashboard. The reason we chose to work with KubeFed in our thesis is that KubeFed is the precursor of Kubernetes Federation projects, and it doesn't use any third-party tools for network connectivity within Kubernetes. It relies solely on Kubernetes Container Network Interface and other native Kubernetes resources.

4.6 Shipper

Shipper is a now discontinued project that has a lot of similar features to KubeFed. Like KubeFed, Shipper also has a management cluster and application clusters. The management cluster is where we install and run Shipper. It contains all the cluster objects and secrets necessary to connect to the application clusters [55, 36]. We deploy the Shipper tool only on the management cluster and not on the application clusters. The application clusters are where we Shipper deploys resources. Unlike KubeFed, Shipper focuses on disaster recovery and rollback strategies rather than high availability and decreased latency. Shipper in the management cluster keeps track of all the rollout steps and facilitates the process of rolling back or aborting a deployment. It also allows the application deployers to override the rollout strategy used for each application cluster [36]. If a problem occurs in an application cluster, the management cluster can revert to the previous deployment. Some of the limitations of Shipper are that it cannot use rollout strategies for certain Kubernetes resources such as StatefulSets, HorizontalPodAutoscalers and ReplicaSets [36]. Moreover, Shipper has a strong integration with Helm and takes Helm as input, limiting the application deployer's use of regular K8s resources. Similar to KubeFed, Shipper has a single point of failure as well. If the management cluster fails, we won't be able to store the application configurations or roll out/revert applications running on application clusters.

4.7 Admiralty

Admiralty is a recent Cluster Federation project that is the most similar to KubeFed among all the listed projects in this chapter [56, 57]. Just like KubeFed, Admiralty uses a set of Kubernetes controllers to propagate resources to multiple member clusters. Admiralty calls these clusters either "source cluster" or "target cluster". Source clusters are similar to KubeFed's host clusters in that they both propagate resources to the member or target clusters; namely, we schedule resources in the source clusters, and they are deployed in the target clusters of Admiralty. All the Admiralty tools are installed in the source cluster. The main difference between Admiralty and KubeFed comes from how they propagate resources to the source or member clusters. When the source cluster schedules a pod to be deployed in a target cluster, Admiralty creates a dummy sleeping pod. Admiralty then uses a mutating pod admission webhook to block the pod creation to start a new round of scheduling and makes a pod placement decision for the federation and deploys the pods in the target clusters [58, 36]. In Admiralty, there are some intricate pod scheduling processes that result in the creation of temporary pods before deploying the final pods on the source clusters. It is a relatively new project with only five developers but has a potential to become the leading open-source Federation project.

4.8 Summary

In this chapter, we touched upon various Kubernetes Cluster Federation Projects, their advantages, and disadvantages. Some of the projects provide a "Network-Centric" approach, focusing on the networking and communication between containerized applications located in multiple clusters. They aim to establish a strong network connection between multiple clusters. Others provide a more "Kubernetes-Centric" approach focusing on changing the existing Kubernetes resources and configurations to create a centralized control plane that manages and orchestrates multiple clusters (e.g., KubeFed, Admiralty, Shipper). All these projects essentially aim to facilitate the management of a. multi-cluster Kubernetes setup where some of these clusters may be located on-premises and some located in the Cloud, etc. To create our solution in the next chapter, we chose to use KubeFed project since it is more popular and have a better community support. KubeFed has much more contributors than the other projects and it is actively being contributed [45].

Chapter 5

Design and Implementation

In the previous chapters, we talked about the main technologies that we will use in our solution. They included Kubernetes, Multi-Cluster architecture and Kubernetes Cluster Federation, focusing particularly on the KubeFed open-source project. We described how multi-cluster setup works and why it is used by most companies for application deployment. The same application can be deployed in several clusters located in different regions. This would improve high availability and provide low latency for the end-users. We also mentioned that all these clusters are managed separately and the application deployer would need to deploy the applications separately to each cluster. We talked about how Cluster Federation helps solve this problem by selecting one of the clusters as the “host-cluster” and making the rest of the clusters the “member-clusters”. The application deployer can simply create resources in the host-cluster and these same resources (e.g., deployments, services, etc.) will be automatically propagated to the member clusters. Thus, the member clusters will have the same functionalities that the host-cluster has. We mentioned in the introduction section that our purpose in this thesis is to provide a solution where the functionalities located in a central cloud cluster will also reside in several local branches outside the central cloud platform. The local branches may consist of machines with low-medium processing units located in regions different from the central cloud’s cluster to improve availability and latency. These local branches are independent from the cloud provider where the central host cluster resides (e.g., they can be machines such as a user’s computer), and all the resources that are created in the central cloud cluster should be automatically propagated in the local branches that may be located all around the world. An example scenario may include small local machines with low computing power for IoT devices, and the end-users can make requests to these local machines, which in turn will execute the computing processes at the edge network. Then, these machines may return the results to the central cloud. Moreover, the resources running in the clusters of the local machines can be easily updated and propagated by the host cluster running in the central cloud based on specific needs.

In this chapter, we will talk about how these concepts and several other techniques can be used to create a connectivity between a cluster running in a local machine and a host-cluster running on a central cloud, and then automatically propagate the resources residing in the central cloud cluster to the local on-premises cluster. Thus, the local machines all around the world can have the same functionalities and resources that the central cloud has. The first section will briefly go through the problem statement and overview the proposed POC solution. The second section will include the requirements that we need to fulfill to achieve the desired solution together with an in-depth overview of how our proposed POC fulfills these requirements. We will then close the chapter with a summary in the third section.

5.1 Proposed Solution

In the last chapter, we described that KubeFed tool can exactly provide the automatic propagation of resources from the host-cluster to the member clusters. The federation controller manager running in the host cluster can manage the deployment of resources on several member clusters. The application deployer connects with the host cluster and applies configuration information for applications to be deployed, and these applications/resources are automatically propagated to member clusters through the KubeFed controller manager. Thus, KubeFed Cluster Federation tool is a good starting point to automatically propagate the functionalities of a central host cluster residing in the Cloud to on-premises member clusters running on local machines. However, there are almost no examples out there creating a Cluster Federation setup between a Central Cloud cluster acting as the host cluster and on-premises clusters running on local machines acting as the member clusters. This is the setup we wish to create as our POC solution. The member clusters are not a part of any cloud provider, but rather they are local machines such as a user's computer. We wish to federate local clusters running in these local machines all around the world through a host cluster in a central cloud, and all these independent local clusters will have the same resources that the host cluster has. Moreover, the configurations of the propagated resources running on the local machines can be automatically reconfigured through the central host cluster based on specific needs of an application. End-users will be able to make their requests to the local machines possibly residing at the Edge Network in various locations instead of making their requests to the host-cluster residing in the central cloud. This will increase availability and decrease latency for the end-users while providing the same resources residing in the central host cluster to the local clusters.

Our POC aims to create "Cluster Federation" between a cluster running on a local machine and a cluster running on a central cloud by first establishing connectivity between these two environments. Once the connectivity is established and the central cluster running on the cloud has access to the local cluster, we will deploy KubeFed Controller Managers into the central cluster and federate the local cluster running on a simple MacBook Pro. All the resource configurations deployed on the central "host-cluster" will be automatically propagated to the local cluster outside the cloud platform; both environments will have the same resources and the same application accessible through two different URLs each.

Our local machine in this POC solution will be a MacBook Pro (2017) running in Europe (Dublin) region, and the cluster running in this local machine will be a Minikube cluster that is only accessible locally by default. The central “host-cluster” running on the cloud will be a GKE cluster located in Google Cloud Platform’s Australia region. After successfully establishing connectivity between the Minikube cluster running locally on a MacBook Pro (EU) and the central GKE cluster running on GCP (AU), we will deploy resources on the host GKE cluster and see the resources automatically propagated to the local Minikube cluster. To establish a connectivity between the local Minikube cluster and the central GKE cluster, we need to expose the Minikube cluster in some way since the Minikube API server cannot be accessed from outside our local machine by default. This will be further discussed in the requirements section.

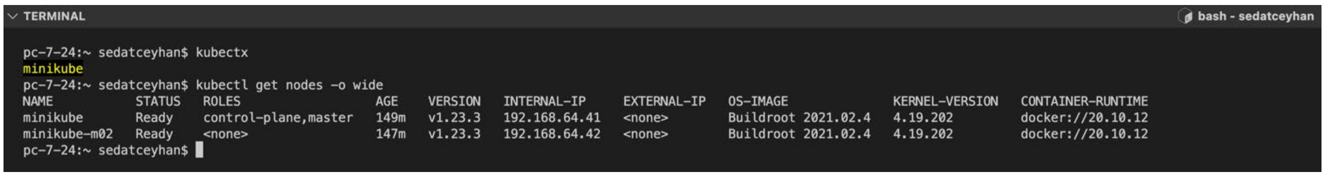
The next section will provide a detailed overview of a list of requirements that need to be fulfilled together with how they are fulfilled to create a working POC solution.

5.2 Requirements and Fulfillments

In this section, we will list the requirements that need to be fulfilled in the order they are fulfilled. For each requirement, we will provide a detailed solution with example commands and screenshots.

5.2.1 Creating a Local Minikube Cluster

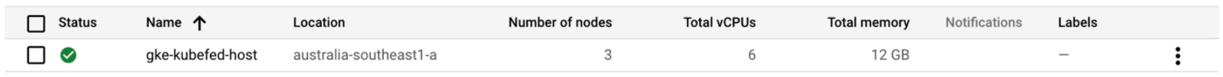
As mentioned previously, we are going to use our MacBook Pro (2017) as our local machine, and we need to create a local cluster in which to automatically deploy the resources of the central cluster running in the cloud. For this, we will use a local Minikube cluster. The Minikube cluster will represent our local branch and the central GKE cluster running in the Google Cloud Platform will propagate resources to this local Minikube cluster. We created the Minikube cluster with one master node and one worker node through our local terminal. We will deploy the workloads on the worker node (minikube-m02). Below is a screenshot showing our local setup:



```
pc-7-24:~ sedatceyhan$ kubectx
minikube
pc-7-24:~ sedatceyhan$ kubectl get nodes -o wide
NAME           STATUS   ROLES                  AGE    VERSION   INTERNAL-IP     EXTERNAL-IP   OS-IMAGE              KERNEL-VERSION   CONTAINER-RUNTIME
minikube       Ready    control-plane,master   149m   v1.23.3   192.168.64.41   <none>        Buildroot 2021.02.4   4.19.202         docker://20.10.12
minikube-m02   Ready    <none>                 147m   v1.23.3   192.168.64.42   <none>        Buildroot 2021.02.4   4.19.202         docker://20.10.12
pc-7-24:~ sedatceyhan$
```

Figure 5.1: Setting up Minikube on the Local Machine

As the next requirement, we need to create certificates for our Minikube cluster and introduce these certificates to the Cloud Platform so that it has access to our local clusters. Moreover, since our local machine resides in Dublin, Europe, our Minikube cluster and all its nodes will be from Dublin, Europe as well. However, the central cluster running in GKE will be from Australia region; it will federate and configure the resources of our local Minikube cluster located in Europe.



Status	Name ↑	Location	Number of nodes	Total vCPUs	Total memory	Notifications	Labels
✓	gke-kubefed-host	australia-southeast1-a	3	6	12 GB		–

Figure 5.2: Central GKE cluster located in Australia Region

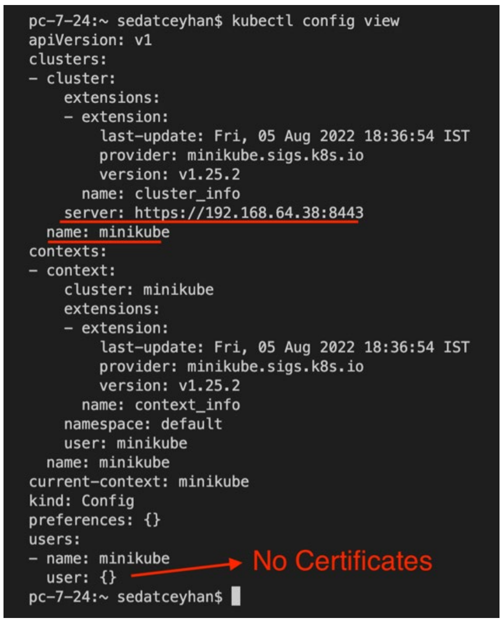
To automatically propagate its resources to the local Minikube cluster, the central GKE cluster needs to connect to our local environment first, which brings us to the next section.

5.2.2 Connectivity between Minikube and GKE

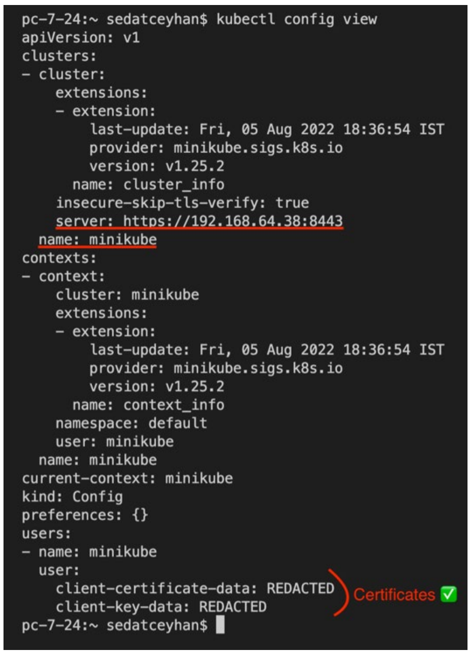
To provide the central GKE cluster's resources to the local Minikube cluster through KubeFed, we need to first establish a connectivity between the GKE cluster and the Minikube cluster. The solution aims to combine the **KubeConfig** files of both platforms to create a unique **KubeConfig_merged** file in Cloud Shell so that the Minikube cluster is visible to the Cloud platform. A Kubeconfig is a YAML file that contains the Kubernetes cluster's credentials, certificate, and secret token. If you are utilizing a managed Kubernetes cluster, the cluster administrator or a cloud platform may provide you with this configuration file. When you use kubectl, it connects to the Kubernetes cluster API using the information in the kubeconfig file. The default Kubeconfig file location has the path of "`$HOME/.kube/config`" [59]. A KubeConfig file contains the following information to connect to the Kubernetes clusters:

1. **certificate-authority-data**: Cluster CA
2. **server**: Cluster endpoint (IP/DNS of master node)
3. **name**: Cluster name
4. **user**: name of the user/service account.
5. **token**: Secret token of the user/service account.

Combining the KubeConfig files of both platforms to create the kubeconfig_merged file does not expose the Minikube for external connections, however it is a necessary step before exposing our local Minikube cluster. As an initial requirement, we need to create client-certificate-data and client-key-data for our local Minikube cluster. Below are the KubeConfig files before and after the creation of these certificates. We get these outputs by running "`kubectl config view`" in our local terminal for our Minikube cluster. As you can see, on the left-hand side, we have the 'minikube' cluster context in our local config file, however, we have no certificates. The "user" key of the config file has no value in it. We get the screenshot on the right-hand side after creating the certificates for the Minikube cluster. Since the certificates are long strings, they are redacted in the output. The KubeConfig file that we get on the right-hand side is the one that the Google Cloud Platform needs in order to connect to our local Minikube cluster:



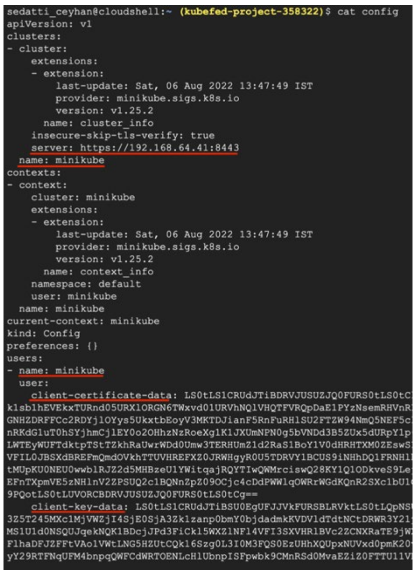
```
pc-7-24:~ sedatceyhan$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    extensions:
    - extension:
        last-update: Fri, 05 Aug 2022 18:36:54 IST
        provider: minikube.sigs.k8s.io
        version: v1.25.2
      name: cluster_info
    server: https://192.168.64.38:8443
  name: minikube
contexts:
- context:
    cluster: minikube
    extensions:
    - extension:
        last-update: Fri, 05 Aug 2022 18:36:54 IST
        provider: minikube.sigs.k8s.io
        version: v1.25.2
      name: context_info
    namespace: default
    user: minikube
  name: minikube
current-context: minikube
kind: Config
preferences: {}
users:
- name: minikube
  user: {}
pc-7-24:~ sedatceyhan$
```



```
pc-7-24:~ sedatceyhan$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    extensions:
    - extension:
        last-update: Fri, 05 Aug 2022 18:36:54 IST
        provider: minikube.sigs.k8s.io
        version: v1.25.2
      name: cluster_info
    insecure-skip-tls-verify: true
    server: https://192.168.64.38:8443
  name: minikube
contexts:
- context:
    cluster: minikube
    extensions:
    - extension:
        last-update: Fri, 05 Aug 2022 18:36:54 IST
        provider: minikube.sigs.k8s.io
        version: v1.25.2
      name: context_info
    namespace: default
    user: minikube
  name: minikube
current-context: minikube
kind: Config
preferences: {}
users:
- name: minikube
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
pc-7-24:~ sedatceyhan$
```

Figure 5.3: KubeConfig files on the Local Machine with and without Certificates

Now we have the necessary certificates of our Minikube so that the central GKE cluster running on Google Cloud Platform can access it. We now need to upload the local kubeconfig file on the right-hand side of the figure 5.1 to the Google Cloud Shell. When we list the files on the Cloud Shell, we need to see the local "config" file in it:

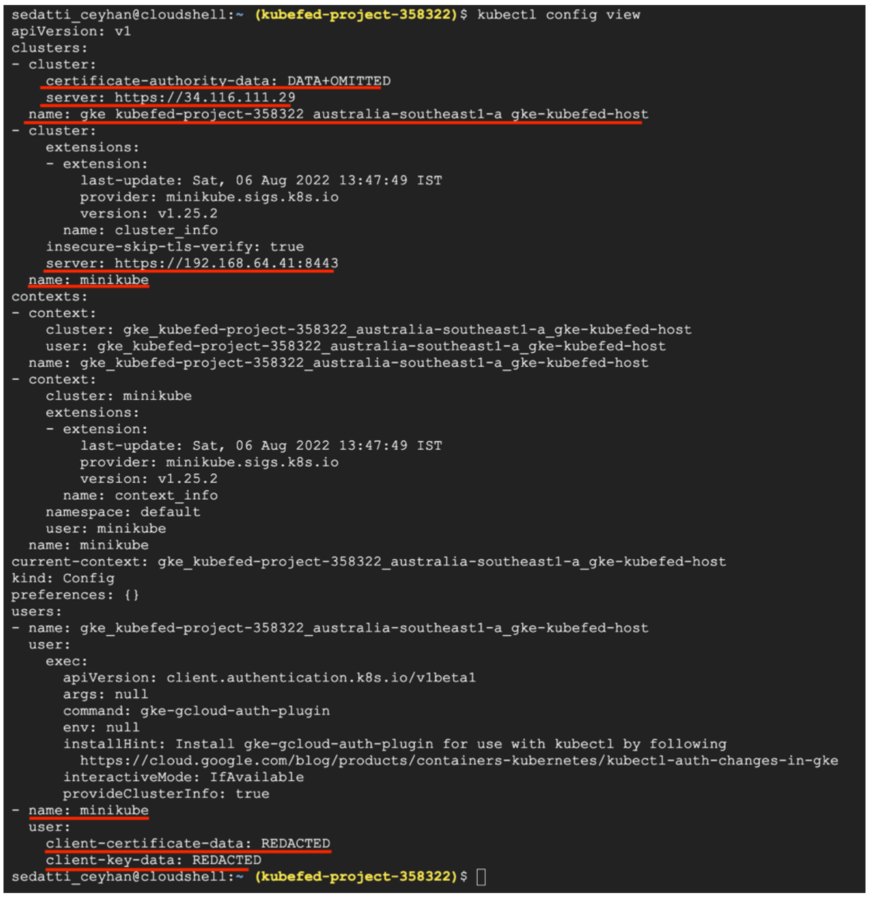


```
sedatti_ceyhan@cloudshell:~ (kubefed-project-358322)$ cat config
apiVersion: v1
clusters:
- cluster:
    extensions:
    - extension:
        last-update: Sat, 06 Aug 2022 13:47:49 IST
        provider: minikube.sigs.k8s.io
        version: v1.25.2
      name: cluster_info
    insecure-skip-tls-verify: true
    server: https://192.168.64.41:8443
  name: minikube
contexts:
- context:
    cluster: minikube
    extensions:
    - extension:
        last-update: Sat, 06 Aug 2022 13:47:49 IST
        provider: minikube.sigs.k8s.io
        version: v1.25.2
      name: context_info
    namespace: default
    user: minikube
  name: minikube
current-context: minikube
kind: Config
preferences: {}
users:
- name: minikube
  user:
    client-certificate-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tC
k1sblhEVEkxTURnd05URXlORGN6TWxvd01URVhNQlVHQTFVRQpDaE1PYzNsemRHVnR
GNHZDRFFCc2RDYjlOYys5UkxtbEoyV3MKTDJianF5RnFuRHlSU2FTZW94NmQ5NEF5c
nRKdGluT0hSYjhmCjlEY0o2OHhzNzRoeXglK1JXUmNPN0g5bVNDd3B5ZUx5dURpY1p
LWTEyWUFTdktpTStTZkhRaUwrWDd0Umw3TERHUmZ1d2RaS1BoY1V0dHRHTXM0ZEswS
VFIL0JBSXdBREFmQmdOVkhTTUVHREFXZ0JRWHgyR0U5TDRVY1BCUS9iNHhDQ1FRNHl
tMUpKU0NEU0wwblRJZ2d5MHBzeU1YWitqajRQYTIwQWMrciswQ28KY1Q1ODkveS9Le
EFnTXpmVE5zNHlnV2ZPSUQ2c1BQNnZpZ09OCjc4cDdPWWlqOWRrWGdKQnR2SXc1bU1
9PQotLS0tLUVORCBDRVJUSUZJQ0FURS0tLS0tCg==
    client-key-data: LS0tLS1CRUdJTiBSU0EgUFJJVkFURSBLRVktLS0tLQpNS
3Z5T245MXc1MjVWZjI4SjE0SjA3Zk1zanp0bmY0bjdadmkKVDVldTdtNCtDRWR3Y21
MS1U1d0NSQUJqekNQK1BDcjJPd3FiCkl5WXZlNFl4VFI3SXVHRlBVc2ZCNXRaTE9jW
FlhaDFJZFFtVAolVWtLNG5HZUtCQk16Szg0L3I0M3FQS0EzUHhXQUpxNUVxd0pmK20
yY29RTFNqUFM4bnpqQWFCdWRTOENLcHlUbnpISFpwbk9CMnRSd0MvaEZiZ0FTTU11V
```

Figure 5.4: Local Environment's "kubeconfig" file uploaded to Cloud Shell

We have successfully uploaded our local environment's kubeconfig file to the Cloud Shell, and we can clearly see the certificates information of our Minikube cluster. This information is necessary for combining local kubeconfig file and GCP's kubeconfig file to create a kubeconfig_merged file in the cloud platform, and this is the next step that we will perform. Running the command below will create a "**kubeconfig_merged**" file that will contain all the information of the local Minikube cluster and our central GKE cluster by combining their kubeconfig files:

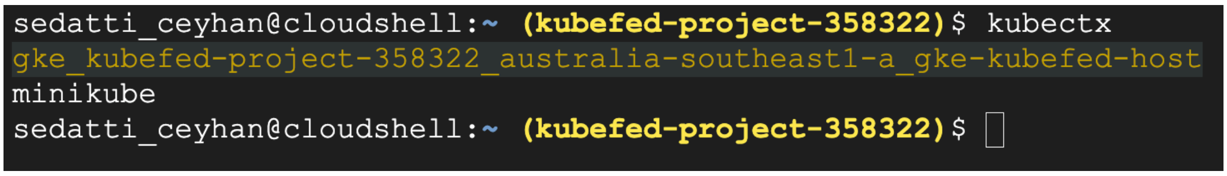
```
KUBECONFIG=<path to GKE kubeconfig file>:<path to the uploaded Minikube
kubeconfig file> kubectl config view --flatten > ~/kubeconfig_merged
```



```
sedatti_ceyhan@cloudshell:~ (kubefed-project-358322)$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://34.116.111.29
  name: gke_kubefed-project-358322_australia-southeast1-a_gke-kubefed-host
- cluster:
    extensions:
    - extension:
        last-update: Sat, 06 Aug 2022 13:47:49 IST
        provider: minikube.sigs.k8s.io
        version: v1.25.2
      name: cluster_info
    insecure-skip-tls-verify: true
    server: https://192.168.64.41:8443
  name: minikube
contexts:
- context:
    cluster: gke_kubefed-project-358322_australia-southeast1-a_gke-kubefed-host
    user: gke_kubefed-project-358322_australia-southeast1-a_gke-kubefed-host
  name: gke_kubefed-project-358322_australia-southeast1-a_gke-kubefed-host
- context:
    cluster: minikube
    extensions:
    - extension:
        last-update: Sat, 06 Aug 2022 13:47:49 IST
        provider: minikube.sigs.k8s.io
        version: v1.25.2
      name: context_info
    namespace: default
    user: minikube
  name: minikube
current-context: gke_kubefed-project-358322_australia-southeast1-a_gke-kubefed-host
kind: Config
preferences: {}
users:
- name: gke_kubefed-project-358322_australia-southeast1-a_gke-kubefed-host
  user:
    exec:
      apiVersion: client.authentication.k8s.io/v1beta1
      args: null
      command: gke-gcloud-auth-plugin
      env: null
      installHint: Install gke-gcloud-auth-plugin for use with kubectl by following
        https://cloud.google.com/blog/products/containers-kubernetes/kubectl-auth-changes-in-gke
      interactiveMode: IfAvailable
      provideClusterInfo: true
- name: minikube
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
sedatti_ceyhan@cloudshell:~ (kubefed-project-358322)$
```

Figure 5.5: Merged kubeconfig files in GCP

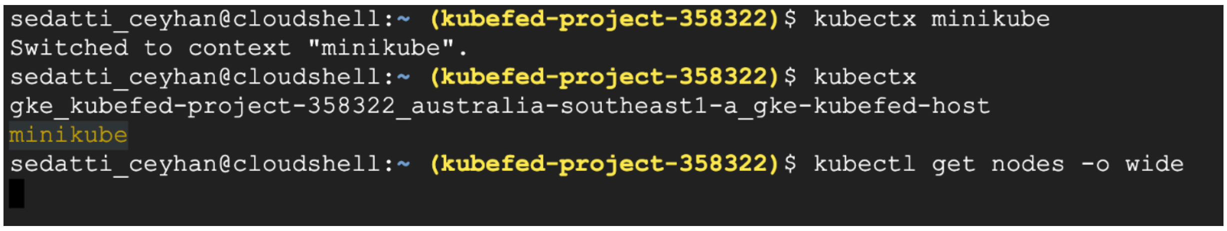
Above is the newly merged kubeconfig file created by combining our local Minikube cluster's kubeconfig and the central cloud cluster's kubeconfig files. The GKE is now using this kubeconfig file as its cluster data. Notice that we have both the Minikube and the central GKE clusters (gke_kubefed-project-358322_australia-southeast1-a_gke-kubefed-host) in the configuration file. We can also see that GKE has access to our local Minikube cluster's client-certificate-data together with its client-key-data. We also see the certificate-authority-data of our central GKE cluster under the same configuration file. Consequently, our cloud platform has access to the local Minikube cluster. When we run `kubectx` to get the cluster contexts in GCP, we should see our local Minikube cluster along with the central GKE cluster:



```
sedatti_ceyhan@cloudshell:~ (kubefed-project-358322)$ kubectx
gke_kubefed-project-358322_australia-southeast1-a_gke-kubefed-host
minikube
sedatti_ceyhan@cloudshell:~ (kubefed-project-358322)$
```

Figure 5.6: GCP having access to our local Minikube as a cluster context

Observe in Figure 5.6 that our central cloud can now see Minikube among its list of cluster contexts. This was a necessary step to establish a connectivity between our local Minikube cluster and the central GKE cluster. However, we still cannot access the API server of our local Minikube cluster to create/query its resources. Below is a screenshot showing that we are still unable to query the node information of our local Minikube from the central GKE cluster. Refer to figure 5.1 to see the running nodes of our Minikube cluster accessed from our local terminal:



```
sedatti_ceyhan@cloudshell:~ (kubefed-project-358322)$ kubectx minikube
Switched to context "minikube".
sedatti_ceyhan@cloudshell:~ (kubefed-project-358322)$ kubectx
gke_kubefed-project-358322_australia-southeast1-a_gke-kubefed-host
minikube
sedatti_ceyhan@cloudshell:~ (kubefed-project-358322)$ kubectl get nodes -o wide
```

Figure 5.7: Unable to access the resources of local Minikube from GCP

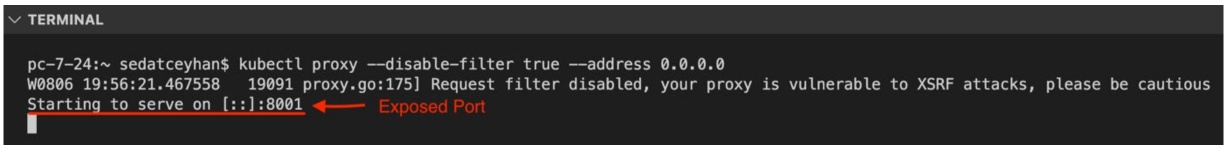
This brings us to the next requirement.

5.2.3 Exposing Minikube API server

Remember that kube-apiserver is one of the Control Plane components of Kubernetes. The API server of the control plane is responsible for exposing the Kubernetes API as the front-end for application developers. When a user wishes to deploy new resources in a Kubernetes cluster, they communicate with the API server. By default, our local Minikube's API server is not publicly exposed. Thus, we cannot propagate resources to our Minikube cluster from the central cloud as can be seen in Figure 5.7. There are three main ways to expose our local API server externally: acquiring a Public IP address, VPN or tunnelling.

To make things simple, we will use the tunnelling approach to expose our local Minikube API server. We will use a software tunnelling tool called **ngrok**. ngrok is a globally distributed reverse-proxy that helps developers expose their applications running on any cloud, private network or local machine; we are interested in exposing the app running on our local machine in this case, and ngrok is one of the fastest ways to expose our Minikube API server running on localhost [60]. We will install a ngrok agent with a specific auth token on our local machine. The ngrok agent will use this tunnel auth token to connect to the ngrok cloud and establish a connection to the ngrok service. Once we have an established connection, we will receive a public endpoint to which we can make external requests (e.g., requests to Minikube API server). This ngrok public endpoint will forward our external requests to our local service (e.g., local API server). When we make a request to this ngrok endpoint, the ngrok edge encrypts the request and forwards it to our locally running ngrok agent, and this ngrok agent is responsible for sending this request to our local upstream service. The communication between the ngrok edge and our local ngrok agent is secure and TLS encrypted. Similarly, the traffic from user to the ngrok edge and traffic from ngrok agent to our local service use the same encryption protocols [60].

Before setting up ngrok tunnelling, we need our kubectl proxy to allow external traffic to our local cluster. We will allow all incoming requests to the Minikube API server:



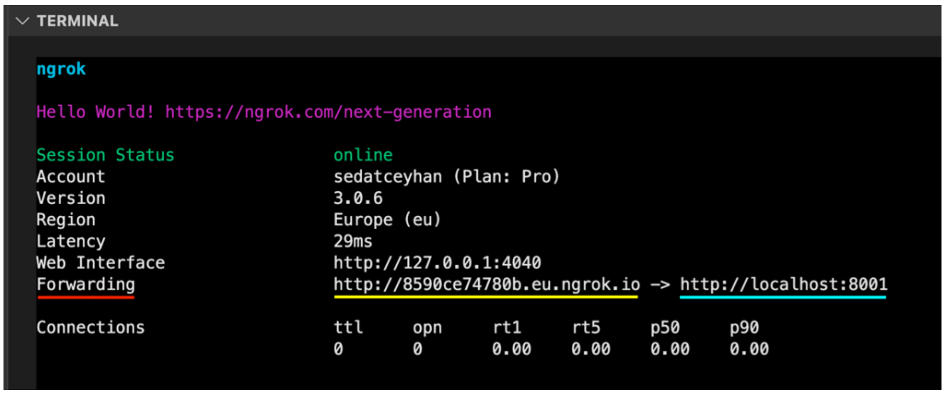
```
pc-7-24:~ sedatceyhan$ kubectl proxy --disable-filter true --address 0.0.0.0
W0806 19:56:21.467558   19091 proxy.go:175] Request filter disabled, your proxy is vulnerable to XSRF attacks, please be cautious
Starting to serve on [::]:8001
```

Figure 5.8: Exposing port=8001 for Minikube API server

Now our API server is listening to the requests made locally on port 8001. The command in Figure 5.8 opened specifically the port 8001 since it is kube-proxy's default port. The Minikube API server is now exposed locally via `http://localhost:8001`. As the next step, we need ngrok to act as a second proxy forwarding requests received from its generated public endpoint to `http://localhost:8001`. When the central GKE cluster makes requests to the public ngrok endpoint, the requests will be automatically forwarded to `http://localhost:8001`.

We run the following command to create a ngrok public endpoint and forward all requests made to this public endpoint to our local API server of the Minikube:

```
ngrok http --scheme=http 8001
```



```
ngrok

Hello World! https://ngrok.com/next-generation

Session Status                online
Account                       sedatceyhan (Plan: Pro)
Version                       3.0.6
Region                        Europe (eu)
Latency                       29ms
Web Interface                 http://127.0.0.1:4040
Forwarding                    http://8590ce74780b.eu.ngrok.io -> http://localhost:8001

Connections                   ttl     opn     rt1     rt5     p50     p90
                              0       0       0.00    0.00    0.00    0.00
```

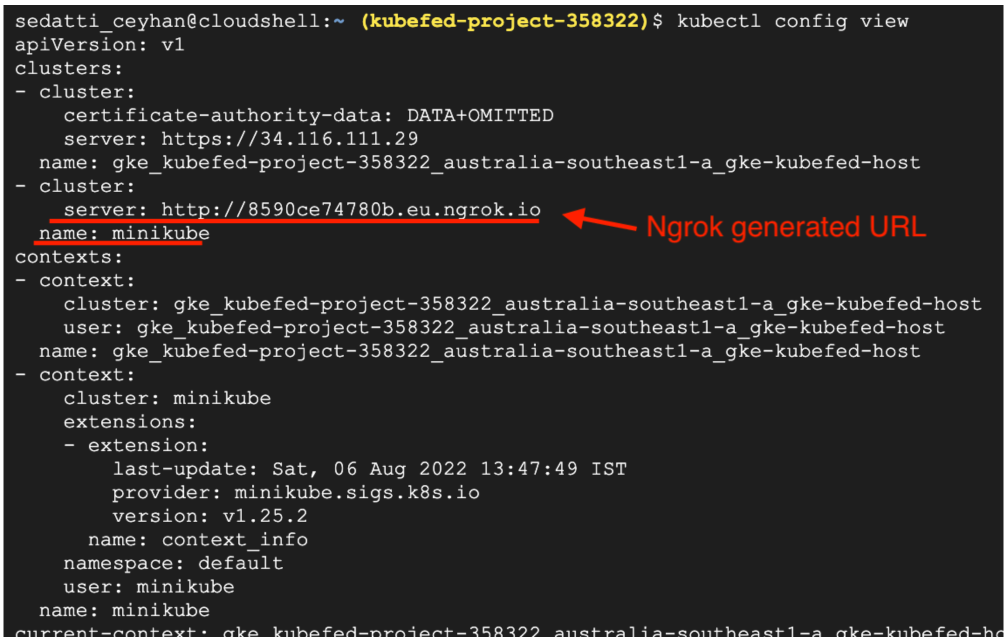
Figure 5.9: ngrok URL forwarding requests to the Local API server of Minikube

The URL that's underlined in yellow in Figure 5.9 is the ngrok generated public endpoint that we will use to access our local Minikube's API server. The blue-underlined address is the local API server to which all the requests made to the ngrok generated URL will be forwarded.

We have now set up everything needed to publicly expose our Minikube API server so that the cloud platform can make queries to create resources in our local branch. We have to make a few changes in the merged kubeconfig file of GCP. Notice in Figure 5.5 that the API server of our Minikube is just a local address, which cannot be accessed externally by the cloud platform. For GKE to make successful requests to Minikube's API server, we need to change this local address with the ngrok generated URL (underlined in yellow) in Figure 5.9:

```
kubectl config set-cluster minikube --server='<ngrok generated url>'
```

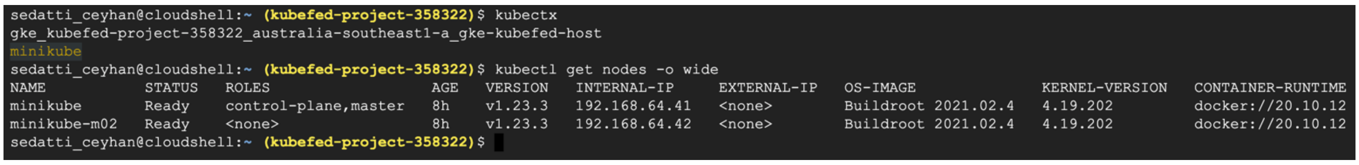
Changing the address of Minikube on Cloud Shell will automatically change the kubeconfig_merged file on the cloud:



```
sedatti_ceyhan@cloudshell:~ (kubefed-project-358322)$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://34.116.111.29
  name: gke_kubefed-project-358322_australia-southeast1-a_gke-kubefed-host
- cluster:
    server: http://8590ce74780b.eu.ngrok.io
  name: minikube
contexts:
- context:
    cluster: gke_kubefed-project-358322_australia-southeast1-a_gke-kubefed-host
    user: gke_kubefed-project-358322_australia-southeast1-a_gke-kubefed-host
  name: gke_kubefed-project-358322_australia-southeast1-a_gke-kubefed-host
- context:
    cluster: minikube
    extensions:
    - extension:
        last-update: Sat, 06 Aug 2022 13:47:49 IST
        provider: minikube.sigs.k8s.io
        version: v1.25.2
      name: context_info
    namespace: default
    user: minikube
  name: minikube
current-context: gke_kubefed-project-358322_australia-southeast1-a_gke-kubefed-ho
```

Figure 5.10: Minikube API server changed to ngrok URL in kubeconfig_merged

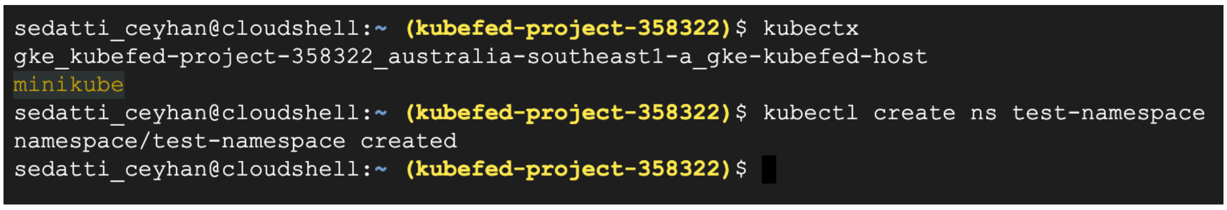
We can now make queries to the local API server of our Minikube from the GCP. The central cloud's kubeconfig_merged file will consider the unique ngrok generated URL to be the API server to which it will make queries for creating resources in Minikube. All the requests we make to the ngrok generated URL from the central cloud will be forwarded to our local Minikube API server (e.g., http://localhost:8001) by the ngrok agent. Below is a successful query to our local Minikube cluster from GCP contrary to Figure 5.7:



```
sedatti_ceyhan@cloudshell:~ (kubefed-project-358322)$ kubectx
gke_kubefed-project-358322_australia-southeast1-a_gke-kubefed-host
minikube
sedatti_ceyhan@cloudshell:~ (kubefed-project-358322)$ kubectl get nodes -o wide
NAME           STATUS   ROLES                  AGE   VERSION   INTERNAL-IP     EXTERNAL-IP   OS-IMAGE              KERNEL-VERSION   CONTAINER-RUNTIME
minikube       Ready    control-plane,master   8h    v1.23.3   192.168.64.41   <none>        Buildroot 2021.02.4   4.19.202         docker://20.10.12
minikube-m02   Ready    <none>                 8h    v1.23.3   192.168.64.42   <none>        Buildroot 2021.02.4   4.19.202         docker://20.10.12
sedatti_ceyhan@cloudshell:~ (kubefed-project-358322)$
```

Figure 5.11: Successfully accessing to the resources of local Minikube from GCP

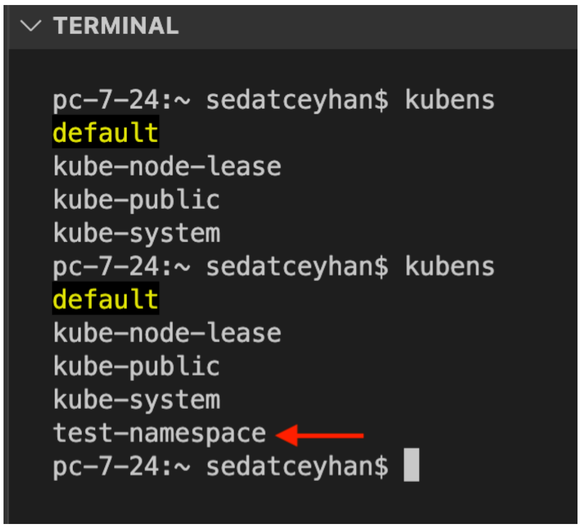
As another example, we will create a namespace called "test-namespace" in our local Minikube cluster through GKE; namely we will use the connectivity that we've just established to the Minikube cluster from the GCP and make a "create a namespace" request to Minikube's API server from our central cloud. As a result, we expect the namespace resource to be created in the Minikube when we check its namespaces locally:



```
sedatti_ceyhan@cloudshell:~ (kubefed-project-358322)$ kubectx
gke_kubefed-project-358322_australia-southeast1-a_gke-kubefed-host
minikube
sedatti_ceyhan@cloudshell:~ (kubefed-project-358322)$ kubectl create ns test-namespace
namespace/test-namespace created
sedatti_ceyhan@cloudshell:~ (kubefed-project-358322)$
```

Figure 5.12: Creation of a test-namespace in Minikube cluster from GCP

We can see from Figure 5.12 that we made a successful request to our Minikube's API server from GCP to create a test-namespace. Below is a screenshot from our local terminal showing that the test-namespace is indeed created in our local Minikube cluster:



```
pc-7-24:~ sedatceyhan$ kubens
default
kube-node-lease
kube-public
kube-system
pc-7-24:~ sedatceyhan$ kubens
default
kube-node-lease
kube-public
kube-system
test-namespace
pc-7-24:~ sedatceyhan$
```

Figure 5.13: test-namespace successfully created in the local Minikube Cluster

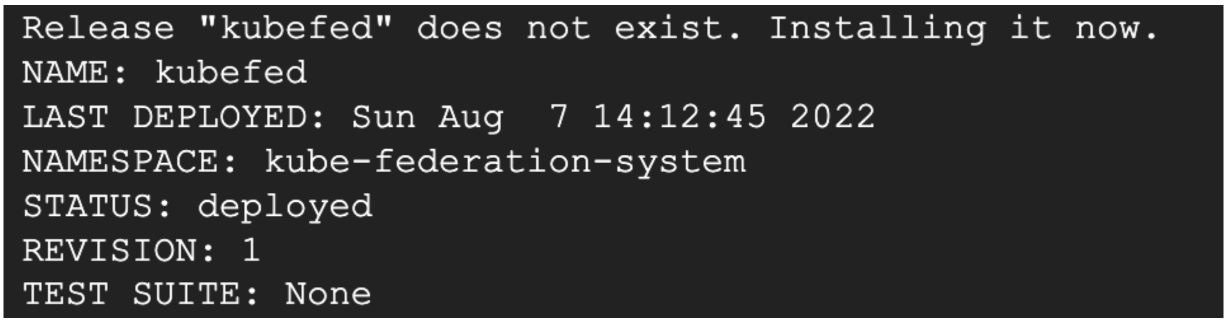
We successfully established a connectivity between our local Minikube cluster and the central GKE cluster. We can now query, create and update resources in our local Minikube cluster through GCP. We're using ngrok generated URL to make our requests; and those requests are forwarded to the local API server. We can remove the "test-namespace" from our local Minikube cluster now that we showed the existence of connectivity.

Now that we have fulfilled the connectivity requirement, the next step is to install all the KubeFed tools on the central GKE cluster. After we install all the KubeFed tools, the GKE cluster will be the host-cluster for the cluster federation. Then, we will make our local Minikube a member cluster. We will deploy resources on the GKE host-cluster and see them automatically propagated to the local Minikube cluster. This will provide the same functionalities that our central cloud has to our local branch.

5.2.4 Installing KubeFed on Central GKE Cluster

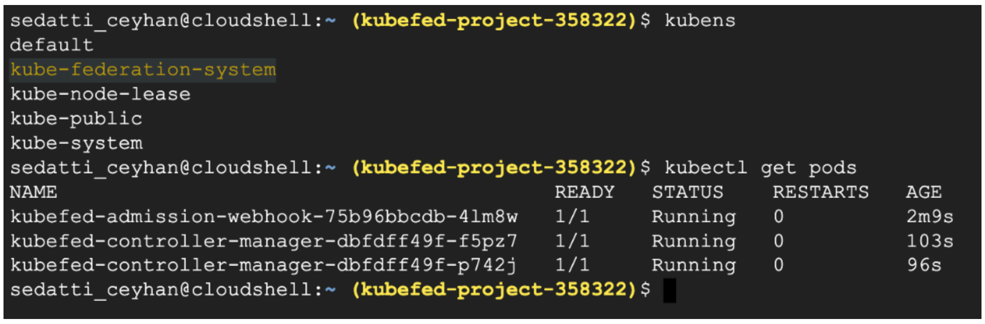
As a next step, we will be deploying the KubeFed Controller Managers on our central GKE cluster residing in Australia. Remember from the previous chapter that KubeFed Controller Manager is very similar to Kubernetes Controller Manager, with the exception that Kubernetes Controller Manager constantly monitors the state of pods, deployments, and nodes within a single cluster, whereas KubeFed Controller Manager monitors the state of all federated clusters to maintain the desired state of the member clusters. The KubeFed Controller Manager utilizes a push reconciler technique in order to synchronize the status of all Federated resources with those deployed in the member clusters. In our case, KubeFed Controller Manager will be responsible for checking whether all the desired resources are automatically propagated to our local Minikube cluster.

Below, we are creating a namespace called "**kube-federation-system**" inside our central GKE cluster whose context name is "**gke-kubefed-host**" (we changed the context name to a simpler name with no underscore character as Kubefedctl doesn't allow underscore in context names). All the KubeFed tools (e.g., KubeFed controller manager and webhook pods) will be deployed in this namespace. The commands used for creating the setup will be provided in the Appendix section at the end of our thesis. The output we get is as shown:



```
Release "kubefed" does not exist. Installing it now.
NAME: kubefed
LAST DEPLOYED: Sun Aug  7 14:12:45 2022
NAMESPACE: kube-federation-system
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

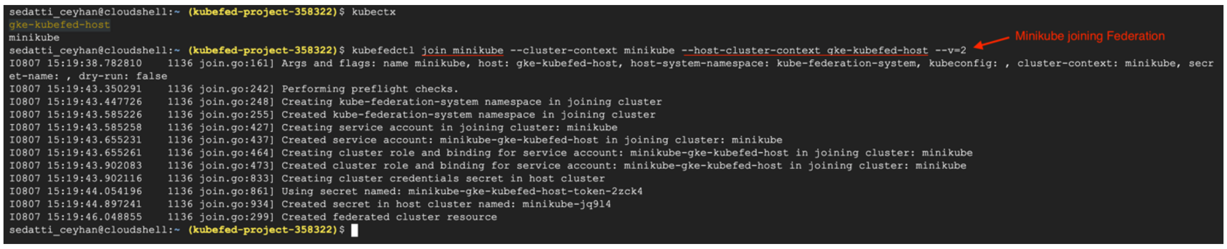
Figure 5.14: Installing KubeFed in GCP



```
sedatti_ceyhan@cloudshell:~ (kubefed-project-358322)$ kubens
default
kube-federation-system
kube-node-lease
kube-public
kube-system
sedatti_ceyhan@cloudshell:~ (kubefed-project-358322)$ kubectl get pods
NAME                                         READY   STATUS    RESTARTS   AGE
kubefed-admission-webhook-75b96bbcdb-4lm8w   1/1     Running   0          2m9s
kubefed-controller-manager-dbfdff49f-f5pz7   1/1     Running   0          103s
kubefed-controller-manager-dbfdff49f-p742j   1/1     Running   0          96s
sedatti_ceyhan@cloudshell:~ (kubefed-project-358322)$ 
```

Figure 5.15: kubefed-controller-manager pods deployed to kube-federation-system Namespace

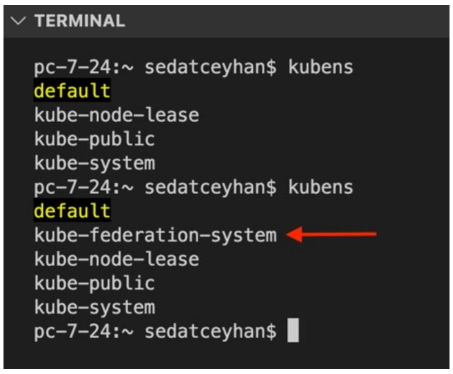
We can see that a namespace called "kube-federation-system" is automatically created in our central GKE cluster; all the federation controller manager pods are deployed in this namespace, effectively turning our central GKE cluster residing in Australia region into a KubeFed host-cluster. Since we have all the components running in our central GKE cluster necessary for the federation of the local Minikube cluster in Europe, we can now add our local Minikube into the federation.



```
sedatti_ceyhan@cloudshell:~ (kubefed-project-358322)$ kubectx
gke-kubefed-host
minikube
sedatti_ceyhan@cloudshell:~ (kubefed-project-358322)$ kubefedctl join minikube --cluster-context minikube --host-cluster-context gke-kubefed-host --v=2
I0807 15:19:38.782810    1136 join.go:161] Args and flags: name minikube, host: gke-kubefed-host, host-system-namespace: kube-federation-system, kubeconfig: , cluster-context: minikube, secr
et-name: , dry-run: false
I0807 15:19:43.350291    1136 join.go:242] Performing preflight checks.
I0807 15:19:43.447726    1136 join.go:248] Creating kube-federation-system namespace in joining cluster
I0807 15:19:43.585226    1136 join.go:255] Created kube-federation-system namespace in joining cluster
I0807 15:19:43.585258    1136 join.go:427] Creating service account in joining cluster: minikube
I0807 15:19:43.655231    1136 join.go:437] Created service account: minikube-gke-kubefed-host in joining cluster: minikube
I0807 15:19:43.655261    1136 join.go:464] Creating cluster role and binding for service account: minikube-gke-kubefed-host in joining cluster: minikube
I0807 15:19:43.902083    1136 join.go:473] Created cluster role and binding for service account: minikube-gke-kubefed-host in joining cluster: minikube
I0807 15:19:43.902116    1136 join.go:833] Creating cluster credentials secret in host cluster
I0807 15:19:44.054196    1136 join.go:861] Using secret named: minikube-gke-kubefed-host-token-2zck4
I0807 15:19:44.897241    1136 join.go:934] Created secret in host cluster named: minikube-jq9l4
I0807 15:19:46.048855    1136 join.go:299] Created federated cluster resource
sedatti_ceyhan@cloudshell:~ (kubefed-project-358322)$ 
```

Figure 5.16: Minikube joining the Federation and becoming a Member-Cluster

From Figure 5.16, we see that Minikube has successfully joined the Federation. This will automatically create a "kube-federation-system" namespace in our local Minikube cluster as we can see on Figure 5.17 below. The creation of this namespace is just to confirm that Minikube is now a member cluster of our cluster federation. No resources will be deployed in this namespace.

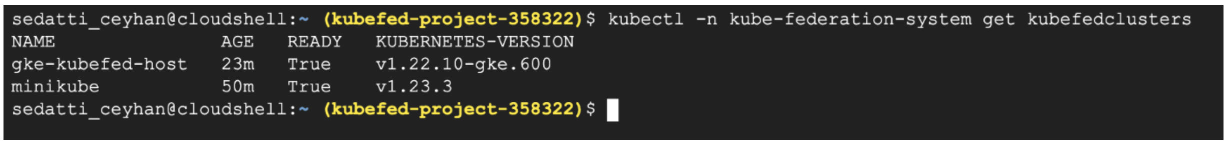


```
pc-7-24:~ sedatceyhan$ kubens
default
kube-node-lease
kube-public
kube-system
pc-7-24:~ sedatceyhan$ kubens
default
kube-federation-system
kube-node-lease
kube-public
kube-system
pc-7-24:~ sedatceyhan$ 
```

Figure 5.17: Confirmation of Minikube's Federation

Since we want to deploy the same resources on both the local Minikube cluster and the central GKE host-cluster, we will make the GKE host-cluster a member of its own; namely, the host-cluster will federate itself, and the federated resources created in the GKE host-cluster will propagate to not only the local Minikube cluster, but also to the host-cluster itself. This way, we will have the same resources and applications running in both the local Minikube cluster and the GKE host-cluster.

We repeat the same process above for the GKE host-cluster and run the following command `kubectl get kubefedclusters -n kube-federation-system` in Google Cloud Shell to see all the federated clusters in our setup. As expected, we have two federated clusters: local Minikube and GKE host-cluster:



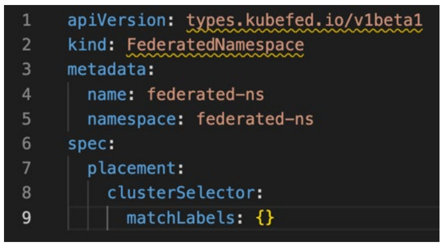
```
sedatti_ceyhan@cloudshell:~ (kubefed-project-358322)$ kubectl -n kube-federation-system get kubefedclusters
NAME               AGE   READY   KUBERNETES-VERSION
gke-kubefed-host   23m   True    v1.22.10-gke.600
minikube           50m   True    v1.23.3
sedatti_ceyhan@cloudshell:~ (kubefed-project-358322)$
```

Figure 5.18: List of Federated Clusters

We have successfully created the Federation environment and we can now create resources in the host-cluster and see them propagated to both our local Minikube cluster in Europe and the central GKE host-cluster itself.

5.2.5 Deploying Resources on GKE Host-Cluster

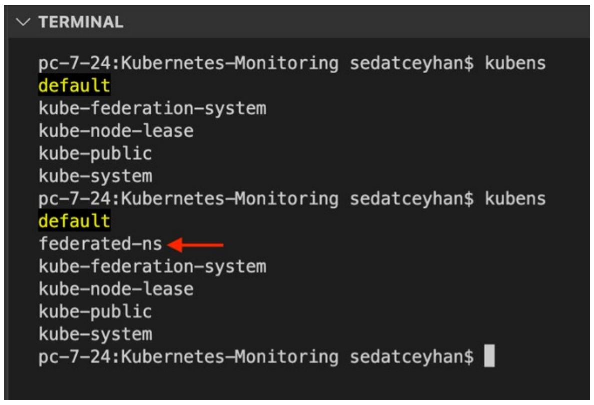
Now, we have a KubeFed host-cluster (central GKE cluster) and two member clusters: local Minikube and the central GKE cluster itself. We want the resources and applications created in the central GKE cluster located in Australia to be propagated to both member clusters. To achieve this, we need to create a "FederatedNamespace" object in our GKE host-cluster. Remember from the previous chapter on KubeFed that, all the federated resources we create in the host-cluster (e.g., FederatedDeployment, FederatedService, FederatedNamespace, etc.) are propagated as regular resources in the member clusters (e.g., Deployment, Service, Namespace, etc). Thus, the FederatedNamespace object that we will create in the GKE host-cluster will be propagated to our local Minikube cluster as a regular namespace object. Below is the FederatedNamespace object called "**federated-ns**" we are creating in the GKE host-cluster:



```yaml
apiVersion: types.kubefed.io/v1beta1
kind: FederatedNamespace
metadata:
  name: federated-ns
  namespace: federated-ns
spec:
  placement:
    clusterSelector:
      matchLabels: {}
```

Figure 5.19: FederatedNamespace object created in the GKE host-cluster

Deploying the FederatedNamespace object in Figure 5.19 inside the GKE host-cluster will automatically create a regular namespace object called "**federated-ns**" in all the member clusters. In particular, we will see the creation of a "federated-ns" namespace in our local Minikube cluster. All the resources created in this FederatedNamespace by the central GKE host-cluster will be propagated to the "federated-ns" namespace of all our member clusters (e.g., local Minikube cluster). Below is a screenshot showing the creation of a namespace called "federated-ns" inside our local Minikube. This namespace is automatically created in our local Minikube cluster since we ran the configuration (Figure 5.19) in the host GKE cluster:

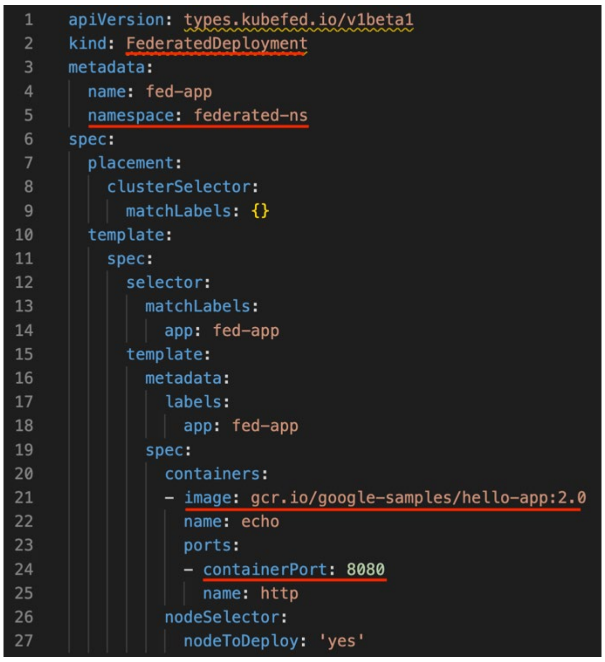


```
pc-7-24:Kubernetes-Monitoring sedatceyhan$ kubens
default
kube-federation-system
kube-node-lease
kube-public
kube-system
pc-7-24:Kubernetes-Monitoring sedatceyhan$ kubens
default
federated-ns
kube-federation-system
kube-node-lease
kube-public
kube-system
pc-7-24:Kubernetes-Monitoring sedatceyhan$
```

Figure 5.20: Namespace called "federated-ns" automatically created in Local Minikube

When we deploy resources in the "federated-ns" namespace of our GKE host-cluster, those resources will be automatically propagated to the "federated-ns" namespace of the local Minikube cluster.

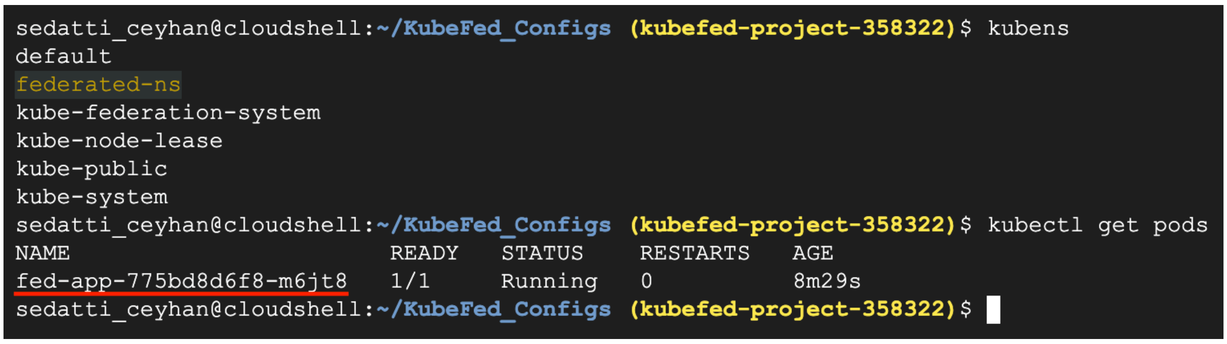
We will now create a simple FederatedDeployment object in the "federated-ns" namespace of our GKE-host cluster. This is a simple web application that outputs a "Hello" message. Once we deploy this application in the GKE host-cluster, we will see the creation of the exact same application in the "federated-ns" namespace of our member clusters (e.g., local Minikube cluster and the self-federated GKE host-cluster). The FederatedDeployment object will be propagated to the member clusters as regular Deployment objects. Below is the FederatedDeployment object we are deploying in the GKE host-cluster:



```yaml
apiVersion: types.kubefed.io/v1beta1
kind: FederatedDeployment
metadata:
  name: fed-app
  namespace: federated-ns
spec:
  placement:
    clusterSelector:
      matchLabels: {}
  template:
    spec:
      selector:
        matchLabels:
          app: fed-app
      template:
        metadata:
          labels:
            app: fed-app
        spec:
          containers:
          - image: gcr.io/google-samples/hello-app:2.0
            name: echo
            ports:
            - containerPort: 8080
              name: http
          nodeSelector:
            nodeToDeploy: 'yes'
```

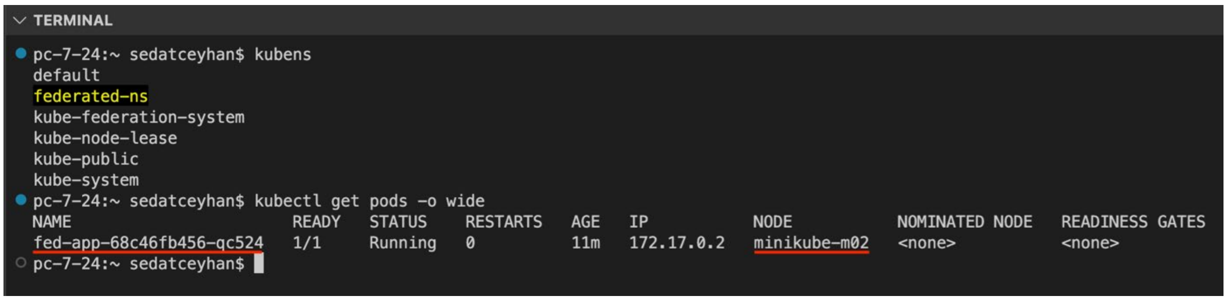
Figure 5.21: Google-Hello FederatedDeployment created in the GKE Host-Cluster

We can observe the “**fed-app**” application pods being created in both the local Minikube cluster and the GKE host-cluster. The first figure below shows the creation of the “fed-app” pods in the GKE host-cluster (since it is a member cluster of its own) and the second figure shows the creation of the pods in our local Minikube cluster residing in Europe:



```
sedatti_ceyhan@cloudshell:~/KubeFed_Configs (kubefed-project-358322)$ kubens
default
federated-ns
kube-federation-system
kube-node-lease
kube-public
kube-system
sedatti_ceyhan@cloudshell:~/KubeFed_Configs (kubefed-project-358322)$ kubectl get pods
NAME                       READY   STATUS    RESTARTS   AGE
fed-app-775bd8d6f8-m6jt8   1/1     Running   0          8m29s
sedatti_ceyhan@cloudshell:~/KubeFed_Configs (kubefed-project-358322)$
```

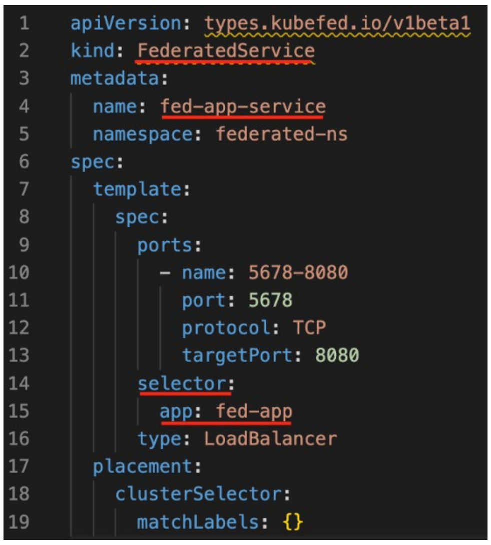
Figure 5.22: Deployment pods automatically created in the GKE host-cluster



```
TERMINAL
pc-7-24:~ sedatceyhan$ kubens
default
federated-ns
kube-federation-system
kube-node-lease
kube-public
kube-system
pc-7-24:~ sedatceyhan$ kubectl get pods -o wide
NAME                       READY   STATUS    RESTARTS   AGE   IP           NODE           NOMINATED NODE   READINESS GATES
fed-app-68c46fb456-qc524   1/1     Running   0          11m   172.17.0.2   minikube-m02   <none>           <none>
pc-7-24:~ sedatceyhan$
```

Figure 5.23: Deployment pods automatically created in our Local Minikube Cluster

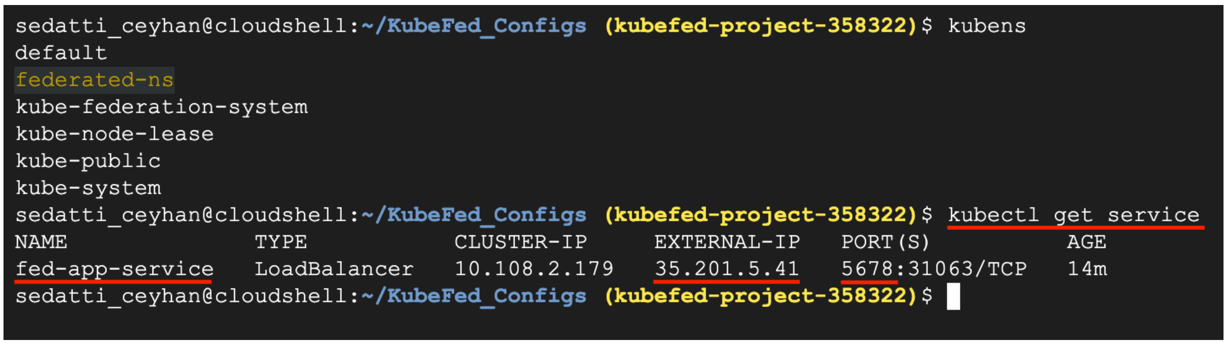
To expose our Deployment, we will attach a Service object to it. Again, we will create a FederatedService resource in the “federated-ns” namespace of our GKE-host cluster and consequently, the FederatedService resource will be created as a regular Service object in the “federated-ns” namespace of our member clusters (e.g., local Minikube cluster and the self-federated GKE host-cluster). Below is our FederatedService resource. Notice that we’re exposing our service at port=5678, and our targetPort=8080 is the container port of our deployment in Figure 5.21. This FederatedService will be propagated to our member clusters as a regular Service object and be attached to the “fed-app” deployment residing in both the local Minikube and GKE host-cluster.



```yaml
apiVersion: types.kubefed.io/v1beta1
kind: FederatedService
metadata:
  name: fed-app-service
  namespace: federated-ns
spec:
  template:
    spec:
      ports:
        - name: 5678-8080
          port: 5678
          protocol: TCP
          targetPort: 8080
      selector:
        app: fed-app
      type: LoadBalancer
  placement:
    clusterSelector:
      matchLabels: {}
```

Figure 5.24: Google-Hello FederatedService created in the GKE Host-Cluster

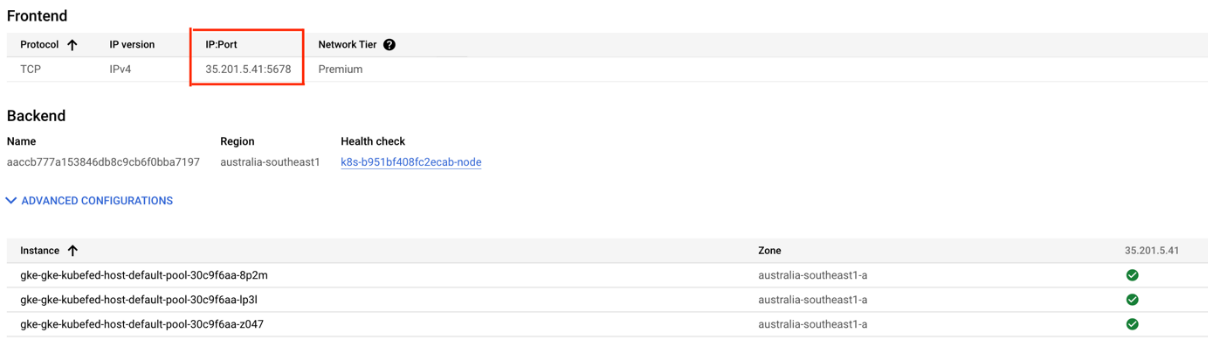
Notice that in the selector field, we have “**app: fed-app**”. This is how we are attaching this service to the deployment in Figure 5.21. Deploying this FederatedService object in the GKE host-cluster will result in an automatic deployment of “**fed-app-service**” Service objects in the member clusters; these new Service objects will be attached to both member clusters’ “fed-app” deployments. These services are used for exposing the “fed-app” deployments. Moreover, we deployed the Service object with type=LoadBalancer as you can see from Figure 5.24. Since the Service is of type LoadBalancer, the Google Platform automatically created a LoadBalancer from Australia region and assigned this automatically created LoadBalancer's external IP address to the Service object created in the GKE host-cluster.



```
sedatti_ceyhan@cloudshell:~/KubeFed_Configs (kubefed-project-358322)$ kubens
default
federated-ns
kube-federation-system
kube-node-lease
kube-public
kube-system
sedatti_ceyhan@cloudshell:~/KubeFed_Configs (kubefed-project-358322)$ kubectl get service
NAME              TYPE           CLUSTER-IP      EXTERNAL-IP    PORT(S)          AGE
fed-app-service   LoadBalancer   10.108.2.179    35.201.5.41    5678:31063/TCP   14m
sedatti_ceyhan@cloudshell:~/KubeFed_Configs (kubefed-project-358322)$
```

Figure 5.25: Service object automatically created in the GKE host-cluster

Notice in Figure 5.25 that the "**fed-app-service**" Service created in GKE host-cluster is assigned an external IP address. This IP address corresponds to the automatically created LoadBalancer in Australia region:



Frontend

Protocol ↑	IP version	IP:Port	Network Tier
TCP	IPv4	35.201.5.41:5678	Premium

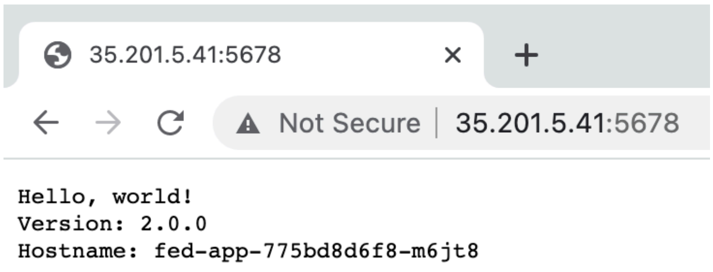
Backend

Name	Region	Health check
aaccb777a153846db8c9cb6f0bba7197	australia-southeast1	k8s-b951bf408fc2ecab-node

Instance ↑	Zone	35.201.5.41
gke-gke-kubefed-host-default-pool-30c9f6aa-8p2m	australia-southeast1-a	✓
gke-gke-kubefed-host-default-pool-30c9f6aa-lp3l	australia-southeast1-a	✓
gke-gke-kubefed-host-default-pool-30c9f6aa-z047	australia-southeast1-a	✓

Figure 5.26: Load-Balancer created for "fed-app-service" Service of GKE cluster

We can now reach the "**fed-app**" application running in the GKE-host cluster using the external IP address specified by the LoadBalancer, and the port number of 5678 specified in the Service configuration file in Figure 5.24. Hitting the web app running at 35.201.5.41:5678, we get the following result:

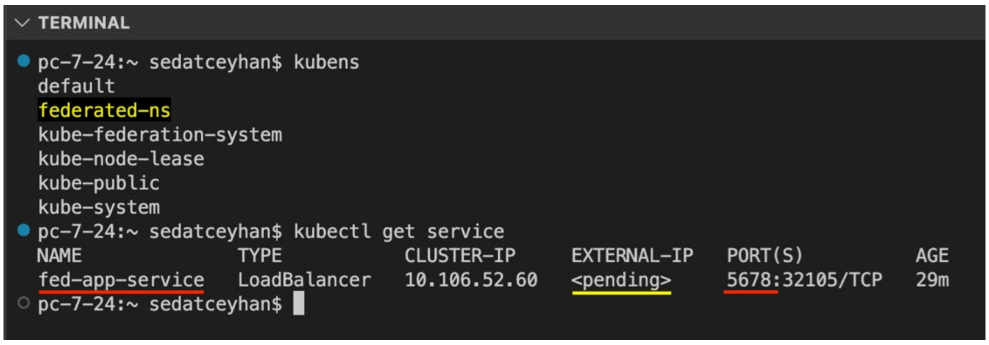


```
Hello, world!
Version: 2.0.0
Hostname: fed-app-775bd8d6f8-m6jt8
```

Figure 5.27: Reaching the "fed-app" deployment running in GKE host-cluster (AU)

We can successfully access the application deployed on the GKE host-cluster.

Since the local Minikube cluster is one of the two federated member clusters, the same Service object is also deployed in the "federated-ns" namespace of our local Minikube cluster. However, unlike the GKE host-cluster, there is no external IP address automatically assigned to our Minikube cluster's Service object. Thus, we cannot access to the "**fed-app**" application deployed in local Minikube cluster (EU) externally:



```
pc-7-24:~ sedatceyhan$ kubens
default
federated-ns
kube-federation-system
kube-node-lease
kube-public
kube-system
pc-7-24:~ sedatceyhan$ kubectl get service
NAME              TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
fed-app-service   LoadBalancer   10.106.52.60   <pending>     5678:32105/TCP   29m
pc-7-24:~ sedatceyhan$
```

Figure 5.28: Service object automatically created in the local Minikube cluster (w/o public IP)

This brings us to the next requirement: exposing the "**fed-app-service**" Service deployed to our local Minikube cluster.

5.2.6 Exposing "fed-app-service" of Local Minikube

We now have a KubeFed host-cluster (central GKE cluster) and two member clusters: local Minikube and the central GKE host-cluster itself. Since the GKE host-cluster is residing in Google Cloud Platform, its Service object was automatically assigned an external IP address through a Google Load Balancer located in Australia. We can thus access the "Google Hello" web-app deployed in the GKE host-cluster as can be seen from Figure 5.27.

However, the Federated Service object propagated to our local Minikube cluster is not assigned any external IP address. Hence, we cannot connect to the "Google Hello" web-app deployed in the Minikube cluster as can be seen in Figure 5.28. To solve this problem, we need to expose the Federated Service object of our local Minikube cluster externally in some way. We will follow the same approach we had for externally exposing the Minikube API server, namely we will use a second ngrok agent in addition to the one we are using for exposing the API server. We will receive a second public endpoint to which we can make external requests (e.g., requests to the local "fed-app-service" Service object). This ngrok public endpoint will forward our external requests to our local service (e.g., local "fed-app-service" Service). The external requests to this additional ngrok endpoint will be forwarded to our second ngrok agent responsible for sending the requests to our local upstream service.

Before setting up the second ngrok tunnelling for our "fed-app-service" resource, we need to do a small forwarding. Remember from our "fed-app-service" FederatedService configuration file in Figure 5.24 that our service is listening on port 5678. However, the target port=8080 (Figure 5.24) is the container port of the "fed-app" deployment (Figure 5.21) that we are interested in. More specifically, targetPort=8080 is the port on which our example application "fed-app" listens; it is the container port to which our Kubernetes Service "fed-app-service" should forward requests. Hence, we need to forward all requests from the service port (5678) to the container port (8080):

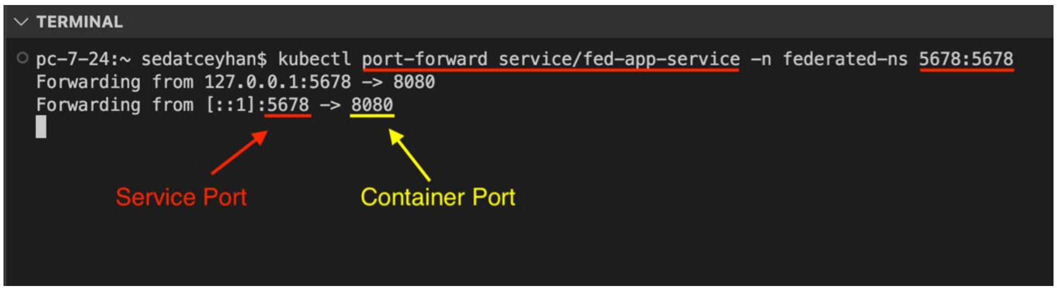
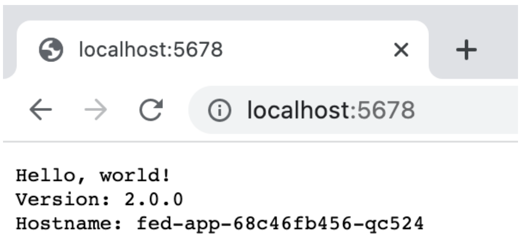

```
pc-7-24:~ sedatceyhan$ kubectl port-forward service/fed-app-service -n federated-ns 5678:5678
Forwarding from 127.0.0.1:5678 -> 8080
Forwarding from [::1]:5678 -> 8080
```

Figure 5.29: Forwarding requests from Service Port to Container Port

We are now forwarding all requests from the service port (5678) to the container port of our "fed-app" deployment (8080) in Minikube. We can now reach our application locally via http://localhost:5678. Below is the same "fed-app" application we are accessing locally in our Minikube cluster. This is the same application we previously accessed in GKE host-cluster using GCP Load Balancer (Figure 5.27).



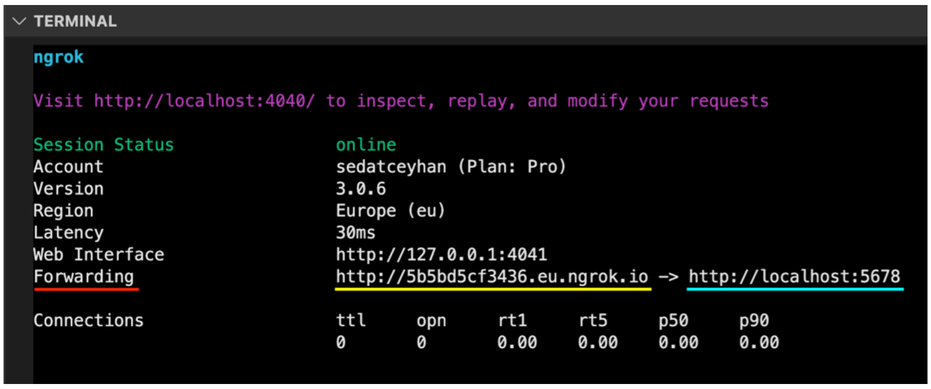
```
Hello, world!
Version: 2.0.0
Hostname: fed-app-68c46fb456-qc524
```

Figure 5.30: Reaching the "fed-app" deployment running locally in Minikube (EU)

Our Federated Service "fed-app-service" is now listening to the incoming local requests on port 5678. The Service object propagated to the member Minikube cluster is now exposed locally via `http://localhost:5678`. Similar to how we exposed our Minikube API server externally, we need to expose our "fed-app-service" resource residing in the local Minikube cluster externally as well. As mentioned previously, we will use a second ngrok tunnelling agent. ngrok will forward requests made to its generated public endpoint onto `http://localhost:5678`. When the end-users make requests to the public ngrok endpoint, the requests will be automatically forwarded to `http://localhost:5678`.

We run the following command to create a ngrok public endpoint and forward all requests made to this public endpoint onto our "fed-app-service" resource running locally at `http://localhost:5678`:

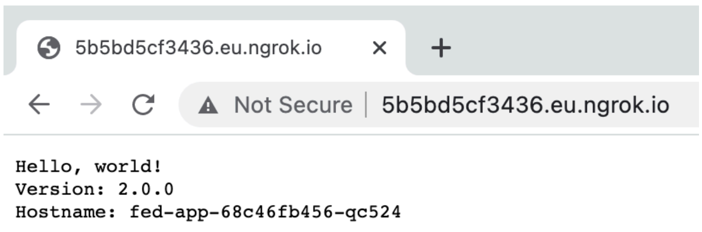
```
ngrok http --scheme=http 5678
```



```
TERMINAL
ngrok

Visit http://localhost:4040/ to inspect, replay, and modify your requests

Session Status                online
Account                       sedatceyhan (Plan: Pro)
Version                       3.0.6
Region                        Europe (eu)
Latency                       30ms
Web Interface                 http://127.0.0.1:4041
Forwarding                    http://5b5bd5cf3436.eu.ngrok.io -> http://localhost:5678

Connections                   ttl     opn     rt1     rt5     p50     p90
                              0       0       0.00    0.00    0.00    0.00
```

Figure 5.31: ngrok URL forwarding requests to the Local Service of Minikube

The URL that's underlined in yellow above is the second ngrok generated public endpoint that we will use to access our "fed-app-service" resource running in our local Minikube cluster. The blue-underlined address is the local "fed-app-service" resource to which all the requests made to the ngrok generated URL will be forwarded. We can now access the "fed-app" application running in our local Minikube cluster externally:



```
Hello, world!
Version: 2.0.0
Hostname: fed-app-68c46fb456-qc524
```

Figure 5.32: Reaching the "fed-app" deployment running in Local Minikube Externally

5.3 Summary

In this chapter, we have successfully created a solution in which the functionalities and resources of a central cluster residing in the Cloud are automatically propagated to a local cluster living in an independent machine from a different region. These local clusters can reside in small computing machines all around the world and receive their configurations and resources from a central cluster. End-users can make their requests to the local machines located in closer branches and be provided the same services residing in the central cluster located further away from them. To achieve this, we used a MacBook Pro (2017) as our local machine and a Minikube cluster as the local cluster. They are located in Dublin, Europe. We used the Google Cloud Platform as our central Cloud and created a GKE cluster located in Australia. Before federating our local Minikube cluster from the central GKE cluster, we established a connectivity between the two environments. We created certificates for the Minikube and shared it with GKE, and then exposed our Minikube API server using ngrok tunnelling tool. ngrok forwarded the requests made to the generated URL onto our Minikube's local API server. After having established connectivity, we started federating the Minikube cluster from the central GKE cluster using the KubeFed tool. We scheduled a simple deployment and a service in the GKE cluster and observed the successful propagation of the same resources into our member clusters (e.g., local Minikube cluster and the GKE cluster itself). GCP automatically created a Load Balancer in the Australian region for the federated service object deployed onto the GKE cluster, and we reached the web application using the Load Balancer's IP. For our Minikube cluster, we used another ngrok agent to create an additional endpoint for the local service. ngrok forwarded the requests made to the generated URL onto our local service, and we exposed the same application running on the Minikube as well. At the end of our implementation, we had the same web application running on both our local Minikube cluster (EU) and the central GKE cluster (AU), both accessible externally.

Chapter 6

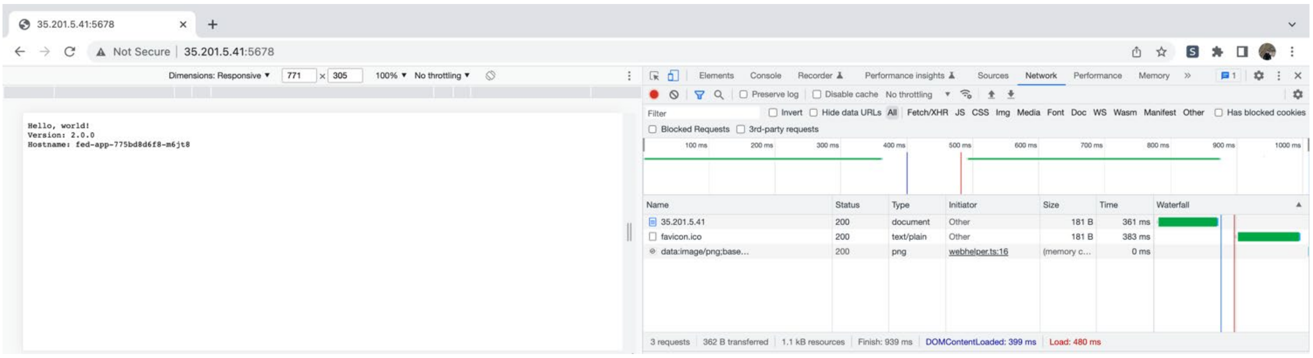
Evaluation

The preceding chapter covered the suggested solution together with the requirements that needed to be fulfilled for a working POC solution. We then described the components used to fulfill those requirements. In this chapter, we will evaluate the POC described in the previous chapter. We deployed the same application on a central GKE cluster in Australia and on a local Minikube cluster living on a machine located in Europe. For our initial evaluation, we will compare the latencies of these applications

6.1 Latency

In our solution, we are providing the same resources and functionalities of a central cluster to a local cluster residing in a different region. When the end-user wants to reach the application running on the central cluster, they will make their requests to the closest local machines near them. In our case, the central cluster is represented by a GKE cluster hosted in Google's Australia region and our local cluster is a Minikube cluster running on a MacBook Pro (2017) machine located in Dublin. Remember from our design & implementation section that we are running the same deployment resource (fed-app) on both clusters. The containerized application is a simple "Google-Hello" web application in our case. Since the application hosted by our Minikube cluster is located in Europe, we are expecting to receive lower latency compared to the application hosted by the central GKE cluster in Australia.

Our first experiment will consist of us making requests to both applications from our local machine in Dublin. We will check the latencies of our requests using Google Chrome's Developer Tools. In particular, we will check the time it takes to receive the results of our requests through the "Network" section of the Developer tools. For our second test, we will use a third-party tool called "dotcom-tools" [72] to check the latencies of both web-servers from a European region other than ours. Below are the results of our first experiment:


35.201.5.41:5678

Not Secure | 35.201.5.41:5678

Dimensions: Responsive 771 x 305 100% No throttling

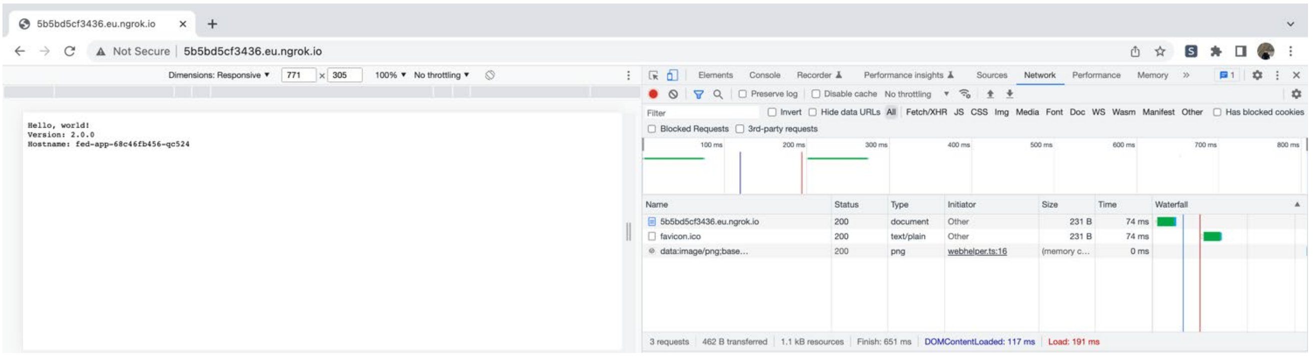
```
Hello, world!
Version: 2.0.0
Hostname: fed-app-775bd8d6f8-m6jt8
```

Name	Status	Type	Initiator	Size	Time	Waterfall
35.201.5.41	200	document	Other	181 B	361 ms	
favicon.ico	200	text/plain	Other	181 B	383 ms	
data:image/png;base…	200	png	webhelper.ts:16	(memory c…	0 ms	

3 requests | 362 B transferred | 1.1 kB resources | Finish: 939 ms | DOMContentLoaded: 399 ms | Load: 480 ms

Figure 6.1: Latency of the central GKE cluster hosted in AU

The figure above reflects the time it takes to get the contents of the web-app hosted in our central GKE cluster (AU). We are making the "GET" request from our own local machine in Dublin. Notice from figure 6.1 that the time it takes to get the contents of the web-app is approximately 375 ms. The high latency was expected since the central cluster is located in a completely different continent. Now, we will make the same experiment with the web application running on our local Minikube cluster:


5b5bd5cf3436.eu.ngrok.io

Not Secure | 5b5bd5cf3436.eu.ngrok.io

Dimensions: Responsive 771 x 305 100% No throttling

```
Hello, world!
Version: 2.0.0
Hostname: fed-app-68c46fb456-qc524
```

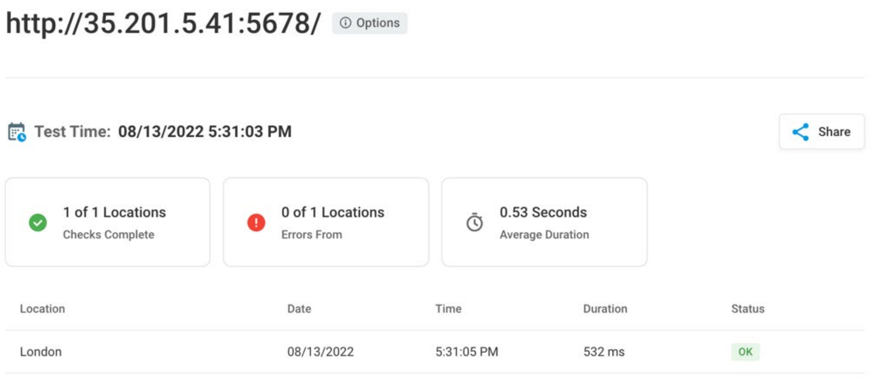
Name	Status	Type	Initiator	Size	Time	Waterfall
5b5bd5cf3436.eu.ngrok.io	200	document	Other	231 B	74 ms	
favicon.ico	200	text/plain	Other	231 B	74 ms	
data:image/png;base…	200	png	webhelper.ts:16	(memory c…	0 ms	

3 requests | 462 B transferred | 1.1 kB resources | Finish: 651 ms | DOMContentLoaded: 117 ms | Load: 191 ms

Figure 6.2: Latency of the Minikube cluster hosted in Dublin

The figure above reflects the time it takes to get the contents of the web-app hosted in our local Minikube cluster (EU). We are making the "GET" request from our own local machine in Dublin and as expected the latency for getting the contents of the web-app is significantly lower than the central GKE cluster. The latency in this case is 74 ms.

In our next experiment, we will use the "dotcom-tools" [72] and check the latencies of our applications from a London region. This tool makes several requests to a web-application in question from different points in that region and takes the average of all the latencies to return a final result. Below are the results of our second experiment:

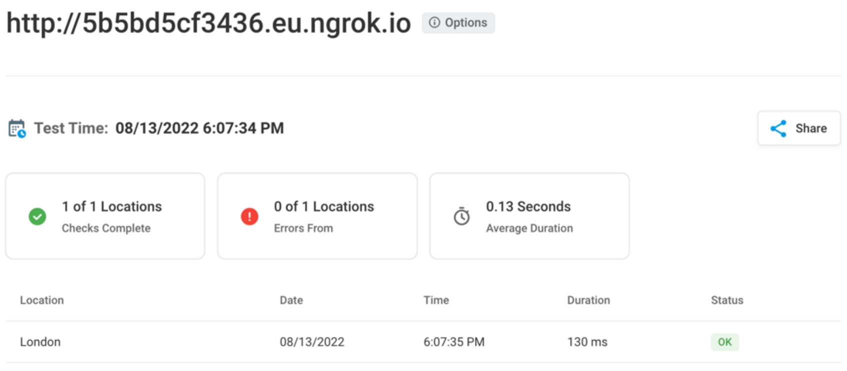

http://35.201.5.41:5678/

Test Time: 08/13/2022 5:31:03 PM

1 of 1 Locations Checks Complete | 0 of 1 Locations Errors From | 0.53 Seconds Average Duration

Location	Date	Time	Duration	Status
London	08/13/2022	5:31:05 PM	532 ms	OK

Figure 6.3: Latency of the central GKE Cluster hosted in AU from London


http://5b5bd5cf3436.eu.ngrok.io

Test Time: 08/13/2022 6:07:34 PM

1 of 1 Locations Checks Complete | 0 of 1 Locations Errors From | 0.13 Seconds Average Duration

Location	Date	Time	Duration	Status
London	08/13/2022	6:07:35 PM	130 ms	OK

Figure 6.4: Latency of the Local Minikube Cluster hosted in EU from London

The average latency to the web-app hosted in the central GKE cluster (AU) from the London region is approximately 532 ms [Figure 6.3] and latency to the web-app hosted in the local Minikube cluster (EU-Dublin) is approximately 130 ms. The results conform to our previous experiment.

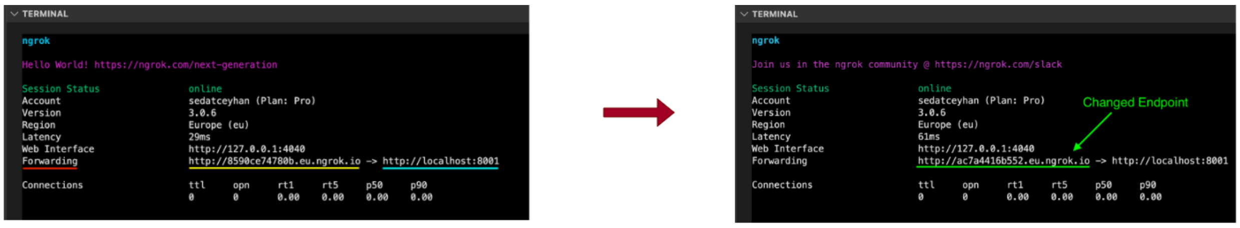
These experiments show that we can propagate the resources and functionalities of our central Cloud cluster onto many local machines near the end-users and can expect a lower latency for the same application when the end-users make their requests to the local machines. Since these local machines are small computing processes, they can be numerous and located in large numbers. In such a scenario, there would always be a local machine with the central cluster's resources near the end-user, which would result in lower latencies. We showed through our design & implementation together with the latency experiments that, in an ideal scenario where all our local machines are resilient to failures, located in large numbers in several regions and are able to successfully host the resources of the central cluster, we would be creating a highly available, low latency environment for the end-users. However, this is not the case. Our POC is not fully resilient as we will demonstrate in the next section.

6.2 Resiliency

Our goal in this thesis was to create an environment where the local clusters living in small computing machines are connected to a central cluster and have the same resources and functionalities of the central cluster living in the Cloud through federation. We also wanted to show that when we access the same application running on the closer local cluster, we would get lower latencies, and we achieved our goal. We did not focus on the resiliency of the solution. Here we will show some of the issues that exist in a few of our components, and in the next chapter we will talk about potential future work that can help mitigate these issues.

6.2.1 Disconnection of ngrok agents

Remember from the sections 5.2.3 and 5.2.6 in Chapter 5 that we used ngrok tunnelling tool to expose our local Minikube cluster's API server and "fed-app-service" resource respectively. We used two ngrok agents to create two public endpoints. These public endpoints forwarded all the requests made to our local Minikube cluster to the corresponding local addresses (e.g, local Minikube API server and the local service resource). However, if the ngrok tunnel connection drops and we reestablish the connection, the ngrok endpoint will change to another URL. The tunnel connection will continue; however, the external requests will fail. This is because we changed the server API address of our local Minikube in the "kubeconfig_merged" file with the previous ngrok endpoint [Figure 5.10]. However, this endpoint is no longer in use and there is a new ngrok endpoint assigned to the API server. Thus, the central cluster will no longer federate the local Minikube cluster unless the API server of the Minikube is changed with the new ngrok url in the "kubeconfig_merged" file of the central cluster. Below is an example diagram showing the change of ngrok endpoint for the Minikube API Server:



```
ngrok

Hello World! https://ngrok.com/next-generation

Session Status                online
Account                       sedatceyhan (Plan: Pro)
Version                       3.0.6
Region                        Europe (eu)
Latency                       29ms
Web Interface                 http://127.0.0.1:4040
Forwarding                    http://8590ce74780b.eu.ngrok.io -> http://localhost:8001

Connections                   ttl     opn     rt1     rt5     p50     p90
                              0       0       0.00    0.00    0.00    0.00
```

```
ngrok

Join us in the ngrok community @ https://ngrok.com/slack

Session Status                online
Account                       sedatceyhan (Plan: Pro)
Version                       3.0.6
Region                        Europe (eu)
Latency                       61ms
Web Interface                 http://127.0.0.1:4040
Forwarding                    http://ac7a4416b552.eu.ngrok.io -> http://localhost:8001

Connections                   ttl     opn     rt1     rt5     p50     p90
                              0       0       0.00    0.00    0.00    0.00
```

Figure 6.5: Changed ngrok Endpoint for the Minikube API Server

The API Endpoint initially provided by the ngrok service no longer exists, and it is changed with a completely new URL. For the central GKE cluster to make successful queries to the Minikube API server, the old URL needs to be changed with the new endpoint in GKE's "kubeconfig_merged". Since this change is not propagated automatically, manual work may be required. Since this will prevent the central GKE cluster from making successful queries to our local Minikube's API server, the Federation will be disrupted. The lack of automation for the replacement of our ngrok Endpoints is an issue in our solution. We will talk about potential solutions in the "Future Work" Chapter.

6.2.2 Disconnection of the Central GKE Cluster

For Cluster Federation, we are using the KubeFed Federation tool [45]. As we know, KubeFed consists of one host cluster and one or more member clusters. The host cluster schedules resources and they are automatically propagated to the member clusters. However, since all the KubeFed controller managers reside in the host cluster, if the host cluster fails, the member clusters will no longer be federated. We checked that the already deployed resources will continue to exist in the local Minikube cluster, but we will no longer be able to federate new resources across the member clusters using the central host cluster. This presents a single point of failure in our POC. We may have multiple Minikube clusters residing in different local machines. The end-users will continue to access the applications already deployed to the local clusters (e.g., fed-app), but the central host cluster residing in the Cloud will no longer be able to federate new resources across the member clusters. A potential solution to this problem will be discussed in the "Future Work" section. We will briefly discuss the solution offered by Saba Feroz Memon from Aalto University [36].

6.3 Security

We are using ngrok tunnelling to expose the local API Sever of our Minikube cluster and also the local "fed-app-service" object deployed in Minikube. As mentioned in the previous chapter, when the end-user makes a request to this ngrok endpoint, the ngrok edge does encrypt the request and forwards it to our locally running ngrok agent, which then sends this request to our local upstream service. The entire communication between the ngrok edge and our local ngrok agent is secure and encrypted using the TLS protocol. Moreover, the traffic from user to the ngrok edge and traffic from ngrok agent to our locally running service use the same encryption protocols [60]. Thus, we can state that the ngrok tunnel used for externally exposing our local services are secure. However, to locally expose our API server, we ran `kubectl proxy –disable-filter true --address 0.0.0.0` . This causes our local API server to listen to all incoming requests without filtering. We followed this approach to facilitate the creation of our POC, however, this is a potential security issue. In a production use case, we need to whitelist only the required sources.

6.4 Summary

In this chapter, we evaluated the implemented POC solution described in the previous section. We first evaluated the latency of our solution. We concluded that requests made to our local Minikube cluster significantly decreased the latency compared to those made to the central GKE cluster located in the Australian region. Through the “Design & Implementation” chapter and the “Latency” section of our “Evaluation” chapter, we showed that we can host the resources of a central Cloud cluster in the local clusters residing in small computing machines located all around the world. Thus, we can provide the same functionalities to the local clusters and also decrease the latency for the end-users. We then evaluated some of the requirements necessary to put our POC solution into production. We mentioned two resiliency issues that our solution currently has. One of them is the temporary endpoints assigned by ngrok to our local services. Since these addresses are not permanent, ngrok generates new endpoints each time there is disconnection, which may disrupt the federation and certain applications. We also mentioned KubeFed’s single point of failure that may disrupt the federation of member clusters. We then talked about the security of our POC. The external requests are all TLS encypted by ngrok, improving the security of our local services. However, our local API Server is not filtering any requests, which may prove to be problematic in production.

Chapter 7

Future Work

In this chapter, we will elaborate on the limitations and issues provided in the “Evaluation” chapter. We will talk about some of the limitations of our POC and try to provide possible approaches to tackle these limitations. We will also point out to the areas that require future work for a more resilient and production ready solution.

7.1 Automating the Solution

As demonstrated in the “Design & Implementation” Chapter, there are lots of moving connectivity parts in our solution. Most of the requirements build on top of each other and any connectivity issue may result in the failure of our POC solution. The implementation required manual entry of certain commands in particular environments (e.g., local terminal and Google Cloud Shell). Moreover, we manually changed certain configurations and forced some of the environments to use the changed configurations. All this requires delicate manual work where the order of actions is critical. Automating such a solution would have lots of benefits, however, it is no trivial task. Our current solution is using ngrok tunnelling, and the automation tool would need to establish the tunnel and use the generated endpoints in different platforms. For instance, the Minikube API Server’s endpoint would need to change automatically in the “kubeconfig” file of the central GKE cluster. Before that, local Minikube’s “kubeconfig” file would need to be uploaded in the central GKE cluster and automatically merged with the central cluster’s “kubeconfig” file. Automating the order of actions together with handling the connectivity failures may require an extensive future work. However, a successful implementation would bring our POC closer to being a production ready solution.

7.2 Resiliency in KubeFed

In KubeFed, the application deployer communicates with the host cluster for the propagation of federated resources among the member clusters. The host cluster ensures that the federated applications on the member clusters are in the desired state through the KubeFed Controller Manager pods. In our solution, we may have a central cluster living in the Cloud and several member clusters living in small computing machines all around the world. As we discussed in the “Evaluation” chapter, the host cluster is a single point of failure. If the host cluster fails, the member clusters will continue to function as expected. All previously federated resources will continue to exist in the member clusters and serve the end-users. However, we would not be able to federate our member clusters any longer, namely we would not be able to automatically create resources in the member clusters from a single point (e.g. host cluster) or ensure that the deployed resources in the member clusters are in the desired state. This is an existing problem in the open-source KubeFed project. In a real-world scenario, we may need multiple host clusters federating our member clusters. When one of the host clusters fails, another host cluster would continue from where the previous cluster left off and starts federating the member clusters, and if the new host cluster also fails, another host cluster would take its place and so on. This would bring resiliency to the federation component in our solution. There is a POC work that focuses on this particular issue. In their work, Saba Feroz Memon [36] describes this problem in detail and suggests a solution that relies on leader election mechanism. As a result, their work adopts a novel strategy by constructing active-passive redundant KubeFed host clusters to ensure resiliency in Kubernetes Federation administration [36]. Multiple host clusters are installed instead of a single host cluster, and one cluster is designated as the leader cluster and operates in the active mode, while the rest of the clusters function as follower clusters (e.g., passive). The leader cluster is similar to our central GKE host cluster, but in addition to its normal functionalities, it replicates all the configuration information of the federated applications onto the follower host clusters. When the leader host cluster fails, one of the follower clusters would become a candidate to turn into the leader cluster. Since all the resources were already replicated, the new leader cluster would be up-to-date and the federation could continue from where it left off. This approach would make the KubeFed component of our solution more resilient and provides high availability of our KubeFed host clusters [36].

7.3 Resource Allocation in KubeFed

If our solution goes into production, we will have thousands of local clusters living in small computing machines located all around the world, and these local clusters would act as the member clusters in the federation. In our POC, we had only two member clusters and one of them was the local Minikube cluster residing in MacBookPro (2017) (EU). Our goal is to have local clusters with the resources and functionalities of a central host cluster all around the world so that the latency of the end-users would be minimal. However, in its current form, this is not quite possible with KubeFed. One of the reasons is the lack of resiliency in KubeFed as explained in 7.2. Another one is due to KubeFed’s limited support for the automated policy-based scheduling. Currently, KubeFed allows application deployers to choose the clusters in which to manually host the federated resources. The problem with this is that KubeFed simply propagates federated applications to the member clusters without any prior checks on whether the member clusters have enough resources. We discussed that in our solution the member clusters are resided in small to medium computing machines, and in a real-world scenario, the local machines can be of any type. However, KubeFed treats all the member clusters as homogenous entities. Thus, KubeFed currently cannot scale to manage hundreds or thousands of member clusters. A more intelligent scheduling scheme may be implemented in KubeFed to avoid any resource misplacement or wastage. Future work should focus on building a multi-cluster scheduling solution that assigns the deployments scheduled on the central host cluster onto one or more clusters from a specific set of member clusters that have enough available resources and capacity to match the deployment request made by the host cluster.

7.4 Disconnection of API Server

To build our POC, we used the simplest tool to expose our local services (e.g., local API Server and the local applications). We used ngrok tunneling tool to generate a public endpoint responsible for forwarding the requests to our local services. However, since these public endpoints are ephemeral, they are regenerated in case of disconnection. Thus, the requests made to the old ngrok public endpoints fail. To solve this issue, we manually changed the old URL with the new public endpoint in the “kubeconfig” file of our GKE host cluster. If this manual work can be replaced with some sort of automation as described in section 7.1, our solution would be more resilient. Other solutions may include static public endpoints assigned to our local services or the use of VPN instead of tunnelling.

7.5 Summary

In this chapter we talked about certain concepts and tools used in our POC that require future work for a production ready solution. Our POC’s aim was to show that we can provide the same resources and functionalities of a central cluster residing in the Cloud to a local cluster located in a computing machine from a different region through KubeFed tool. We also showed that end-users can make their requests to local clusters instead of the central cluster (as they have the same applications) to decrease latency. However, in a real-world use case, we will have thousands of local clusters living in different local machines from different regions. If our POC can be adapted to this scenario, our solution would provide many advantages, but there are certain parts that would require future work for our POC to become feasible in production. One of these requirements is improving KubeFed’s resiliency and resource scheduling logic. Without these requirements, KubeFed cannot be used in a production scenario. Another area that may require future work is the automation of the entire process. However, since there are lots of moving connectivity parts in the solution, this may prove to be challenging. Finally, we discussed using a static public IP address or VPN for exposing the local services running on local clusters.

Chapter 8

Conclusion

This research aimed to determine whether we could use a Cluster Federation tool to create a solution where the functionalities and resources of a central cluster in the Cloud are automatically propagated to local clusters residing in a smaller independent machine in a separate location.

The research achieved this through:

- Using a Kubernetes Cluster Federation tool called KubeFed
- Creating a Minikube Cluster in a local machine (MacBook Pro) residing in Dublin
- Externally exposing the Minikube API Server through tunnelling
- Connecting the central GKE cluster located in the Australian region with the local Minikube
- Installing all the KubeFed tools on the central GKE cluster
- Federating the member clusters (e.g., local Minikube cluster) and scheduling federated resources in the member clusters through the central GKE cluster
- Exposing the resources on the local Minikube Cluster
- Comparing latencies of the same application residing both in the Minikube (EU) and the central GKE cluster (AU)

We have shown that with a resilient implementation of such setup with thousands of local clusters all around the world, we can easily deploy the same resources automatically across the local clusters through a single central cluster in the Cloud. On top of providing the same resources and functionalities across many local clusters through a central control plane, this would decrease the latency for the end-users who can access the same resources from the closest local machine. However, improving the resiliency of the POC is no trivial task and is required for the solution to go in production. As mentioned in the previous chapter on "Future Work", there are several areas that need further research and improvements. Most of these problems originate from KubeFed's lack of maturity. Currently, the KubeFed project is still in development and requires an extensive work to improve its resiliency to support the federation of thousands of member clusters at the same time.

Bibliography

[1] *What is a container.* URL: https://www.docker.com/resources/what-container/

[2] *Containerization vs Virtualization.* URL: https://www.burwood.com/blog-archive/containerization-vs-virtualization

[3] *Comparison of orchestration tools.* URL: https://rafay.co/the-kubernetes-current/container-orchestration-tools-comparison

[4] *Containerization.* URL: https://www.ibm.com/cloud/learn/containerization

[5] *Multiple Clusters vs Single Cluster*. URL: https://www.containiq.com/post/kubernetes-multiple-clusters-vs-single-cluster

[6] *Multi-Cluster Kubernetes rationale.* URL: https://platform9.com/blog/multi-cluster-kubernetes-deployments-when-and-why/

[7] *Admiralty Federation.* URL: https://caylent.com/blog/kubernetes/kubernetes-cluster-federation-with-admiralty

[8] *Google Anthos.* URL: https://cloud.google.com/anthos/docs/tutorials/explore-anthos

[9] *Anthos Concepts.* URL: https://cloud.google.com/anthos/docs/concepts/overview

[10] *History of Kubernetes.* URL: https://blog.risingstack.com/the-history-of-kubernetes/

[11] *Kubernetes Concepts Overview.* URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

[12] *K8s Advantages & Disadvantages.* URL: https://devspace.cloud/blog/2019/10/31/advantages-and-disadvantages-of-kubernetes

[13] *K8s Nodes.* URL https://kubernetes.io/docs/concepts/architecture/nodes/

[14] *K8s Cluster.* URL: https://www.vmware.com/topics/glossary/content/kubernetes-cluster.html

[15] *K8s Container.* URL: https://kubernetes.io/docs/concepts/containers/

[16] *K8s Pod.* URL: https://kubernetes.io/docs/concepts/workloads/pods/

[17] *K8s Components.* URL: https://kubernetes.io/docs/concepts/overview/components/

[18] *K8s kube-scheduler.* URL https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

[19] *K8s kube-controller-manager*. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/

[20] *K8s etcd.* URL: https://etcd.io/

[21] *K8s container-runtime.* URL: https://kubernetes.io/docs/setup/production-environment/container-runtimes/

[22] *K8s CRI.* URL: https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/

[23] *K8s kubelet.* URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet

[24] *K8s objects.* URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/

[25] *K8s namespaces.* URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/

[26] *K8s replicaset.* URL: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/

[27] *K8s deployment VMware example.* URL: https://www.vmware.com/topics/glossary/content/kubernetes-deployment.html

[28] *K8s deployment.* URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

[29] *K8s service.* URL: https://kubernetes.io/docs/concepts/services-networking/service/

[30] *K8s configmap*. URL: https://www.aquasec.com/cloud-native-academy/kubernetes-101/kubernetes-configmap/

[31] *K8s labels & selectors*. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/

[32] *Assigning workloads to nodes.* URL: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/

[33] *K8s Custom Resource Definitions.* URL: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

[34] *K8s CRDs Openshift def.* URL: https://docs.openshift.com/aro/3/dev_guide/creating_crd_objects.html

[35] *K8s multi-cluster.* URL: https://www.mirantis.com/cloud-native-concepts/getting-started-with-kubernetes/what-is-kubernetes-multi-cluster/#why_multi_cluster

[36] *KubeFed Resiliency Thesis*. URL: https://aaltodoc.aalto.fi/handle/123456789/110500

[37] *K8s Multi-Cluster.* URL: https://www.getambassador.io/learn/multi-cluster-kubernetes/

[38] *K8s single vs multi-cluster.* URL: https://blog.equinix.com/blog/2020/05/26/is-just-one-kubernetes-cluster-enough/

[39] *Comparison of cluster amounts.* URL: https://learnk8s.io/how-many-clusters

[40] *Istio Federation tool.* URL: https://www.revolgy.com/insights/blog/istio-multi-cluster-federation-and-hybrid-cloud

[41] *Cluster Federation Advantages*. URL: https://unofficial-kubernetes.readthedocs.io/en/latest/concepts/cluster-administration/federation-service-discovery/

[42] *Cluster Federation Advantages*. URL: https://unofficial-kubernetes.readthedocs.io/en/latest/concepts/cluster-administration/federation/

[43] *Cluster API.* URL: https://cluster-api.sigs.k8s.io/

[44] *Google Anthos how to.* URL: https://cloud.google.com/anthos/clusters/docs/attached/how-to/attach-kubernetes-clusters

[45] *KubeFed official doc.* URL: https://github.com/kubernetes-sigs/kubefed

[46] *KubeFed detailed overview.* URL: https://www.sobyte.net/post/2022-03/kuberentes-federation/

[47] *Federation Overview.* URL: https://people.wikimedia.org/~jayme/k8s-docs/v1.16/docs/concepts/cluster-administration/federation/

[48] *Cilium* URL: https://docs.cilium.io/en/stable/intro/

[49] *Cilium Federation.* URL: https://cilium.io/blog/2019/03/12/clustermesh/

[50] *K8s network.* URL: https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/

[51] *Network Service Mesh.* URL: https://networkservicemesh.io/

[52] *Submariner Federation.* URL: https://submariner.io/getting-started/architecture/

[53] *Consul.* URL: https://www.consul.io/docs/connect

[54] *Consul Federation.* URL: https://www.consul.io/docs/k8s/installation/multi-cluster/kubernetes

[55] *Shipper Federation.* URL: https://github.com/bookingcom/shipper

[56] *Admiralty Federation Source.* URL: https://github.com/admiraltyio/admiralty

[57] *Admiralty Federation Overview.* URL: https://admiralty.io/docs/quick_start/

[58] *Admiralty Federation – Proxy Pods.* URL: https://admiralty.io/docs/concepts/scheduling/#proxy-pods

[59] *Configure Access to Multiple Clusters.* URL: https://kubernetes.io/docs/tasks/access-application-cluster/configure-access-multiple-clusters/

[60] *Ngrok Tunnelling tool.* URL: https://ngrok.com/

[61] *Cloud Native Computing Foundation.* URL: https://www.cncf.io

[62] Jain, N., and Choudhary, S., "*Overview of virtualization in cloud computing*", 2016 Symposium on Colossal Data Analysis and Networking (CDAN), 2016, pp. 1-4, doi: 10.1109/CDAN.2016.7570950.

[63] Truyen, E., Van Landuyt, D., Reniers, V., Rafique, A., Lagaisse, B., and Joosen, W. "*Towards a container-based architecture for multi-tenant saas applications*", December 12-16 2016, pp. 1-6.

[64] Nguyen, Nguyen & Kim, Taehong. (2021). "*Balanced Leader Distribution Algorithm in Kubernetes Clusters*", Sensors. 21. 869. 10.3390/s21030869, pp. 1-13.

[65] *Reasons Kubernetes won the Container Orchestration War.* https://hackernoon.com/how-did-kubernetes-win-the-container-orchestration-war-lp1l3x01

[66] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., & Wilkes, J. (n.d.), "*Large-scale cluster management at Google with Borg"*, Bordeaux, France, 2015

[67] Schwarzkopf, M, Konwinski, A, Abd-El-Malek, M, Wilkes, J., "*Omega: flexible, scalable schedulers for large compute clusters"*, Prague, Czech Republic, 2013

[68] *5 Reasons why every CIO should consider Kubernetes. URL*: https://www.sumologic.com/blog/why-use-kubernetes/

[69] Vohra, D., "*Kubernetes Management Design Patterns: With Docker, CoreOS Linux, and Other Platforms*", USA, 2017

[70] F. Faticanti, D. Santoro, S. Cretti and D. Siracusa, "An Application of Kubernetes Cluster Federation in Fog Computing," *2021 24th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, 2021, pp. 89-91, doi: 10.1109/ICIN51074.2021.9385548.

[71] Kim, D.; Muhammad, H.; Kim, E.; Helal, S.; Lee, C. "*TOSCA-Based and Federation-Aware Cloud Orchestration for Kubernetes Container Platform*". *Appl. Sci.* 2019, *9*, 191. https://doi.org/10.3390/app9010191

[72] *Dotcom-tools.* URL: https://www.dotcom-tools.com/web-servers-test

A1 Appendix

In this appendix, we will provide the instructions necessary to create the setup we have in the "Design & Implementation" section (Chapter 5). The local cluster will be the Minikube Cluster and the central Cloud cluster will be the GKE cluster. To create the setup, one will need to install Minikube in their local machine; a GCP account and a GKE cluster in the GCP project.

A1.1 Creating the Minikube and its Certificates

Assuming we have successfully installed kubectl, docker and Minikube on our local Machine, we need to follow these steps in our local terminal:

Deleting existing Minikube to avoid any conflict:

```
minikube delete
```

Starting a local Minikube cluster in our machine:

```
minikube start
```

Adding an additional worker node to our local cluster:

```
minikube node add --worker
```

Labelling the worker node so that the workloads are deployed there:

```
Kubectl label node minikube-m02 nodeToDeploy=yes
```

Resetting all the existing configurations and certificates to avoid any future conflict

```
kubectl config unset users.minikube.client-key
kubectl config unset users.minikube.client-certificate
kubectl config unset clusters.minikube.certificate-authority
```

Creating new client certificates for the Minikube. This will change its kubeconfig

```
kubectl config set users.minikube.client-certificate-data $(base64 -b
0 ~/.minikube/profiles/minikube/client.crt)
```

```
kubectl config set users.minikube.client-key-data $(base64 -b 0
~/.minikube/profiles/minikube/client.key)
```

```
kubectl config set clusters.minikube.insecure-skip-tls-verify true
```

A1.2 Exposing the Minikube

In this part, we will be exposing our local Minikube's API server. The commands below are run in separate terminals. They should be kept alive during the entirety of this demo.

On a separate terminal run the below command to expose port 8001 of Minikube API Server:

```
kubectl proxy --disable-filter true --address 0.0.0.0
```

On a separate terminal run the below command to create a ngrok endpoint and assign this endpoint to our local API Server to expose it externally. This step requires the installation of the ngrok tunnelling agent [ngrok installation]:

```
ngrok http --scheme=http 8001
```

A1.3 Google Cloud Console: Merging Config Files

Before starting this section, we need to create a GKE cluster in the Australian region using GCP. We named this cluster "gke-kubefed-host". This is the host cluster running in the central cloud.

In this section, we create one global 'kubeconfig_merged' file in the gcloud Shell. This config file will contain both the Minikube and the gke-host-cluster contexts. Below are the steps that need to be followed in order. They are all executed in the Google Cloud Shell:

Upload the CONFIG file of the local Minikube cluster to Google Cloud Shell:

We need to upload our Minikube's kubeconfig file to Google Cloud Shell. When uploading the local config file, look for the file located in **~/.kube/config** and upload it to Cloud Shell.

Remove the GCP's own Kube config file to delete all the previous contexts and avoid conflicts

```
rm .kube/config
```

Create a new .kube/config file for the central GKE cluster itself. To do this, right click on the gke cluster's name on the GKE dashboard and click the options next to the cluster name. Copy paste the line to the gcloud Shell. This will connect you to the central GKE cluster and create its own config file. The command would look like this:

```
gcloud container clusters get-credentials gke-kubefed-host --region
us-central1 --project vital-keep-357119
```

Combine the Local Minikube's KubeConfig file with our GKE kubeconfig file to create a final kubeconfig_merged file. This kubeconfig will contain all the information of local minikube cluster and our GKE host cluster

```
KUBECONFIG=~/.kube/config:~/config kubectl config view --flatten >
~/kubeconfig_merged
```

We need to ask GKE to use this newly formed configuration as the new KubeConfig file for the cluster

```
export KUBECONFIG=~/kubeconfig_merged
```

For the GKE to externally access (e.g., query, create, delete) the Minikube cluster, change the API server's local URL with the ngrok endpoint we generated in section A1.2

```
kubectl config set-cluster minikube --server='<ngrok generated url>
```

Changing the context name for the GKE cluster so that it works for KubeFed as KubeFed does not accept underscore in cluster context names. Replace the old context name with your old context name

```
kubectl config rename-context gke_vital-keep-357119_us-central1_gke-
kubefed-host gke-kubefed-host
```

Label one of the nodes of our central GKE cluster to deploy all the workloads on that particular central node. Replace the node name with yours

```
kubectl label node gke-kubefed-host-default-pool-5cb8421a-2zjc
nodeToDeploy=yes
```

A1.4 Installing KubeFed on the gke-host-cluster

In this section, we will install the KubeFed client together with KubeFed itself on our gke-host-cluster residing on the cloud. All the commands in this section are executed in the gcloud Shell.

Install the KubeFed Client. These commands follow each other

```
VERSION=0.9.2

OS=linux

ARCH=amd64

curl -LO https://github.com/kubernetes-
sigs/kubefed/releases/download/v${VERSION}/kubefedctl-${VERSION}-
${OS}-${ARCH}.tgz

tar -zxvf kubefedctl-*.tgz

chmod u+x kubefedctl

sudo mv kubefedctl /usr/local/bin/
```

Now, install the KubeFed tool through helm. This will create a new namespace called "kube-federation-namespace where all the KubeFed controller manager pods are deployed"

```
helm repo add kubefed-
charts  https://raw.githubusercontent.com/kubernetes-
sigs/kubefed/master/charts

helm --namespace kube-federation-system upgrade -i kubefed kubefed-
charts/kubefed --version=0.9.2 --set
controllermanager.clusterHealthCheckPeriod=30s --create-namespace
```

Federate the local Minikube cluster together with the gke-kubefed-host cluster

```
kubefedctl join minikube --cluster-context minikube --host-cluster-
context gke-kubefed-host --v=2

kubefedctl join gke-kubefed-host --cluster-context gke-kubefed-host --
host-cluster-context gke-kubefed-host --v=2
```

Check that both clusters joined the federation

```
kubectl -n kube-federation-system get kubefedclusters
```

A1.5 Application Deployment on gke-host-cluster

In this section, we will deploy the resources on the gke-host-cluster, and as shown in chapter 5, these resources will be automatically created in the member clusters. All the commands are executed in the gcloud Shell.

<u>Create a namespace called "federated-ns" in the gke-host-cluster (global-federated-ns.yaml)</u>

```
cat << EOF | kubectl apply -f -
apiVersion: v1
kind: Namespace
metadata:
  name: federated-ns

EOF
```

<u>Create a FederatedNamespace called "federated-ns" in the gke-host-cluster's "federated-ns" namespace created above (federated-ns.yaml)</u>

```
cat << EOF | kubectl apply -f -
apiVersion: types.kubefed.io/v1beta1
kind: FederatedNamespace
metadata:
  name: federated-ns
  namespace: federated-ns
spec:
  placement:
    clusterSelector:
      matchLabels: {}

EOF
```

Create a FederatedDeployment resource in the FederatedNamespace. This resource will be propagated as a regular deployment resource across the member clusters (federated-googleHello-deployment.yaml)

```
cat << EOF | kubectl apply -f -
apiVersion: types.kubefed.io/v1beta1
kind: FederatedDeployment
metadata:
  name: fed-app
  namespace: federated-ns
spec:
  placement:
    clusterSelector:
      matchLabels: {}
  template:
    spec:
      selector:
        matchLabels:
          app: fed-app
      template:
        metadata:
          labels:
            app: fed-app
        spec:
          containers:
          - image: gcr.io/google-samples/hello-app:2.0
            name: echo
            ports:
            - containerPort: 8080
              name: http
          nodeSelector:
            nodeToDeploy: 'yes'

EOF
```

Create a FederatedService resource in the FederatedNamespace. This resource will be propagated as a regular service resource across the member clusters (federated-googleHello-service.yaml)

```
cat << EOF | kubectl apply -f -
apiVersion: types.kubefed.io/v1beta1
kind: FederatedService
metadata:
  name: fed-app-service
  namespace: federated-ns
spec:
  template:
    spec:
      ports:
        - name: 5678-8080
          port: 5678
          protocol: TCP
          targetPort: 8080
      selector:
        app: fed-app
      type: LoadBalancer
  placement:
    clusterSelector:
      matchLabels: {}

EOF
```

A1.6 Exposing “fed-app-service” of Minikube

The FederatedService resource “fed-app-service” just created above is propagated to our local Minikube cluster as a regular service object. In order for this service object to be globally accessible, we need to expose it. Below commands are run in separate terminals in our local environment. The previous two local terminals running as described in A1.2 should still be running.

On a separate terminal run the below command to forward all requests from the service port (5678) to the container port of our “fed-app” deployment (8080)

```
kubectl port-forward service/fed-app-service -n federated-ns 5678:5678
```

On a separate terminal run the below command to create a second ngrok public endpoint and forward all requests to our “fed-app-service” resource running locally:

```
ngrok http --scheme=http 5678
```