INS Institut für Netzwerke und Sicherheit

OST Eastern Switzerland University of Applied Sciences

Cloud Native Intent Automation

Bachelor Thesis

Department of Computer Science
OST - University of Applied Sciences
Campus Rapperswil-Jona

Spring Term 2023

Authors:	Marc Eberhard, Laurent Dina and Lukas Schlunegger
Supervisor:	Prof. Laurent Metzger
Advisor:	Marco Martinez
External Co-Examiner:	Philip Schmid
Internal Co-Examiner:	Prof. Dr. Olaf Zimmermann

Abstract

Introduction

Infrastructure providers, such as mobile providers, heavily rely on Kubernetes to orchestrate containerized applications. More modern and flexible cloud approaches have increased and outpaced the development of flexible infrastructure. Nowadays, additional clusters are often being deployed manually, which is error-prone and leads to non-standardized infrastructure. As running workloads on single clusters is manageable, having hundreds of clusters, each running different applications, depicts new challenges. Deploying and running remote clusters is not only a question of service orchestration, it brings up the need for high-performing, flexible, and private networking. Bringing this challenge further was key to our thesis. We evaluated Nephio as intent-based automation framework for service orchestration utilizing the concept of Single-Source-of-Truth using GitOps. Networks between the source and destination must be as flexible as the clusters to automatically ensure private and secure network routing along newly deployed clusters and workloads. Running a new cluster for a new customer intends to manually deploy Kubernetes, container network interface (CNI) plugin, workloads, and network connectivity. This thesis aims to prove possible automation of these challenges with state-of-the-art technologies like SRv6 packet-routing, Cilium networking for Kubernetes, and Nephio service orchestration platform.

Methods

Multiple IPv4 and IPv6 networks simulating a network infrastructure were set up. The routers were connecting multiple Kubernetes environments. In the first stage, the Kubernetes environments were deployed using a customized Ansible playbook based on Kubespray. The following deployment installed Cilium as a CNI plugin on the deployed Kubernetes control-plane node. Eventually, the third deployment integrated the Kubernetes environment into the central Nephio management cluster. Nephio allowed us to deploy not only applications but network configurations as well. A substantial effort was put into network device configuration deployment to ensure networking between Kubernetes containers. An additional optional use case was achieved by scaling Kubernetes clusters to the public cloud, which were connected to our on-premise environment.

Results

The Kubernetes deployment was executed successfully, enabling low-effort cluster deployments. Cilium networking was deployed, allowing new networking technologies, like source-determined routing called SRv6. Nephio, which is still in a pre-launch condition, implements straightforward and effective workload deployments for multi-cluster environments. As cloud-native infrastructure covers a wide range of topics, we were able to demonstrate the capabilities of further deployments like network devices or public cloud enrollment. Some parts of cloud environments are more challenging than others to deploy, and with new technologies coming up, cloud environment automation will remain a fascinating topic. Our thesis, conducted as a proof-of-concept, demonstrates that many manually configured infrastructures may be automatically deployed.

Discussion

Nephio is a new product with a lot of potential and numerous features on the roadmap. In the future, enterprises with large fleets of decentralized Kubernetes clusters should keep an eye on Nephio. Single-Source-of-Truth and GitOps are featured by Nephio and will be adapted in networking in a broader spectrum with promising potential. SRv6 is still an emerging and exciting technology, offering flexible routing between containers in extensive networks.

Management Summary

Initial Situation

In recent years the transformation from locally hosted to cloud-based applications has accelerated. Applications are running decentralized within containers in tiny snippets of workloads. Therefore, connections between these locations must be reliable, stable, and customizable. Data and business logic are primarily stored and run in distant data centers. A well-known example of the shift towards cloud-native technologies are mobile network providers. The evolution from 4G to 5G significantly changes the backbone networks and its core services. As the network logic is being distributed, the need arises for more standardized deployments. Having cloud infrastructure successfully deployed still leaves the need for running services on the new infrastructure efficiently. Managing multiple clusters, different applications and policies is hard done by hand, bringing service orchestration into play. Orchestration tools allow engineers to deploy and manage services to different locations easily. Running multiple locations with different applications for multiple customers results in ever-increasing network challenges. Different types of traffic with specific needs must be handled and kept as private as possible. This results in decoupling of networks and locations, enabling a movable and flexible cloud architecture. Eventually, cloud infrastructure providers must stick to the trend of easily deployable and flexible cloud infrastructure. This bachelor thesis aims to prove the possibility of automated deploying cloud infrastructure with combined networking and service orchestration solutions.

Procedures and Technology

At the start, we discussed the tasks with our supervisor and advisor. It was decided that in the first phase, we would compare different tools against each other to manage different clusters and applications. In the second stage, we created a multi-cluster network topology to run our deployments. This allowed us to test multiple deployments such as Kubernetes, Cilium, Nephio, and network device configuration. With public cloud providers increasing, spanning the local network into the public cloud was also tested. The group focussed on an optional use case, establishing SRv6 packet routing from container to container, allowing specific routing for applications such as VOIP, streaming, and low-priority messaging on a per customer basis.

Results

The bachelor's thesis was completed within time and according to restrictions. Along the way, the team members faced various challenges regarding the availability and maturity of products and tools. The non-optional use cases were developed, tested, and documented. Intent-based deployments successfully initialized infrastructure like Kubernetes, container network interfaces, and Nephio as a service orchestration tool. This proved that an efficient and reproducible way of deploying infrastructure is possible. Scaling workloads to the public cloud connected to the on-premise infrastructure and having SRv6 routing functional showing the vast potential of new technologies. The findings and challenges were reported throughout the thesis, and experiences were added to the documentation.

Outlook

This bachelor thesis, elaborated as a proof of concept, demonstrated the possibility of a streamlined cloud infrastructure deployment. With the present work done, the deployments must be configured within different configuration files. Bootstrapping the deployment, further lowering the barriers, and using stable releases is vital for a production environment. Managing various applications on numerous clusters is critical for efficient administration. Nephio, still under development but already highly skillful, should be kept an eye on in the future. SRv6 is a capable approach of customized IP packet routing, especially suitable for scalable and converging infrastructure. SRv6 will play an extensive role in the industry in the coming years.

Assignment of Project

In this chapter, the official ask assignment, with the sections "Introduction", "Goals of the project" and "documentation", is included.

Introduction

The infrastructure layer in a cloud native environment is usually configured statically or by a one on one basis. In order to ensure that the infrastructure is in sync, a third party tool could be used. Such a tool should not only help to keep the current infrastructure in a desired state, it should also help scale the infrastructure if needed. Infrastructure should in a first step be an on-premise cloud-native solution, in a further step a cloud-based cloud-native approach can be considered. The surrounding network should also be considered in both approaches. In this thesis the goal is to evaluate different approaches and solutions to this problem. A given solution should be verified and validated in a proof-of-concept.

Goals of the project

The scope of this project includes the following tasks:

- Research on nephio (https://nephio.org/) and fluxcd (https://fluxcd.io/)
- Compare given products in respect to automating cloud native infrastructure
- Create a proof-of-concept for one of the products or more.
- Test and validate the developed solutions on-premise
- Optionally: Test and validate the developed solutions in a cloud environment

Documentation

This project must be documented according to the guidelines of the "Informatik" department. This includes all analysis, design, implementation, project management, etc. sections. All docu- mentation is expected to be written in English. The project plan also contains the documentation tasks. All results must be complete in the final upload to the archive server. Documentation will only be handed-in in digital form, no prints are required.

Acknowledgments

We want to point out our thanks to Prof. Laurent Metzger for the supervisor role. He enabled us to continue our work on Cilium, Segment Routing version 6 (further referred to as SRv6), and networking from our Semesterarbeit (further referred to as SA) elaborated in the previous semester. The group appreciated the idea of this topic and his uncomplicated way of answering organizational questions in a straightforward and pleasant manner.

Special thanks to our advisor, Marco Martinez, from the Institute for Network and Security (further referred to as INS). He acted in this role during his exchange semester in Mexico and took place in meetings remotely. Marco Martinez provided the team assistance on technical challenges and gave us feedback for improvements at any time.

Many thanks to Prof. Dr. Olaf Zimmermann from the Institute for Software at OST, who gave us valuable feedback, along with other things to remember, during the mid-term presentation. Having many years of experience supervising bachelor's theses, he reminded us to keep relevant things in mind regarding project management with quality attributes and to point out the project value to the industry.

We want to acknowledge Philip Schmid, Solutions Architect at Isovalent. He provided us with an early-access version of the SRv6 feature in Cilium and technical assistance on the setup. In addition, he gave us valuable input from his expertise in the industry of cloud-native infrastructure.

Contents

Part I.

Technical Report

CHAPTER 1

Introduction

1.1. Problem Scenario

Today, many enterprises use the beneficial features of Kubernetes. Its ability to deploy, manage and scale containerized applications abstracts the complexities of managing containers. However, these powerful functionalities come with a cost. The cost of a complex and time-consuming installation process for each cluster.

With the growing demand for deploying network functions on Kubernetes clusters, such as for the 5G broadband cellular network coverage, thousands of new clusters must be set up. These clusters connect thousands of 5G antennas across the globe, with the backbone of each service provider.

Several steps are necessary to deploy and manage a Kubernetes environment. An underlying virtual machine has to be deployed to run Kubernetes on it. Afterward, multiple package installations are required to set up a Kubernetes node. The node must join a controller to become part of a cluster. A network plugin is required to enable network communication on pods when a cluster is built. One can then deploy applications on it, but with an increasing amount of workloads, the complexity of managing all those workloads also increases.

These steps need a form of automation and a central management point, as well as a way to deploy multiple clusters worldwide automatically, configure network settings, and deploy workloads on the clusters. Those are the challenges we want to solve in this thesis.

1.2. Proof of Concept

A handful of established and new technologies might be the answer to provide central and scalable mechanisms for provisioning Kubernetes clusters and application workloads distributed over long distances. This thesis is a Proof Of Concept (further referred to as PoC) where we want to combine a set of tools to prove that it is possible to provide a reliable and scalable platform for Kubernetes cluster provisioning and package deployment by realizing a set of use cases that capture the steps mentioned above. The goal is to deploy a production-ready environment with the network functions to enable communication between edge clusters. The following sections describe the use cases addressing these challenges.

1.3. Target Audience

This document is intended to serve as a valuable resource and reference for individuals involved in managing, operating, and optimizing Kubernetes cluster and workload deployments, such as Internet Service Provider (further referred to as ISP) and operators of large fleet clusters.

- Internet Service Providers (ISP): This guide is ideal for individuals working in ISPs of varying sizes and complexities, including local, regional, and global providers. Whether you are responsible for infrastructure management or service deployment in a cloud-native environment within an ISP, this guide will offer valuable insights and practical recommendations. In recent years, the shift to all-new standalone 5G applications has driven the need for network function orchestration.
- Operators of large fleet clusters: For professionals involved in managing large Kubernetes fleet clusters, such as cloud service providers, data centers, or distributed computing environments, this guide is tailored to address the unique challenges and requirements of managing Kubernetes clusters and the workloads on it. It provides relevant information on network architecture, scalability, and provisioning.
- Network Administrators and Engineers: Network administrators and engineers responsible for maintaining and optimizing ISP networks or large fleet clusters will find this guide helpful. It covers topics related to network design and routing protocols specific to ISP and large fleet cluster contexts.

1.4. Definition of 'Cloud Native Intent Automation'

The title 'Cloud Native Intent Automation' was adapted from the advisors' task assignment. We will briefly explain what we mean by this, as the title was chosen generically.

1.4.1. Cloud-Native

With cloud-native, we refer to a design philosophy and set of practices aimed at developing and deploying applications that leverage the capabilities of cloud computing platforms. Cloud-native applications are typically built using modular and loosely coupled components that can scale dynamically and take advantage of cloud-native services like containers.

1.4.2. Intent-based Automation

As Intent-based automation, we understand the approach to automate operations based on desired outcomes or intents rather than specific instructions or scripts. As in Kubernetes, where a desired state is defined, the system dynamically adapts and responds to changes in the environment. Intent-based automation focuses more on "what should be" than "how do we get there" of operations.

1.4.3. Cloud-Native Intent-based Automation

Combining these two concepts, 'Cloud-Native Intent-based Automation' refers to applying intent-based automation principles within cloud-native architectures. It involves automating cloud-native application deployment, management, and scaling based on the desired outcome.

1.5. Evaluation of Nephio and FluxCD

Most Kubernetes operators use a continuous delivery tool for managing Kubernetes Resource Models (further referred to as KRM) on Kubernetes environments instead of applying all manifests imperatively on the CLI. These tools use a Git source to store KRMs enabling version control. In contrast, a sync container on the Kubernetes checks for changes in the repository and automatically applies changes to the existing resources. FluxCD is a popular continuous delivery solution accepted as a graduated project by the Cloud Native Computing Foundation (further referred to as CNCF). It works with any Kubernetes and all standard tooling, including support for Kustomize, Helm, GitLab, GitHub, and custom hooks. It works with multiple clusters and can integrate notification channels to be informed about changes and errors.

On the other hand, there is the new ongoing Linux Foundation project Nephio, which has many telecommunication providers and enterprises as drivers. It is a Kubernetes-based intent-driven automation of network functions and the underlying infrastructure that supports those functions. Nephio aims to provision and manage multi-vendor, multi-site deployments across multiple clusters. It can deploy and manage KRMs similarly to FluxCD but is focused on deploying network functions rather than applications. Network functions are software-based functions or services that provide specific functionality within a computer network.

As a first step in the thesis, both tools are evaluated and compared to identify the appropriate practices. Both are installed and tested by provisioning container deployments. For the rest of the thesis, the team will use Nephio to implement a series of use cases as it is a newer and promising technology despite only being available for a PoC version. The PoC version already supports the GitOps mechanism similar to FluxCD.

1.6. Provisioning of Kubernetes and Nephio

Telecommunication providers and enterprises could manage a large fleet of Kubernetes clusters distributed over large distances to meet customers' requirements by having a close infrastructure to avoid high latencies in their applications. Keeping an overview of all clusters and maintaining configurations is challenging. There is a need for a centralized and flexible approach. A solution to deploy production-ready Kubernetes clusters and centrally manage applications and configurations workloads would be very beneficial. When using managed Kubernetes services from cloud providers, the setup of a node is simple and can be rapidly scaled horizontally. One needs access to the underlying virtualization infrastructure for on-premise infrastructure. Kubespray is a well-known community tool to automate deployments. It is a bootstrap of Ansible scripts to deploy Kubernetes on existing remote machines without having an agent installed.

In this thesis, Kubernetes is deployed on existing remote machines in the Lab Topology Builder (further referred to as LTB) using the Ansible based Kubespray. In addition to provisioning a complete Kubernetes cluster, Nephio will also be deployed using bare Ansible scripts. It should be ready for centralized package deployment after Nephio's deployment.

The entire fleet configuration to deploy Kubernetes with Kubespray and Nephio is stored and applied on GitLab, which provides a pipeline to trigger changes made. An Ansible Docker container in the pipeline is used to run tasks declared in YAML on remote machines. This approach would make GitLab the single source of truth, having all configurations of the entire fleet deployment. As Nephio in its current form only supports GitHub and Google Cloud Repository Services to store deployment packages, we will use GitHub to store Nephio packages.

1.7. Provisioning of Cilium CNI

A Kubernetes node is only ready to run pods with a Container Network Interface plugin (further referred to as CNI plugin). The CNI plugin is responsible for the pods' network communication and allows them to communicate with destinations inside and outside the cluster. There is a large variety of CNI plugins, which differ in technologies and policies they support. Operators must choose the best CNI plugin that meets requirements and offers appropriate technologies.

The second use case is deploying Cilium CNI plugin to all Kubernetes nodes with the proper set of parameters that are necessary to support the routing technologies Border Gateway Protocol (further referred to as BGP) and SRv6 used in the following use case to enable Layer 3 Virtual Private Network (further referred to as L3VPN) between Kubernetes clusters. The provisioning also follows the GitOps approach and is integrated into Nephio or alternatively into the Ansible scripts of use case 1.

1.8. Deployment of Cilium BGP and SRv6 Workloads for L3VPN

With a large fleet of Kubernetes clusters, network communication between pods of different clusters is necessary to reach specified requirements. The common practice is to work with port forwarding and static routes, making the entire setup more complex and prone to errors. SRv6 is a trending technology that can segment the routing of networks that are isolated from each other. Multiple customer prefixes can be routed through the same network infrastructure, with customers only having available routes to their networks. This method is called L3VPN and allows service providers to offer IP-based Virtual Private Network (further referred to as VPN) services to their customers or different sites. L3VPN enables enterprises to securely connect their remote sites to their central network by connecting them to a backbone network. SRv6 is currently being developed for Cilium CNI plugin in order to segment pod prefixes, allowing communication between pods that are in the same Virtual Routing and Forwarding (further referred to as VRF). Cilium supports BGP, the routing protocol used to exchange the corresponding SRv6 Segment ID (further referred to as SID) for every network prefix. With a network infrastructure supporting SRv6, it is possible to route traffic between pods of two independent clusters as long as they belong to the same VRF. Figure 1.1 shows how the traffic of two pods on the same cluster is routed differently through the network infrastructure. With this approach, one can not only provide connectivity between clusters but also configure predefined paths with SRv6 to use the optimal path for traffic (high bandwidth, low latency), also called Traffic Engineering (TE).

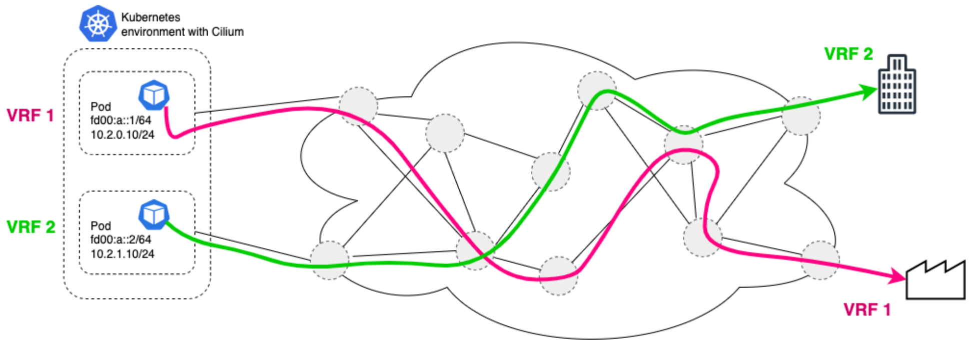


Figure 1.1.: Abstraction of Segment Routing in Kubernetes

Nephio was initially designed to deploy network functions for 5G networks across multiple clusters. In this thesis, the network functions are the Cilium BGP and SRv6 configurations in YAML that are managed and deployed with Nephio. Use Case 3 consists of the deployment of BGP and SRv6 workloads that are validated by end-to-end connectivity between two pods on different Kubernetes clusters.

1.9. Extension of Kubernetes Infrastructure to the Cloud for Scalability

Being able to deploy Kubernetes clusters in the cloud dramatically enhances the scalability of on-premise infrastructures. Therefore the on-premise infrastructure in LTB will be extended with 'Compute Engine' deployments in Google Cloud Platform (further referred to as GCP). The Virtual Machine (further referred to as VM) will be manually started and configured with the Ansible deployment from the previous use case. The BGP and SRv6 workloads should be adapted to enable L3VPN between the pods in GCP and the ones in LTB.

A pod running on the Google Cloud Compute Engine should be able to communicate with a pod running on a VM in LTB.

1.10. Structure of Document

In this part, we list the chapters of this thesis and briefly explain each.

1.10.1. Theory

After introducing this thesis, the technologies and their concepts are explained in chapter 2 Theory. The detailed analysis, evaluation, and comparison of FluxCD and Nephio are also described.

1.10.2. Methods

The chapter 3 Methods consists of an overview of the infrastructure in LTB used in this project. The central part of this thesis is the implementation of all use cases that are also represented in this chapter.

1.10.3. Results

The second chapter described the implementation of all use cases. Afterward, the actual results and the validation of the PoC are represented in chapter 4 Results.

1.10.4. Discussion

In chapter 5 Discussion, we retake the initial concept we aimed to prove and mention our key findings. We also explain our challenges, limitations, and recommendations regarding this bachelor thesis.

1.10.5. Challenges and Decisions

We explain our technical challenges and decisions during the bachelor thesis in more detail in chapter 6 Challenges and Decisions. We explain how, when, and why we decided to make specific changes and use certain technologies and how we responded to technical challenges.

1.10.6. Notable Points

This part contains some topics that did not fit the Chapter Challenges and Decisions but were also crucial for our thesis. It contains technical information and things we tried.

1.10.7. Project Documentation

This thesis contains a part regarding the project documentation not available in the official version. It is only available in the internal version.

Reflection

A reflection from the team's perspective was written in chapter Reflection in the project documentation. It only contains parts deemed necessary for future projects.

Personal Retrospective

The chapter Personal Retrospective in the project documentation contains each team member's personal review of the project.

1.10.8. Appendix

The part Appendix contains all references in the bibliography, a glossary, a list of abbreviations, and all configuration listings.

1.11. Project Scope

The INS provided us with a founding network infrastructure. It contained the following:

- Routers with interfaces connected
- Virtual machines with interfaces connected to the edge routers and internet access to install Kubernetes and Cilium.

We configured all routers and virtual machines. The following will not be part of this thesis:

- No evaluation in regards to comparing our setup to others.

CHAPTER 2

Theory

2.1. General

In the theory chapter, we first address how cloud infrastructure has been deployed throughout time and what new challenges exist. We explain the concepts of various tools used in this thesis and how they are connected.

2.2. History of Cloud Infrastructure Deployment

A decade ago, containers were unpopular, and it was common practice to deploy an on-premise virtualized infrastructure running applications on virtual machines. These virtual machines were deployed by a hypervisor on multiple physical servers. In 2012 the release of Ansible, a scripting tool for automation, made deployments easier, and it was not necessary anymore to connect to every machine manually. It uses Secure Shell Protocol (further referred to as SSH) to connect to destination hosts and apply the defined tasks. One can run the same tasks for hundreds of hosts, define variables for specific hosts and groups, reuse tasks, read logs, and many other automated processes. The community and vendors of products would create modules to automate specific tasks without the users needing to create a script for every task on their own.

At the same time, the usage of public cloud services was rapidly growing in enterprises. Cloud services offer significant cost savings compared to on-premise infrastructure, as only the used resources are billed. It also saves the costs and time of investing in and maintaining its hardware and infrastructure, which leads to more flexibility and scalability as there are no hardware limits for the user. Managing infrastructure in a cloud environment was often a complex and error-prone process involving a lot of manual configuration and scripting. There were many providers, each with a different Application Programming Interface (further referred to as API) for its services. In 2014, Terraform was invented by HashiCorp to simplify the process of managing and provisioning infrastructure in a cloud environment and unified various providers' different APIs and tools into a language called HashiCorp Language (HCL). With Terraform, infrastructure can be defined in a declarative language, which is easier to understand and modify than complex scripts. Terraform also provides a consistent workflow for provisioning and managing infrastructures across different cloud providers and environments, making it easier to scale and manage infrastructure as it grows.

In the late 2010s, containerization became very popular with the rise of Docker. Every application had to run on a different virtual machine to be isolated from each other. Deploying applications on those virtual machines was complex and time-consuming, and the environments strongly depended on the operating system. Docker introduced a standardized format for container images, which made it easier to package applications and their dependencies into a single container that could run consistently across different environments. Containers are lighter than virtual machines as they share the host operating system's kernel and do not require a separate guest operating system. They can easily be scaled horizontally, allowing multiple instances for failover scenarios. Containers have great portability and can be easily moved between infrastructures.

Managing multiple containers needed an orchestration platform. By 2017, Kubernetes had become the dominant container orchestration platform and was widely adopted by organizations of all sizes. Today, Kubernetes is the most popular container orchestration platform with a large community of developers and operators. It offers high availability and scalability of applications by running multiple Kubernetes nodes that form a Kubernetes cluster. A cluster consists of controllers and worker nodes. A controller node is the control plane of the Kubernetes cluster and manages the cluster's state, performing pod scheduling, scaling, and monitoring. The worker nodes run the containerized applications on pods. A pod consists of one or multiple containers that operate together. Each worker node has an agent called Kubelet that receives instructions to schedule pods from the controller node. Each node has a container runtime installed to run containers and a CNI plugin to enable communication between pods on different cluster nodes and provide support for network policies. Kubernetes is installed on virtual machines to benefit from the flexibility of moving the host system between physical servers.

A Kubernetes environment is usually deployed with tools like Kubespray, which is a collection of Ansible install scripts or software like Rancher that runs on Kubernetes. Rancher can deploy nodes on the public cloud or even in an on-premise cloud with the underlying virtual machine. More tools and applications are moving to the Kubernetes ecosystem as it benefits application development and delivery. Kubernetes is everywhere, and it has even begun to be used for cloud infrastructure deployments. AWS Controllers for Kubernetes (further referred to as ACK) is an example of a new service that allows for the creation of cloud resources by creating Kubernetes objects.

2.3. Challenges with Kubernetes Clusters in 5G Networks

With the increasing coverage of 5G networks, a larger and more distributed infrastructure is needed to manage a wide area of radio units. Telecommunication providers are switching from old legacy systems to modern containerized applications, which are more efficient and simplify application development and delivery. This need for distributed infrastructure is also the case with 5G network functions currently used for the control planes and services of 5G. A network function is a software application that performs a network task. Some tasks could be routing, content filtering, authentication, and authorization. A 5G network consists of four layers[1], and spacious countries could have 100'000 remote edges connected to radio units to cover an appropriate area. Figure 2.1 shows the current 5G coverage in Canada with the cyan-colored area. One can see that it is only a small coverage.

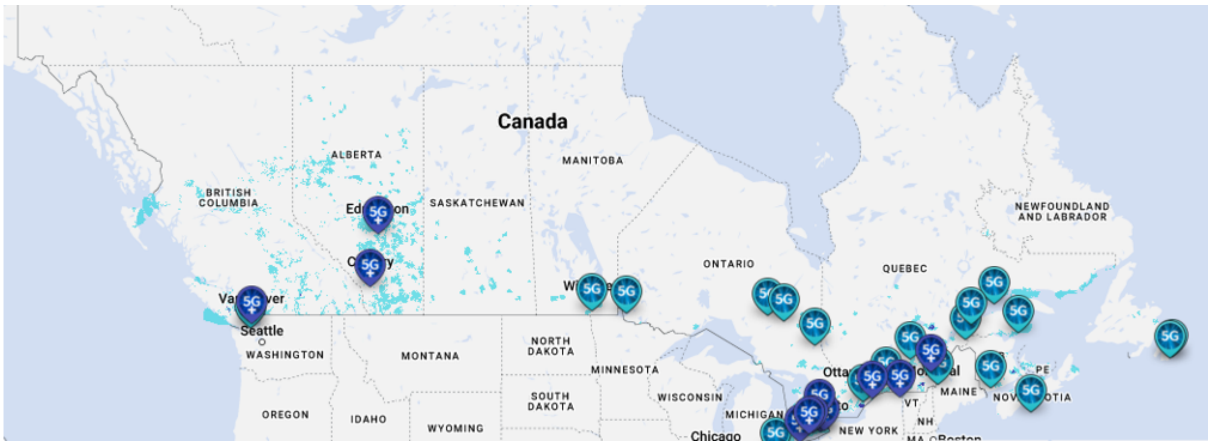


Figure 2.1.: 5G Coverage in Canada by Bell Canada [Bel23]

[1]Nephio Intro and Pre-Release Demo [BW22]

With the expansion of 5G in the future, an infrastructure with a large amount of Kubernetes clusters is needed to manage network functions.
Mobile Edge Computing (MEC) is a computing platform that enables applications to run closer to remote sites. It is a distributed computing environment that reduces latency by reducing the distance to end-users. In the case of 5G, a Kubernetes cluster is a MEC. Depending on the specific use case and requirements, Kubernetes clusters can be deployed as MEC in every layer.

Telco Remote Edge

A Telco Remote Edge contains one or multiple Radio Units (RU) providing wireless connectivity to devices and processing radio signals. Multiple Radio Units are connected to one Virtualized Distributed Unit (vDU). A vDU processes the radio signals received from the RU and forwards them to the next layer. The vDU is deployed to containers in a Kubernetes cluster close to the RUs.

Telco Pre-agg

Telco Remote Edges forward traffic to the second layer, the Telco Pre-aggregation. It contains a Centralized Unit - Control Plane (CU-CP), which manages radio access network (RAN) functions, such as radio resource management, mobility management, and session management. It helps reduce loads on the vDUs and RUs by providing a centralized control plane. The other component is the Centralized Unit - User Plane (CU-UP), which handles user traffic and forwards it to the Telco Aggregate layer. CU-CP and CU-UP network functions can be deployed to Kubernetes. A Telco Pre-aggregation is typically at a central Telco Remote Edge in a larger residential or industrial area.

Telco Agg

The third layer, the Telco Aggregation, aggregates traffic from multiple Telco-Pre Aggregates and forwards them to the public cloud. A Telco Aggregate is typically located in a regional data center and is a gateway between the RAN and the core network. The network functions are routing, load balancing, and security. These network functions run on Kubernetes as well.

Public Cloud Region

Last, Telco Aggregates are connected to the public cloud region where the Access and Mobility Function (AMF) and the Session Management Function (SMF) are located. The AMF performs tasks such as device authentication, authorization, security, and mobility management, including device handover between different network segments. The SMF is responsible for session establishment, manages the user's Quality of Service (QoS) settings, and performs policy enforcement.

Understanding the components of a 5G network is not essential for this thesis. However, one can see that many clusters are needed to provide such an extensive infrastructure. What about all the network functions needed on the different layers? The management of those workloads is complex in such a large infrastructure. The current approach uses a tool like FluxCD to deploy workloads over GIT. The current tools have yet to be optimized for deploying many clusters and consist of many manual steps. Tools like Terraform or Rancher are used to deploy the Kubernetes clusters in the public cloud or to an on-premise infrastructure. Terraform and Rancher are appropriate for simple cluster deployments but struggle when complex configurations are needed. Nephio could improve these concepts and decrease the complexity of deploying infrastructures and workloads.

2.4. Kubespray

Kubernetes has become increasingly popular in recent years. Since the shift from legacy applications to cloud-native workloads has increased, various ways of installing Kubernetes have been described. As cloud-native applications have become widely adopted, tools on top of Kubernetes for further customization have emerged one by another. Manual deployments of multiple Kubernetes clusters are error-prone and becoming more complex to more tools and plugins such as CNI plugins. Beginning in 2016, Kubespray[2] was developed as a solution. The Ansible backed solutions aim to streamline the deployment of Kubernetes clusters. Deploying Kubernetes using Ansible playbooks allows a simple reproducible installation on bare-metal, VM, and cloud environments. Ansible is based on a declarative approach, meaning users define the desired state of their environment in configuration files or within the command line. Single deployment tasks are written in YAML playbooks, split into roles and different Kubernetes tools. These roles include tasks, such as control plane or worker node jobs, executed according to the desired state on a per-host basis. The environment is specified, as it is with most Ansible based deployments, within a .ini file. After being specified, the hosts must be added to groups according to their role (control plane, etcd node, worker). Kubespray offers excellent flexibility not only with the core deployments but with CNI plugins as well. The CNI plugin can be specified during the initial deployment configuration. Kubespray supports different CNI plugins such as Flannel, Cilium, or Calico, no CNI plugin may be installed for later installation as well. Not only does Kubespray deploy new Kubernetes clusters, it offers more functionalities like scaling out existing clusters, scaling down, or changing/removing control plane nodes. The specified nodes are being checked for previous Kubernetes instances (via Ansible fact-gathering) to determine what changes to which host must be deployed. Dependent on Kubespray being a set of tools and based on Ansible, strict software version requirements must be met. Playbooks may be self-edited and customized, extended, and optimized. This allows integrating Kubernetes-deployments in existing Ansible deployments and administrational tasks.

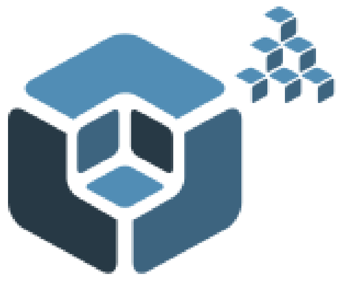


Figure 2.2.: Kubespray Logo [kub23b]

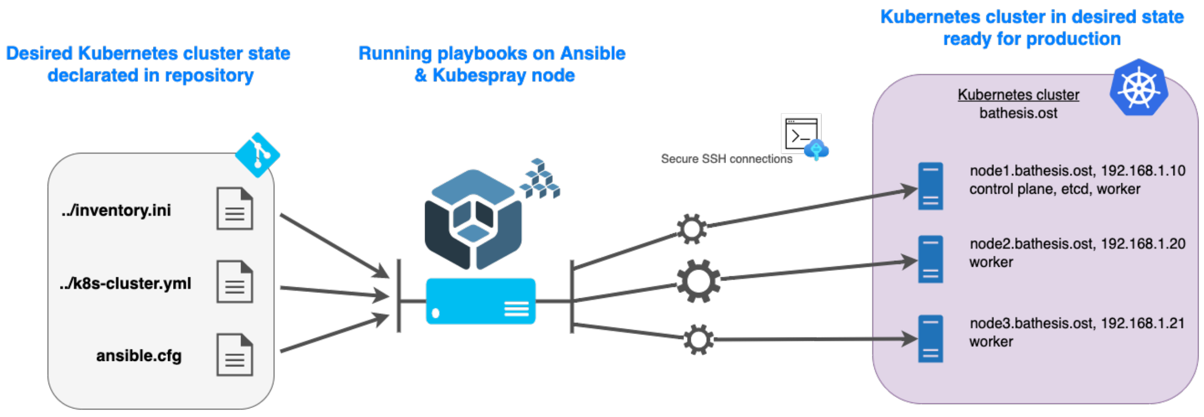

Desired Kubernetes cluster state declarated in repository
Running playbooks on Ansible & Kubespray node
Kubernetes cluster in desired state ready for production
../inventory.ini
../k8s-cluster.yml
ansible.cfg
Secure SSH connections
Kubernetes cluster
bathesis.ost
node1.bathesis.ost, 192.168.1.10
control plane, etcd, worker
node2.bathesis.ost, 192.168.1.20
worker
node3.bathesis.ost, 192.168.1.21
worker

Figure 2.3.: Kubespray Workflow Schema

[2]Link to the official Kubespray website [SIG23]

2.5. GitOps

In order to understand either Nephio's or Flux's application field, one has to understand the usage of GitOps. Many Kubernetes operators manage their resources using Command Line Interface (further referred to as CLI) tools like Kubectl, graphical user interfaces, or declarative configuration tools such as Terraform and Helm, which directly control the Kubernetes API. On a small scale which tool is used is mainly driven by preference and familiarity. As companies become more extensive and expand the number of Kubernetes deployments and clusters, creating and enforcing consistent configuration and security files becomes difficult. At this point, management tools are no longer chosen by preference but by capabilities. To overcome this challenge, platform engineers increasingly use the GitOps approach, which enables consistent and version-controlled configuration deployment across multiple clusters and environments.[3]

Simply put, GitOps is used to automate the process of deploying infrastructure. Like how software teams store source code in GIT repositories, operation teams that adopt GitOps, store infrastructure configuration files as code, mostly YAML, in GIT. Moreover, these configuration files deploy infrastructure to various destinations like Kubernetes[4].
GitOps requires the core components shown in the following figure.

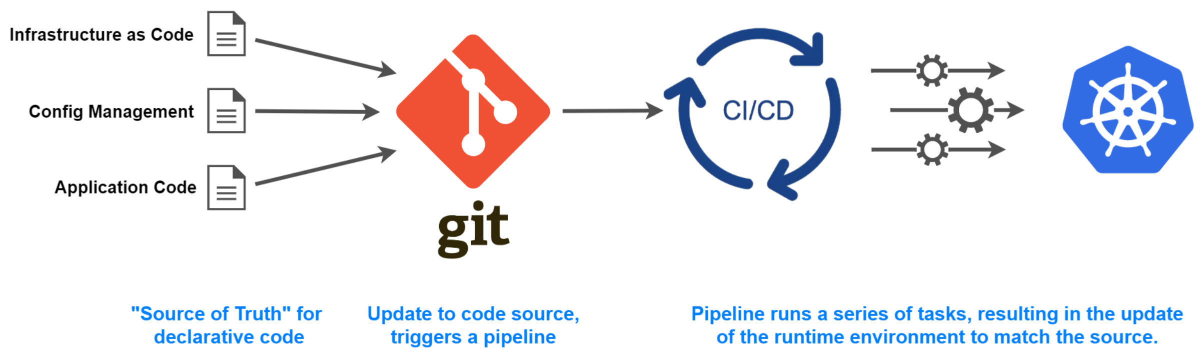

Infrastructure as Code
Config Management
Application Code
git
CI/CD
"Source of Truth" for declarative code
Update to code source, triggers a pipeline
Pipeline runs a series of tasks, resulting in the update of the runtime environment to match the source.

Figure 2.4.: GitOps Overview

In our case, we used GIT to store our Infrastructure as Code (further referred to as IaC) and establish a single source of truth. Continuous delivery tools like Nephio and FluxCD track changes in a specific GIT repository and automatically deploy changes to Kubernetes clusters. These tools ensure that the actual state in the application, called 'actual state', matches the state declared in the configuration state, called 'desired state'.[5]

[3]Why GitOps? [kpt23]
[4]Gitlab: What is GitOps? [git23]
[5]What is GitOps? [Car22]

2.6. Nephio

On April 12, 2022, the Linux Foundation, Google Cloud, and a dozen leaders across the telecommunications industry, including Bell Canada, officially launched the Nephio project. Building, deploying, and managing large scalable 5G networks across multiple edge locations is complex. The telecommunications industry needs cloud-native automation to be simpler, easier, and faster while achieving optimization and agility in cloud-based deployments. The Linux Foundation and Google founded Nephio to address these hurdles.[6]

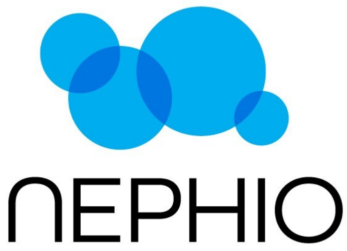


Figure 2.5.: Nephio Icon [Nep23c]

The most significant bridge Nephio aims to cross is providing a "carrier-grade" option for telecommunication operators to automatically deploy their 5G networks across multiple edge locations in a cloud-native approach. This platform will allow for faster network function onboarding across cloud environments, miming the DevOps process in the software industry. Historically, this "carrier-grade" option has been a concern, as telecom operators work in a highly regulated space that requires government and public safety regulations, which mostly surpass those for traditional cloud operators.[7]

Nephio uses Kubernetes as the control plane to configure all aspects of the distributed cloud or network functions. However, it does not provide cloud infrastructure or network functions directly, enabling users to deploy them automatically.[8]

The Nephio automation framework is built on the Google Open Source projects Config Sync and Kubernetes Package Tool (further referred to as KPT). They implement the What You See Is What You Get (further referred to as WYSIWYG) approach.[9] A classic example of a WYSIWYG tool would be Microsoft Word. The document on the screen looks the same as on another device, like a printer.[10] KPT adds functionalities to the already declarative approach of Kubernetes.

2.6.1. Nephio Environment

Figure 2.6 shows a high-level diagram of Nephio, with the blue area being the Nephio environment on a cluster. One can interact with Nephio through the KPT CLI or the Nephio UI. Another component is the controller set that lets Nephio manage infrastructure deployment. The controllers[11] integrate APIs of various public cloud providers such as ACK (AWS), Azure Service Operator, and Google Config Connector to manage cloud resources with IaC using Cloud resource definitions as Kubernetes objects. The Porch (Package Orchestration) API is responsible for the package lifecycle. It manages the connected package repositories, discovers configuration packages and KPT functions, and supports version control. Porch is the component that deploys and manages packages from a catalog repository to edge Kubernetes clusters.

[6]The Linux Foundation: Launch of Nephio [Fou22]
[7]sdxcentral: Launch of Nephio [May22]
[8]Linux Foundation launches Nephio to automate 5G edge sites [Har22]
[9]About Nephio [Nep23a]
[10]WYSIWYG [Com18]
[11]About Nephio [Nep23a]

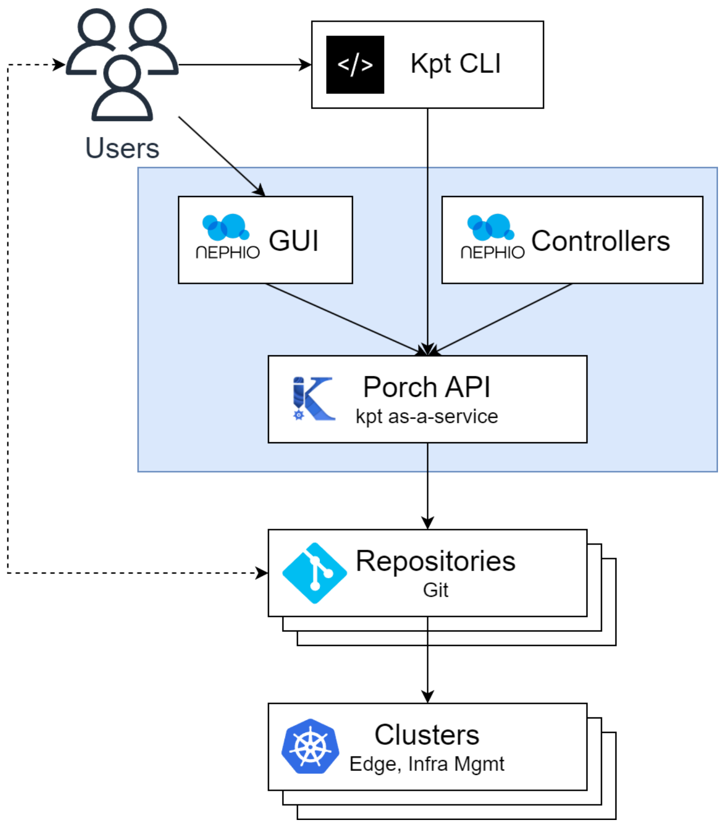


Figure 2.6.: Nephio High Level Diagram [BW22]

2.6.2. Nephio Components

KPT

Nephio is based on the packing tool KPT, an open-source tool by Google used to deliver bundled KRMs. It is similar to Helm and can also use Helm charts in a KPT package. The difference between KPT and Helm is that Helm is mainly used for application deployment, while KPT focuses on package deployments with many configuration parameters in a package. As an example, Helm can be used to deliver a software application by choosing the application Helm chart that bundles web application, API server, and database and then specifying simple parameters like customer name, version, amount of instances, and storage location for different instances. KPT, on the other hand, is used for complex configurations where parameters have to meet requirements. An example of a deployment with many configuration parameters is a network function that uses network configurations as parameters when creating a release from a package.

KPT follows the concept of "Configuration as Data"[12] where configuration parameters are treated as data and stored in a structured format rather than just being embedded directly within the code. This concept simplifies maintenance tasks, enables version control systems to better track and audit configurations, makes it easier to adapt to different environments, and allows the management and manipulation of configuration data using automation and tooling. KPT implements the latter using functions[13] as containerized programs to validate or change KRM definitions in package deployments. These functions can be programmed in Golang or JavaScript and are executed during deployments. A validator checks the entire package release to see if the configuration meets the requirements or business rules. A mutator goes over the entire package and changes values based on the intent. For example, we could use a mutator to automatically set labels in multiple lines in a large and complex package instead of manually inserting them. A validator could validate the resources against their OpenAPI schema.

[12]KPT Concepts [KPT23d]
[13]KPT Declarative Function Execution [KPT23f]

KPT comes with its own CLI tool that manages packages on the client side. It can pull packages from repositories, manipulate them, and deploy them to a Kubernetes cluster.

Porch API

Porch stands for Package Orchestration service and is an API to manage the lifecycle of packages supported by KPT. It registers package repositories and accesses and manages them. Porch is also responsible for the discovery of configuration packages and KPT functions and performs Create Read Update Delete (further referred to as CRUD) operations on packages triggered from the KPT CLI or UI. It creates revisions of package changes by using the GIT mechanism of the repositories enabling a history of changes. The branching mechanism of GIT creates deployment drafts that need approvals before application. Once a package is approved, Porch applies the changes to the original package and deploys them to a destination repository.

Config Sync

Another component in the Nephio environment is Config Sync, another open-source tool by Google running as a pod on a Kubernetes cluster that subscribes to a deployment repository and performs package ingestion. When Porch pushes a modified package to the deployment repository, the Config Sync pod will see the package, fetch it, and create the Kubernetes resources on the cluster. Newly approved revisions will be immediately patched by Config Sync.

Nephio Controllers

Nephio includes a set of controllers interacting with the Porch API to deploy packages. The central Nephio controller uses and extends KPT with new intent automation features for 5G network functions. Nephio uses existing controllers to create cloud resources through Kubernetes manifests, which are used to create Kubernetes clusters over the Nephio control plane. The cloud controllers include ACK, Azure Service Operator, and Google Config Connector and are integrated into the Nephio ecosystem. The cloud resources are handled like configuration packages and deployed to an infrastructure repository which an infrastructure controller then fetches with Config Sync to create the Kubernetes clusters and integrate them to Nephio by registering a repository and subscribing to it with its own Config Sync instance.

2.6.3. Nephio Workflow

Sandeep Sharma[14] from Aarna Networks explained an example workflow of Nephio in a technical overview presentation. It shows a scenario where an operator creates an infrastructure in the cloud with Nephio and deploys some workloads once it is ready.

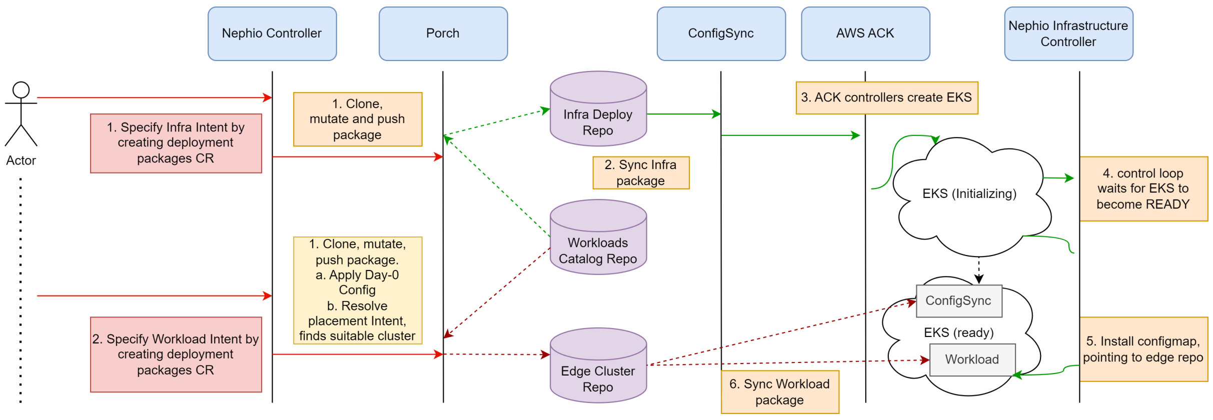

Nephio Controller
Porch
ConfigSync
AWS ACK
Nephio Infrastructure Controller
Actor
1. Specify Infra Intent by creating deployment packages CR
1. Clone, mutate and push package
Infra Deploy Repo
3. ACK controllers create EKS
2. Sync Infra package
EKS (Initializing)
4. control loop waits for EKS to become READY
Workloads Catalog Repo
1. Clone, mutate, push package. a. Apply Day-0 Config b. Resolve placement Intent, finds suitable cluster
ConfigSync
EKS (ready)
Workload
2. Specify Workload Intent by creating deployment packages CR
Edge Cluster Repo
6. Sync Workload package
5. Install configmap, pointing to edge repo

Figure 2.7.: Nephio Workflow [Sha22]

It begins by specifying an infrastructure intent on the Nephio UI or the CLI with KPT. The user chooses an infrastructure package from the blueprint catalog, modifies the custom resources, and enables the deployment.

[14]Nephio Technical Overview [Sha22]

The Porch server on the management cluster pushes the modified package to the infrastructure repository, where the deployment packages of the already existing fleet lie.
As the second step, the associated cluster for the infrastructure repository, in this case, is the management cluster, which uses Config Sync to check the repository for updates. Once it sees the new package, Config Sync fetches the package and uses the corresponding cloud controller as the third step to create the infrastructure resources by creating KRMs on the management cluster. The example shows the AWS ACK controller creating an Elastic Kubernetes Service cluster.
In step 4, the Nephio infrastructure controller waits until the ACK controller creates a ready remote edge cluster using a control loop.
In step 5, the Nephio infrastructure controller creates a deployment repository for the newly created cluster. The infrastructure controller also installs Config Sync on the cluster and creates a ConfigMap that points to the deployment repository. The new Kubernetes remote edge cluster is ready to receive workloads from the management cluster.
The user specifies a workload intent in the same manner as the infrastructure intent. It chooses a package from the blueprint repository, modifies the custom resources based on the requirements, and chooses the newly created remote edge cluster as the destination cluster. The user proposes and approves the deployment, which triggers the Porch API to push the package to the deployment repository of the remote edge cluster. The Config Sync pod on the remote edge cluster fetches the workloads from the package in step 6 and executes them locally. Nephio enables the provisioning of a Kubernetes cluster and the workloads on it with little effort.

It is important to note that the first use cases in this thesis are realized on an on-premise infrastructure. Nephio does not support infrastructure intents for on-premise environments. Thus, Ansible is used to deploy the Kubernetes clusters on existing virtual machines, to install Nephio on it, and map it with a repository.

2.6.4. Managing a large Kubernetes Fleet with Nephio

Nephio is being initially developed to overcome the challenges in deploying complex 5G network functions across multiple distributed Kubernetes clusters. Its features to spin up Kubernetes clusters at the edge and deploy packages from the same control plane make it also interesting for everyday applications outside the 5G spectrum. The full version of Nephio lets operators define comprehensive intents for cluster management and package deployments with a few interactions.
A developer team could create KPT packages of the application and deploy them to a private application catalog repository. The operation team uses the packages as blueprints, configures the configuration parameters, optionally executes validators and mutators, and deploys them to a large set of clusters.

However, only a PoC version of Nephio is available, which only supports the deployment of packages to remote clusters in a limited way and does not offer the infrastructure feature. The full version was supposed to be released in May 2023 but has been delayed to the third quarter of 2023. In this thesis, we use Nephio to deploy routing configurations and simple pods acting as applications. Kubernetes clusters are deployed to an on-premises infrastructure with the tool Kubespray that we extend to integrate the clusters into the Nephio environment. The chapter 3 Methods describes the usage of Nephio.

2.7. Flux

Figure 2.8.: FluxCD Icon [Com23c]

FluxCD is a GitOps operator for Kubernetes, which synchronizes the state of manifests in a GIT repository to what is running in a cluster. On July 15, 2019, FluxCD was accepted to CNCF and reached the Graduated project maturity level on November 2022, and it is a well-known tool in the GitOps space.[15] [16] Its approach is relatively simple, running directly in the cluster to which updates would be applied. FluxCD was initially developed by Weaveworks and is now a CNCF project licensed under Apache 2.0 on GitHub.[17] It officially supports installations with Helm releases and Kustomizations. With a primary focus on deploying the software delivery cycle, FluxCD is easy to install and maintain, like deploying any other typical pod in a cluster. It can watch a single GIT repository per installation, and it will be able to apply changes to the namespace in which the underlying service account has permission to make changes. There is also a 'fluxctl' CLI tool to help with the installation and interaction with the FluxCD daemon in the cluster to perform management tasks.[18] FluxCD can use different controllers to deploy Kustomizations and Helm releases natively.

2.7.1. FluxCD Components

The current major release version is Flux v2[19] consisting of a set of APIs and controllers interacting with various artifact sources. The Source Controller monitors various artifacts sources on changes and forwards information to two other controllers. There are multiple sources supported: GIT repositories, Helm Charts and repositories, Amazon S3 Buckets, and Oracle Cloud Infrastructure (OCI) repositories. In addition to monitoring changes and artifacts acquisition, the Source Controller also handles source validation, authentication to sources, update policies, and notifies interested parties of source changes.

The Kustomize Controller is a Kubernetes operator that operates with the Kustomize SDK of Kubernetes. It is notified of source changes by the Source Controller, fetches the artifacts from the source, and generates the corresponding Kubernetes objects. Once the Kustomize Controller detects changes between the desired and the cluster states, it immediately patches the objects.

The Helm controller is the counterpart to the Kustomize Controller that manages Helm chart releases from a source by creating KRMs based on the release. Every Helm release is defined in a Kubernetes Custom Resource called HelmRelease, which describes the desired state and is compared with the actual state on the cluster.

Both Kustomize and Helm controllers support the idea of multi-tenancy, where tenants receive their specific namespaces to operate. The Kubernetes objects created by the controllers are isolated from other tenants and cannot access other objects by defining the namespace of another tenant in the source. FluxCD works with the Role Based Access Control API of Kubernetes to handle authorization of tenants.

[15] FluxCD accepted to CNCF [Com23c]
[16] FluxCD graduated project [dho22]
[17] GitHub Flux2 [Flu23]
[18] FluxCD [Por+20]
[19] Flux2 [Flu23]

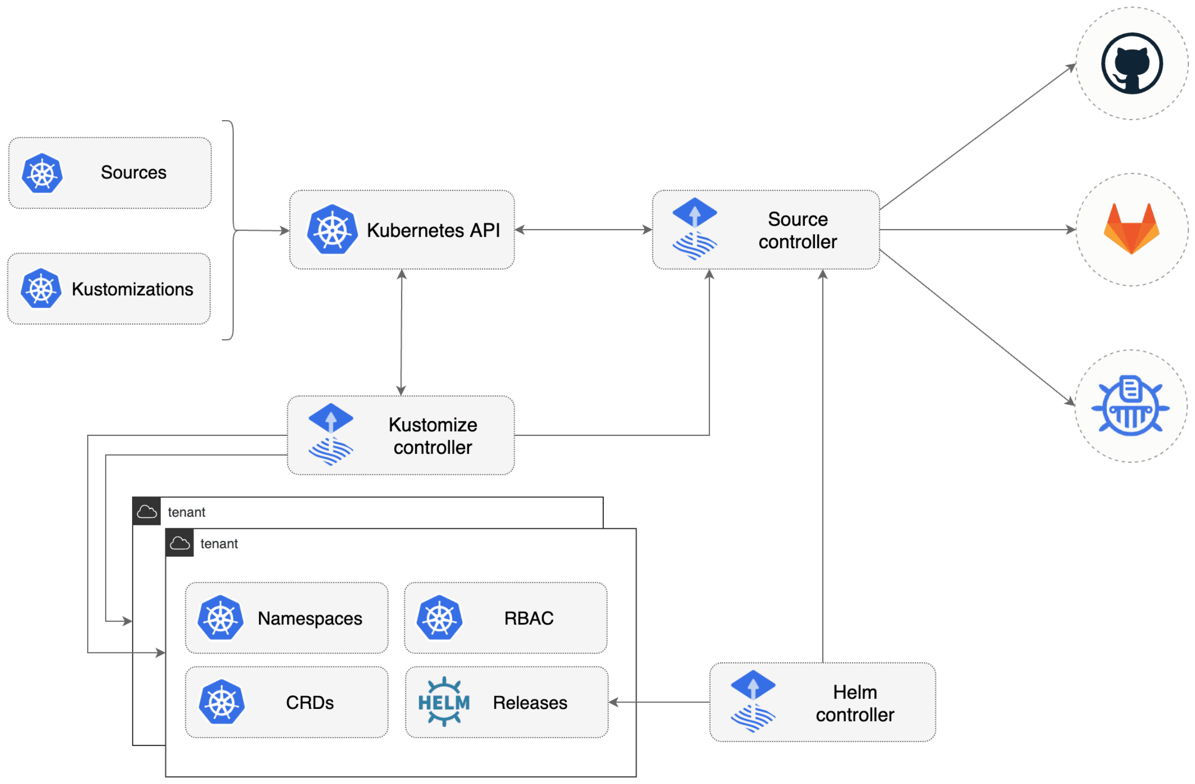


Figure 2.9.: Flux Components [Flu23]

2.7.2. Flux Bootstrap

Bootstrapping Flux[20] refers to installing the set of controllers on a cluster and mapping it to sources like GIT repositories. FluxCD offers the Flux bootstrap CLI, a tool that lets cluster operators choose the controllers to use and the sources to be connected. The Helm, Kustomize, Notification, and Source controllers are installed by default. One has to provide the source information and credentials when bootstrapping a cluster. The bootstrap process connects to the source (e.g., GitHub repository) and creates a repository for the infrastructure containing a folder with the cluster name. Flux CLI adds the components manifests to the created repository in the cluster directory and deploys them on the Kubernetes cluster. The first step is only the installation of FluxCD, where the cluster is configured to run the FluxCD controllers and registers itself in a GIT repositories, but no source is configured.
A manifest has to be created and pushed to the repository in the cluster directory to add a source for the cluster. The following manifest creates a GitRepository source containing the URL of the source and the interval, defining the period FluxCD should poll the source for changes in the artifacts. Fluxctl CLI can also create sources instead of manually writing a YAML file.

```yaml
---
apiVersion: source.toolkit.fluxcd.io/v1beta2
kind: GitRepository
metadata:
  name: podinfo
  namespace: flux-system
spec:
  interval: 30s
  ref:
    branch: master
  url: https://github.com/laurentdina/podinfo
```

Listing 2.1: Git Repository Source in FluxCD

[20]Demystifying GitOps-Bootstrapping Flux [Alt22]

The source controller finds the source manifest in the repository and fetches it to create the GitRepository source object on the cluster.
The manifest only defines the source, not the package to be deployed. As the next step, we have to configure a Kustomization or Helm package source from a path in the same URL. A second manifest defines the path to the application in the GIT source. The podinfo repository contains the Kustomization with the manifests of the packaged deployment.

```
---
apiVersion: kustomize.toolkit.fluxcd.io/v1beta2
kind: Kustomization
metadata:
  name: podinfo
  namespace: flux-system
spec:
  interval: 5m0s
  path: ./kustomize
  prune: true
  sourceRef:
    kind: GitRepository
    name: podinfo
  targetNamespace: default
```

Listing 2.2: Kustomization Source in FluxCD

This configuration triggers the Source controller to fetch the artifacts from the package source. The Kustomize controller then instantiates the package from the path and generates the Kustomization and Kubernetes objects on the cluster. The interval defines when the Kustomize controller should compare the actual state of the cluster with the desired state defined in the source controller. The interval of the FluxCD Kustomize manifest should always be greater than the one from the Git source because multiple code changes in a short period should not lead to unstable multiple recreations of objects on the cluster.

2.7.3. Managing an extensive Kubernetes Fleet with FluxCD

The example in the last subsection described a simple setup of FluxCD on a cluster to deploy workloads from a Git Repository containing an application release in the form of Kustomization deployment. FluxCD supports onboarding multiple clusters and tenants for more complex scenarios, allowing the loose coupling of developers and operators. Operators manage the fleet of clusters and their resources but also register the repositories of the tenants on the clusters (e.g., production and staging clusters) and ensure tenants are distinct from access to resources. Developers create either Kustomization deployments or Helm releases from an application repository and push them to the tenant's repository. The FluxCD controllers on the Kubernetes clusters fetch new changes made in the tenant repository and update the running resources on the cluster.

However, in this thesis, we are more interested in managing hundreds of clusters and their workloads. While for most infrastructure providers, a few clusters and tenants are sufficient. Some need to have control of a large fleet distributed over long distances. Can FluxCD also be used to spin up new Kubernetes clusters? Weaveworks, the organization and driver behind FluxCD, introduced GitOps with the Kubernetes Cluster API (CAPI)[21] to manage clusters in addition to workloads. The combination of GitOps and the Cluster API enables the declarative configuration of clusters in the form of Git pull requests. The Cluster API consists of three controllers: Cluster API Core Manager, Bootstrap Provider, and Infrastructure Provider. The Cluster API Core Manager handles the cluster lifecycle without any infrastructure details. When deploying a virtual machine, the Bootstrap Provider generates initialization scripts to set up a Kubernetes node. The infrastructure provider defines the creation and management of infrastructure resources needed to run the Kubernetes environment, such as a cloud network, virtual machines, gateways, and more.

Working with Git Checkouts and Pull Requests to manage clusters becomes very arduous at a large scale. The same applies to managing workloads by bootstrapping clusters and onboarding new repositories when extending the fleet. Creating repositories now and then and keeping pace with the workload becomes a significant challenge, especially if we need to temporarily deploy a group of clusters for a specific use case. It takes time until the GitOps environment is set up, and the packages are deployed. The environment has to be dismantled

[21]Manage Thousands of Clusters with GitOps and the Cluster API [Wea20]

again, and a set of clusters need to be deployed somewhere else. There needs to be more scalability and agility.

Another approach based on FluxCD presented at the Kubecon 2022 in Valencia[22] consists of a collection of shell scripts.[23] It uses a central management cluster to deploy workloads on worker clusters using the kubeconfig files to access the clusters and create the Kustomize deployments. The approach introduces the concept of workspaces, where multiple clusters reconcile the same workloads from a common workspace repository, allowing deployments to multiple clusters with one upload to the source. Nevertheless, initializing new clusters by attaching them with shell scripts and kubeconfigs is only feasible with a few clusters, and applying patches to deployments between clusters in a workspace takes much work.

2.8. Comparing Nephio and FluxCD

This section describes the comparison between Nephio and FluxCD in addition to the evaluation. Both Continuous Delivery tools were developed to address different challenges but in a similar way. FluxCD is focused on large-scale telecommunications networks, which involve different network management standards and network functions from multiple vendors. It wants to solve the challenges in delivering network functions to multiple distributed edge clusters. While FluxCD can be used by any DevOps team to automatically reconcile cloud-native applications from various sources and to a Kubernetes environment. It focuses on simply synchronizing clusters with repositories by fetching artifacts in intervals and comparing them with the actual state. The comparison has been conducted based on five significant quality attributes when managing multiple clusters and workloads.

2.8.1. Functionality

Nephio and FluxCD use packaging tools to bundle Kubernetes resources for repeated deployments. FluxCD uses Helm and Kustomize, which are used by almost any DevOps team to deliver application releases in Kubernetes. FluxCD supports them by creating specialized controllers to compare actual and desired states and patches the running resources when differences are found. Various notification providers are supported to inform the team on events such as newly applied patches.

Nephio uses KPT as a packaging tool, which is barely known in the cloud-native world. KPT works similarly to Helm but focuses more on the package release's configuration and supports functions that check or automate configuration parameters. A KPT package can be a standard Helm chart with an additional Kptfile containing metadata on configurations and functions.

FluxCD reconciles what is defined in the source, meaning the releases must be configured separately and uploaded to the source. Creating deployments for different environments is done with the GIT workflow and pull requests. Nephio prefers to use package blueprints listed in a Git repository. The KPT CLI or Nephio UI are used on a management cluster to copy a blueprint package, fill it with configuration parameters, create a revision, approve, and deploy it to a deployment repository reconciled from a cluster.
Both tools can deploy to multiple clusters, but FluxCD needs multiple clusters subscribing to a single source. The full version of Nephio can mark multiple clusters based on labels (e.g., location, hardware type) as the destination for a deployment.
Nephio comes with pre-built controllers to create Kubernetes clusters in the cloud, while Weaveworks offers a fee-based feature to create clusters with pull requests using the Cluster API in Kubernetes.

2.8.2. Maturity

One of the important differences is the maturity level of both tools. FluxCD has reached graduate maturity status in CNCF and is used by many enterprises in production. It could be installed and operated without any challenges during the evaluation phase of this bachelor thesis. There is also a significant and active community behind it, and it has comprehensible documentation.
Nephio is still being developed and has yet to see an official release. The PoC version, the only available

[22]Managing Thousands of Clusters and Their Workloads with Flux [Wer20]
[23]Flux multi-cluster example repository [Wer22]

version, supports a limited set of features, such as deployments to single clusters and a Web UI. The setup lacks comprehensible documentation and has unstable components.

2.8.3. Performance

Delivering a release once it is pushed to a source takes a short time until they are being created on a Kubernetes environment. This property counts for both Nephio and FluxCD, and they even allow configuring the time interval for reconciliation loops. Nephio takes seconds until a cluster receives the artifacts because the deployment is manually triggered by pushing a package to a deployment repository. FluxCD deploys whatever changes in the source occur but waits a few minutes until it changes the actual to the desired state to not allow multiple repository changes in a short time to break the deployment.

The advantage of FluxCD regarding performance is the lower resource utilization in the control plane. The controllers are lightweight, and the clusters only fetch the sources they must sync to the local cluster. Nephio uses a centralized system with many more controllers. It has to download packages from a catalog and edit them on the management cluster before the provisioning. Nephios Web UI is poorly optimized in the proof-of-concept version and needed, together with all other management components, a VM with four virtual CPUs and 16 GB of RAM to run stable.

2.8.4. Usability

There are Web UIs available for both tools. Nephio comes with a Web UI, which can be installed optionally. It is based on the KPT UI called Backstage[24], which integrates Nephio as a plugin. It is simple: Browse a catalog of packages, choose a package, edit it, and deploy it to a remote cluster.
Weaveworks offers Weave GitOps[25], a tool to manage and monitor FluxCD instances with a UI. It can be used to manually sync sources and clusters, observe events, configure component files in repositories, and debug pipelines.

2.8.5. Scalability

The most significant quality attribute in this thesis is scalability. When managing a large fleet of Kubernetes clusters and workloads, a tool should be easy to use. FluxCD fulfills all requirements to operate a typical cloud environment with a few clusters and a dozen applications. A strict and standardized workflow is needed to set up sources and onboard them on clusters.

However, in a scenario with hundreds of distributed clusters across different cloud regions and thousands of containerized applications, it is very time demanding to create branches, perform pull requests for every deployment, and configure clusters to which packages they use for synchronization and which not. Here comes Nephio's advantage, which it has been designed for. Every cluster is associated with a deployment repository. One can choose the package to deploy, modify it and choose the deployment repositories where to push it. One does not have to specify the artifacts the clusters have to fetch. They reconcile all packages defined in their deployment repository.

For example, an international enterprise wants to deploy an application on all European cloud regions. With FluxCD, a source must be configured to be synced on all clusters. For every specific group of clusters, a source in a new repository or directory has to be created and configured on all clusters so that they can synchronize the packages. This procedure takes much time and becomes difficult to monitor. With Nephio, one can choose all European clusters by specifying a selector label that automatically selects deployment repositories associated with clusters with the Europe label. The package is pushed to all deployment repositories, and the Kubernetes clusters reconcile the workloads.

The practical part of this thesis was conducted with Nephio as it has many potentials. The exact explanation is described in Chapter 6, Challenges and Decisions.

[24]KPT Backstage Plugins [KPT23b]
[25]Weave GitOps [Wea23]

2.9. Extending Kubernetes with SRv6

A flexible infrastructure needs flexible networking. Most data centers have complex networking technologies to provide optimal and secure traffic between physical and virtual machines. Kubernetes has been designed to keep networking as simple as possible and uses the concept of CNI plugins that implement networking capabilities within a cluster. In an infrastructure with hundreds of Kubernetes clusters, a flexible networking solution is needed for Kubernetes to provide optimal and secure traffic between containerized applications.

2.9.1. Kubernetes Networking Model

A Kubernetes cluster consists of multiple nodes that host various pods to achieve benefits such as redundancy, load balancing, and failovers. Each pod receives an Internet Protocol version 4 (further referred to as IPv4) or Internet Protocol version 6 (further referred to as IPv6) address to communicate with other pods in the same cluster. The communication between pods on different nodes is to be ensured as well. Nodes have their IP addresses to communicate with each other. The Kubernetes network model has some requirements for the networking implementation for pods. Pods must communicate with each other without using Network Address Translation (further referred to as NAT). How this is achieved is not described by Kubernetes, and node communication also depends on the node's location. They could be in the same subnet or completely separated between multiple networks. Kubernetes introduced the concept of CNI plugins to address different network architectures. A CNI plugin allows pods to communicate by abstracting an overlay network between nodes without using NAT. This approach lets pods adapt to the node IP networking environment. CNI plugins can receive the task of taking over the IP address allocation and managing the IP pool.

2.9.2. Networking between Kubernetes Pods

Most CNI plugins can choose between two variants to allow pod-to-pod connectivity in a cluster. The first one is direct routing or flat networking, where the packets of pods are directly routed to the infrastructure network. With this method, it has to be ensured that the prefixes of the pod networks can be routed in the infrastructure. With flat networking, pod traffic leaves the node unmodified and is routed to another node, which forwards it internally to the destination pod. Instead of relying on NAT, direct routing enables pods to communicate directly with external hosts using their IP addresses. To summarize, there is no encapsulation, no NAT performed when pod data packets leave the node. These properties make flat networking fast but more challenging to configure, as pod network prefixes need to be routable in the underlying network of the nodes.
The alternative to direct routing is overlay networking to connect pods on different nodes in a cluster. It encapsulates the pod packets to cross the underlying networking to a pod in another node. Most CNI plugins support the use of Virtual Extensible LAN (further referred to as VXLAN) or IP-in-IP encapsulation. When pods communicate within the same cluster in overlay mode, the CNI plugin encapsulates the packets with an additional header that includes the outer source and destination IP addresses. The outer header contains the source address of the sending node, and the destination address contains the destination address of the destination node. The inner header contains the source and destination addresses of the source and destination pods. Thus, the routers in the underlying network infrastructure do not have to be aware of the pod Classless Inter-Domain Routing (further referred to as CIDR) and enable scalable operations by abstracting the underlying physical network infrastructure. Another benefit is that encapsulation enables the support of multiple tenants by creating multiple overlays between all the nodes in a cluster and therefore maintaining network segmentation and security. The drawback is that the CNI plugin must perform NAT when pods want to communicate with external destinations because it cannot span an overlay for every external node. The encapsulation also takes more bandwidth as an additional header is needed.

In this thesis, networking focuses on multiple clusters distributed across large distances. The networking between workloads on different clusters should be intelligent, efficient, and scalable. Overlay networking brings these characteristics but only between nodes in a cluster. Once traffic leaves the nodes to external hosts, NAT is performed. NAT offers limited scalability, complicates peer-to-peer communication and becomes a bottleneck with numerous concurrent connections.

What if an operator could choose the network path for a workload in a Kubernetes cluster? An operator could specify the network in a deployment, and the system would handle the rest. What about the segmentation of pod networks and reuse of address prefixes based on tenancy? Containerized applications could be directly routed to a customer's infrastructure connected to the provider's cloud network.

A large distributed architecture needs robust, direct, and scalable routing in a cloud-native way.

2.9.3. SRv6 Segment Routing over IPv6

SRv6 is a new and emerging networking technology that enhances routing capabilities by leveraging IPv6 addresses. Multiple IPv6 addresses are directly embedded into a Segment Routing Header (further referred to as SRH) within an IPv6 header as routing instructions, introducing the concept of network programming. The routing instruction defines a segment in the form of an IPv6 address. The segment's 128-bit IPv6 address, called SID, consists of the locator and the function identifier. The locator defines to which destination the packet should be routed. The function identifier part consists of a function and an optional argument defining what behavior should be executed when the packet arrives at the destination.

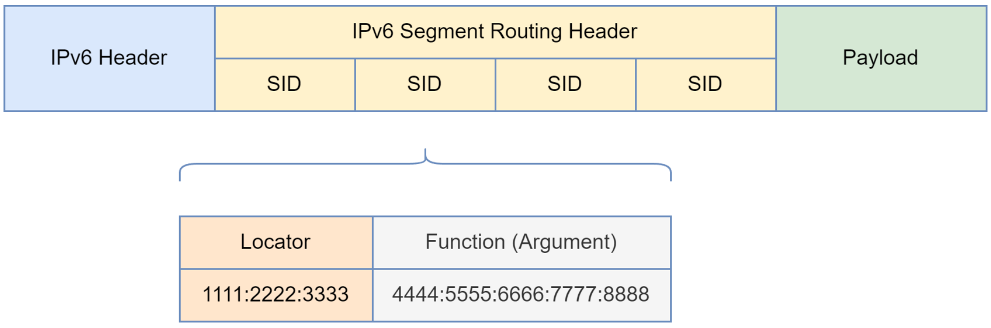


Figure 2.10.: SRv6 SID Composition

The routing in SRv6 works by setting the active segment from the list as the destination address in the outer IPv6 header. The most straightforward operation a segment could possess is the END behavior. When the packet arrives at the first segment node, the node looks up the function identifier in the source address of the header, sees an END behavior, decrements the Segment Left field in the SRH, replaces the destination address with the second SID from the segment list and forwards it to the locator address of that segment. The destination nodes define these behaviors as SIDs triggered by source nodes by specifying the SID in the destination address field of the IPv6 header. An advantage of SRv6 is that not all networking devices have to understand it. Only the source node and the destinations in the segment list called segment endpoints have to know about that mechanism. All other nodes are transit nodes - they forward the encapsulated packets based on the destination IPv6 address in the outer header.

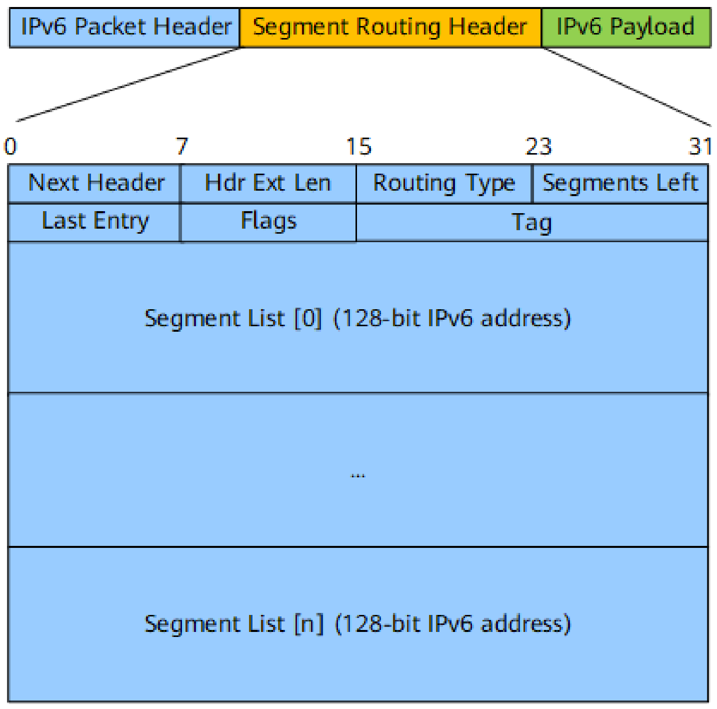


Figure 2.11.: SRv6 SRH[26]

The following behaviors[27] are most prominent in SRv6:

END - Default Endpoint

1. Decrement Segments Left
2. Update destination address
3. Forward according to the new destination address

END - Default Endpoint, then Xconnect

1. Decrement Segments Left
2. Update destination address
3. Forward on the interface associated with the Xconnect segment

END.DX4 - Endpoint with Decapsulation and IPv4 Xconnect

1. Segments Left must be 0
2. Next Header must be IPv4
3. Decapsulate the inner packet
4. Forward on the interface associated with the Xconnect segment

[27]SDN lecture: SRv6 [Met23]

END.DX6 - Endpoint with Decapsulation and IPv6 Xconnect

1. Segments Left must be 0
2. Next Header must be IPv6
3. Decapsulate the inner packet
4. Forward on the interface associated with the Xconnect segment

END.DT4 - Endpoint with Decapsulation and IPv4 Table lookup

1. Segments Left must be 0
2. Next Header must be IPv4
3. Decapsulate the inner packet
4. Look up the IPv4 destination of the inner packet and forward accordingly

END.DT6 - Endpoint with Decapsulation and IPv6 Table lookup

1. Segments Left must be 0
2. Next Header must be IPv6
3. Decapsulate the inner packet
4. Look up the IPv4 destination of the inner packet and forward accordingly

Most functions are used at the last node of a path to trigger a particular behavior. Typical behaviors are END.DT4 and END.DT6, which nodes use for VPN de-capsulation. SRv6 enables L3VPN, creating VPN tunnels for tenants through a backbone network, allowing overlapping network prefixes between tenants. A tenant has a network connected to the Provider Edge (further referred to as PE) and wants to send traffic to another proprietary network connected to another PE using the L3VPN service of the provider. The tenant's packets (IPv4 or IPv6) are encapsulated with an IPv6 header by the source node (PE), containing the SID with the VPN de-capsulation function to send it to the endpoint PE that has to de-capsulate it and forward it to the correct tenant network based on the function. The SID has to be the last or only SID in the segment list. Once the PE receives the packet, it looks up the destination SID in the destination IPv6 field, compares it with its function, sees it has a VPN de-capsulation behavior, strips off the IPv6 header and forwards it to the second tenant network. The payload of the encapsulated IPv6 packets consists of the original IPv4 or IPv6 packets sent from the tenant to the PE. In this case, the source node has to know what SIDs to define in the segment list to reach the right router and trigger the intended behavior. A routing protocol like BGP is used to inform routers what SID with which behavior (in this case, END.DT4 or END.DT6) they have to use when encapsulating customer traffic to reach a specific network connected to another PE.
An entire path can be defined in the segment list to steer the packets through multiple networks without the routers knowing where all networks are located. The routers become "stateless" as the source node defines the routing. The ability to select efficient and intelligent network paths by providing a list of segments that a packet has to traverse is called Traffic Engineering.

SRv6 brings many benefits, including:

- Simplified Network Operations: Embedding network programs at a source node eliminates the need for complex network overlays, simplifies the routing, and reduces operational overhead in the control plane of routers.
- Traffic Engineering: A source node can program an ideal path to improve network performance by defining the path of segments in a segment list in the SRH. Every node can choose the best path to achieve connectivity attributes like load balancing, low latency, high bandwidth, and many other benefits.

- Scalability and Flexibility: SRv6 leverages the never exhausting IPv6 address space to define network programming. SRv6 avoids the need for maintaining a per-flow state in the network by encapsulating packets with segments.
- L3VPN: SRv6 also supports L3VPN by encapsulating the tenant's or customer's packets and forwarding them through the backbone with the SID of the PE router that connects another tenant's network. The PE decapsulates and forwards the IP payloads to the other tenant network. This approach separates the VRF of tenants, making VPN tunnels secure and fast without needing encryption like Internet Protocol Security (further referred to as IPsec)-based VPN.

This section describes how powerful, scalable, and intelligent SRv6 can be. In the context of cloud infrastructure, this is an ideal solution to provide flexible networking between Kubernetes clusters. The behaviors END.DT4 and END.DT6 significantly benefit pod networking as they could separate routing between non-related pods and interconnect workloads between clusters.

2.9.4. Cilium CNI Plugin

Cilium is a CNI plugin for Docker and Kubernetes and is based on the Linux kernel technology extended Berkeley Packet Filter (further referred to as eBPF). It is a CNCF project that provides, secures, and observes network connectivity between containers. Cilium offers high-performance networking, transparent encryption, visibility, and fine-grained security policies. Many big tech companies use it to scale Kubernetes clusters with direct routing or overlay mode. It can enforce broader network policies than the default Kube-proxy policies and provide better security and observability. Cilium also provides extensive Layer 3 network functionalities and can enforce policies from Layer 3 to Layer 7. These properties make it one of the most popular CNI plugins for Kubernetes, and it is available for selection when creating a managed Kubernetes cluster at the big three cloud providers AWS, Microsoft Azure, and Google Cloud Platform.
Cilium is mainly written in Golang and was launched by Isovalent[29]. It is available as an open-source project on GitHub[30] can be installed with Helm or with the Cilium CLI.

Figure 2.12.: Cilium Icon [28]

Cilium is one of the first CNI plugins to support SRv6 for Kubernetes. The clusters in this thesis are deployed with Cilium as CNI plugin to use this powerful networking technology. The routing protocol BGP was available long before. It is used to route entire pod prefixes of clusters in an extensive network, avoiding static routing sprawl when using the direct routing approach. BGP is also used to simplify the control plane of SRv6 by advertising the SIDs for L3VPN services of pods belonging to different tenants.

The BGP and SRv6 configurations are created as custom resources in manifests. The continuous delivery tool Nephio is used to provision the configurations to the clusters. In the first step, BGP peerings between the clusters and a PE router are configured to advertise the pod CIDRs enabling connectivity between pods in different clusters. Next, SRv6 is configured to enable L3VPN for pods with specific VRF labels. Cilium generates the SIDs with END.DT4 and END.DT6 behaviors and advertises them to the peer with BGP. Another node receives the SRv6 SIDs with BGP through the PE, uses them to encapsulate packets in an IPv6 header, and sends them to the Kubernetes cluster. Cilium hooks the destination address with eBPF in the Linux kernel and does an SID lookup. There will be a mapping to a VRF associated with a pod, and based on the routing table, it will be forwarded to the pod belonging to the tenant. Other tenants cannot send traffic without an encapsulation policy with the SID and the destination prefix.

[29] Isovalent - eBPF-based networking, security, and observability [Iso22]
[30] GitHub Cilium [Cil22]

CHAPTER 3

Methods

3.1. General

The methods chapter addresses the methods used to reach our use cases. Starting with the Kubespray deployment of Kubernetes to prepared VMs. Next, we explain the network infrastructure we designed and configured in LTB for the finished deployments. They are followed by deploying Nephio and what is needed to connect edge clusters with their management cluster. We then show how we installed and configured the Cilium CNI plugin to deploy L3VPN workloads to edge clusters and what an extension to the public cloud would look like. Last, we briefly overview the complete GitLab deployment pipeline.

3.2. Kubernetes

3.2.1. Preparations

Before tackling the installation of Kubernetes on your VM, primary conditions must be met and preparations fulfilled. We decided to go with Kubeadm as a basis for the Kubernetes environment as it is powerful, widely adopted, and featured by various tools. Kubeadm is best-practice for a multi node Kubernetes environment. The downsides of Kubeadm when it comes to resource utilization are worth mentioning. The minimal requirements are listed below. Any deployment using Ansible requires an SSH connection used for running commands on the VMs. As this thesis is written in the form of a PoC, we used standard password authentication instead of public-private-key authentication, which is best practice. We chose Containerd as container runtime as it is used in the industry and we have worked with it before. The SSH connection to the virtualized nodes was provided by the INS and changed within the project to another method but with the same functionality and minor changes. Having a fully functional forward and reverse DNS server running in the network where the clusters are placed is best practice. This allows to use hostnames instead of IPs, providing more flexibility.

```
OS: Ubuntu 22.04 LTS (or other, various supported)
Memory: 2GB or more
Disk: 40GB or more
CPU: 2 vCPUs
```

Listing 3.1: Minimal resources for Kubernetes with Kubeadm

Using swap in terms of memory swapping while running Kubernetes is to be avoided. Therefore swap needs to be disabled on all the nodes (control plane and workers). Most commands require escalated privileges.

```
$ sudo swapoff -a
$ sed -i '/ swap / s/^\(.*\)$/#\1/g' /etc/fstab
```

Listing 3.2: Disable swap for Kubernetes

3.2.2. Overview

For the Kubernetes deployment to the LTB, we decided to use a two-node Kubernetes environment. As there are no differences in behavior between a two-node and 100-node environment, we decided to use a minimal size-environment with challenges regarding resources in mind. The VMs are running at least two interfaces, one pointing directly to the INS network used for internet access and SSH connections and the other pointing to the infrastructure containing routers and other clusters. Both IPv4 and IPv6 are configured on the second interface allowing for different routing technologies such as SRv6 in the future. We used the pre-prepared user "ins", with privilege escalation permissions, for most tasks performed. Initially, running two Kubernetes clusters, each running on two nodes, in two different locations, emphasized by being placed in different Autonomous System (further referred to as AS), were intended. More information regarding the network design and configuration may be found in the according network section.

3.2.3. Approach

As Kubernetes has become widely adapted during the last years, the question of a standardized deployment of Kubernetes clusters has been an intensely discussed topic. Tools like Rancher and Kubespray are well-established in the industry. We did not consider Rancher, as it is a complete Kubernetes platform. Some team members did have some little experience regarding Kubespray. We evaluated Kubespray with its pros and cons superficial. In the first attempt, we decided to automate the deployment of Kubernetes by ourselves. Using a self-engineered solution brings several advantages, such as customization to make the product fit our needs as best possible. The solution would be flexible for further improvements such as new features, enhancements, or upgrades. Another critical argument with the self-engineered solution was our interest in gaining experience and learning.

3.2.4. Ansible Kubernetes Deployment

We decided to implement an Ansible deployment based on "pure" bash commands. As there are no valuable collections or best practices available.

Ansible runs with little to no configuration on most systems. Prior running tasks with Ansible host_ke_checking must be disabled within the Ansible configuration (ansible.cfg). SSH connections to nodes can not be established without the installation of sshpass[1]. These required tasks will be described more precisely within the following sections.
The following code snippet shows the plain bash command to initialize a cluster. The output generated by the cluster after a successful initialization would be registered. On a further stage, this output would be used to cut out the join command to use on the designated worker nodes. The join command contains a discovery token and a certificate hash. These arguments are specified with the control plane to create a binding.

```
$ ansible-playbook playbook.yml -i dev_inventory
```

Listing 3.3: Running an Ansible playbook with Specifying Inventory

```
- name: Step02 - cluster init
  become: true
  become_user: root
  ansible.builtin.command: kubeadm init --control-plane-endpoint=10.114.0.5 #local interface to
      listen / create bindings
  register: output_clusterinit

- debug:
```

[1]Link to the description of SSHPASS [Viv23]

```
    var: output_clusterinit.stdout_lines
```

Listing 3.4: Code Snippet as Example of Ceased Self-Developed Deployment

As the development remained incomplete, the configuration within the playbooks had to be set on a per-playbook basis. Not all variables have yet been placed in according to "host_var" and "group_var" folders or been inserted using templating engines like Jinja2. Therefore, one playbook per host is being run by Ansible.

```
inventory.ini
playbook_deploy-ClustersMaster.yml
playbook_deploy-ClustersWorker01.yml
playbook_deploy-ClustersWorker02.yml
```

Listing 3.5: Files to Run a Deployment (Excluding Variables Folders)

Due to various challenges and project schedules, the group decided to switch to Kubespray well-founded. The external expert also raised the possibility of using Kubespray at the mid-term presentation. The project advisor was consulted to have the decision approved. The next chapters will discuss the challenges that led to this decision and Kubespray.

Difficulties with joining a cluster

The group faced various challenges during the deployments with the self-engineered playbooks. Some behavior was also suspicious while installing the cluster by hand on the VMs deployed within the provided LTB.

Multiple interfaces - Certificates

As mentioned, multiple interfaces were attached to the VMs. Running a single node Kubernetes cluster does not present a problem, as no workers must be added to the cluster. While initiating the cluster, certifications are created. The certificates contain on the one hand the hostname and on the other hand, IPv4 addresses. As the certificate contains IPv4 addresses which differ from the IPv4 we specified, the join command throws an error and aborts. This was partly solved by disabling the first interface, on which the internet connection and SSH connection were running on. It became clear that Kubernetes uses the first interface's IPv4 address and localhost addresses for its certificates. Specifying the endpoint interface did not help (as shown below). Research in the internet showed, that this issue with multiple interfaces was known.

```
$ sudo kubeadm init --control-plane-endpoint=10.33.10.11
  --pod-network-cidr=10.2.0.0/16,fd00:a::/64 --service-cidr=10.96.0.0/24,2001:db8:a::/112
      --skip-phases=addon/kube-proxy
```

Listing 3.6: Specification of the Interface while Initiating the Cluster - the Certificate does not contain this IP

Low Resources

Another challenge was dealing with the low resources available within the provided VMs. The VM had about 1.8GB of memory and 9GB of disk space (4.5GB left). Troubleshooting and logs showed that some critical processes were killed or frequently restarted and, therefore not listening for requests. Having this issue discussed with our advisor, a new LTB topology was deployed. The issue was resolved but other persisting challenges avoided the successful join of worker nodes.

Container Runtime not Ready

Having overcome challenges such as the low resources and running with multiple interfaces, the container runtime would not be ready when the init command was entered. As this issue was known in the internet some years ago, there were multiple possible solutions to be tested. One solution was to pause the deployment for 2-3 minutes to give the container time more time. The second approach was to remove the Containerd configuration file[2]. After a service restart, the issue should be resolved. The solution proved to be effective.

[2] Link to the resources [Sah23]

```
$ rm /etc/containerd/config.toml
$ systemctl restart containerd
$ kubeadm init
```

Listing 3.7: Commands to remove the containerd configuration file

Kubeadm Services not Answering Requests
As the group resolved most problems with workarounds or tricks mentioned above, connecting to the control plane should have been no problem. An unexpected problem, such as encountering not answering services, occured. The dedicated worker was in a waiting loop after entering the command to join the cluster. Troubleshooting and investigating the situation with TCPdump[3] showed that the packets were reaching their destination and a TCP connection was established. Despite the TCP connection being established, no packages with payload were sent or received.

3.2.5. Kubespray

With the project schedule already having advanced, the team had to decide how to go on with the Kubernetes deployment. As the team encountered multiple challenges and some of them were not fixable to bring the deployment to a responsible state of maturity, the decision to switch to Kubespray was made.
Kubespray is strict with different software versions, this chapter is being handled within the section "Deployment" as software is being installed within the GitLab pipeline.
As Kubespray is based on Ansible, only necessary configuration will be described in detail.

Ansible configuration

Some settings for a successful deployment using Ansible must be set in advance within the Ansible configuration file. The configuration has to be set within the "ansible.cfg" file, which should to be present in the Ansible working directory to avoid specifying the configuration file location. An "ansible.cfg" file has been placed within the repository and working directory to keep it as straightforward as possible.

The most important statements are shown below and described inline.

```
host_key_checking=False # is a security feature, but comes with challenges , e.g. when deploying
    new nodes with the same IP address
gathering = explicit # fact gathering must be enabled explicitely within a playbook or playbook
    command
deprecation_warnings=False # depreceated modules and collections are being used, therefore the
    warning should be disabled
```

Listing 3.8: Important statements of ansible.cfg file

Kubespray is quite vulnerable to not correctly placed files. Therefore, the sample folder (already containing an inventory file) "/kubernetes/inventory/sample" should be copied and renamed to the current project at the same location, such as "bachelorthesis". We decided to use the format ".ini" instead of the newer ".yaml" as there are no differences in functionality, but most resources use an "ini" file. The inventory file needs to be placed within this folder, located at "/kubernetes/inventory/bachelorthesis". This folder contains all necessary files folder-dependencies for the deployment. As the group needed to decide to deploy single node Kubernetes clusters, the following inventory file shows a typical deployment for a single node environment. Deploying a single node Kubernetes cluster requires node1 to be added to the control plane group, etcd group, and the kube_node group. The interface to connect via SSH connection is specified with 'ansible_host' whereas the 'ip' needs to be set to the interface where the Kubernetes binding shall be established on. Adding more workers is effortless, as these additional nodes only need to be added to the kube_node group. Existing groups like [edge_controller_node], [calico_rr], and more shall not be deleted, as these are required by Kubespray.

[3]Link to the resources [lin23]

The group "management_controller_node" is essential for the Nephio deployment and must be specified, as this specifies the management controller. The provided web interface by the controller allows the deployment and management of workloads.

```
[all]
node1 ansible_port=22 ansible_user=*** ansible_ssh_pass=yourPW ansible_host=10.18.10.25
     ip=10.33.10.11 ansible_become=yes ansible_become_password=***
#node2 ansible_port=20005 ansible_user=*** ansible_ssh_pass=yourPW ansible_host=10.18.10.7
     ip=10.33.10.11 ansible_become=yes ansible_become_password=***

[all:vars]
ansible_connection=ssh

[kube_control_plane]
node1

[etcd]
node1

[kube_node]
node1
#node2

[kube_virtualmachines:children]
kube_control_plane

[management_controller_node]
management-node ansible_port=20002 ansible_user=*** ansible_ssh_pass=*** ansible_host=10.18.10.25
     ansible_become=yes ansible_become_password=***

[edge_controller_node]
node1

#Do not delete
[calico_rr]

#Do not delete
[k8s_cluster:children]
kube_control_plane
kube_node
calico_rr
```

Listing 3.9: Sample Inventory File

Kubernetes Deployment Configuration

The Kubernetes configuration file, originally "k8s-cluster.yml" located at "/kubernetes/inventory/bachelorthesis/group_vars/", needs to be customized. This customization is needed in order to be ready for future CNI plugin deployments such as Cilium. All configuration entries are required and must net left blank.
Below, a fully functional configuration file shown and the important values marked and described beneath.

```
---

kube_config_dir: /etc/kubernetes
kube_script_dir: "{{ bin_dir }}/kubernetes-scripts"
kube_manifest_dir: "{{ kube_config_dir }}/manifests"
kube_cert_dir: "{{ kube_config_dir }}/ssl"
kube_token_dir: "{{ kube_config_dir }}/tokens"
kube_api_anonymous_auth: true
kube_version: v1.26.3
local_release_dir: "/tmp/releases"
retry_stagger: 5
kube_owner: kube
kube_cert_group: kube-cert
kube_log_level: 2 # needs to be defined prior deployment
credentials_dir: "{{ inventory_dir }}/credentials"
kube_network_plugin: cni # needs to be defined prior deployment
kube_network_plugin_multus: false
kube_service_addresses: 10.97.0.0/16# needs to be defined prior deployment
kube_pods_subnet: 10.7.0.0/16 # needs to be defined prior deployment
kube_network_node_prefix: 24
enable_dual_stack_networks: true # needs to be defined prior deployment
kube_service_addresses_ipv6: 2001:db8:7b::/112 # needs to be defined prior deployment
kube_pods_subnet_ipv6: 2001:db8:37::/112 # needs to be defined prior deployment
kube_network_node_prefix_ipv6: 120
kube_apiserver_ip: "{{ kube_service_addresses|ipaddr('net')|ipaddr(1)|ipaddr('address') }}"
kube_apiserver_port: 6443
kube_proxy_remove: true # needs to be defined prior deployment
kube_proxy_mode: ipvs
kube_proxy_strict_arp: false
kube_proxy_nodeport_addresses: >-
  {%- if kube_proxy_nodeport_addresses_cidr is defined -%}
  [{{ kube_proxy_nodeport_addresses_cidr }}]
  {%- else -%}
  []
  {%- endif -%}
kube_encrypt_secret_data: false
cluster_name: example.net # needs to be defined prior deployment
ndots: 2
dns_mode: coredns
enable_nodelocaldns: true
enable_nodelocaldns_secondary: false
nodelocaldns_ip: 169.254.25.10
nodelocaldns_health_port: 9254
nodelocaldns_second_health_port: 9256
nodelocaldns_bind_metrics_host_ip: false
nodelocaldns_secondary_skew_seconds: 5
enable_coredns_k8s_external: false
coredns_k8s_external_zone: k8s_external.local
enable_coredns_k8s_endpoint_pod_names: false
resolvconf_mode: host_resolvconf
deploy_netchecker: false
skydns_server: "{{ kube_service_addresses|ipaddr('net')|ipaddr(3)|ipaddr('address') }}"
skydns_server_secondary: "{{ kube_service_addresses|ipaddr('net')|ipaddr(4)|ipaddr('address') }}"
dns_domain: "{{ cluster_name }}"
container_manager: containerd
kata_containers_enabled: false
kubeadm_certificate_key: "{{ lookup('password', credentials_dir + '/kubeadm_certificate_key.creds
     length=64 chars=hexdigits') | lower }}"
k8s_image_pull_policy: IfNotPresent
kubernetes_audit: false
default_kubelet_config_dir: "{{ kube_config_dir }}/dynamic_kubelet_dir"
podsecuritypolicy_enabled: false
volume_cross_zone_attachment: false
persistent_volumes_enabled: false
event_ttl_duration: "1h0m0s"
auto_renew_certificates: false
kubeadm_patches:
  enabled: false
  source_dir: "{{ inventory_dir }}/patches"
  dest_dir: "{{ kube_config_dir }}/patches"
```

Listing 3.10: Sample Kubernetes Cluster Configuration File

Some of the values are explained in more detail below:
kube_log_level:

- Used to set the granularity of logging. Default is 2, set to 5 for troubleshooting.

kube_network_plugin:

- As Kubespray support the deployment of various CNI plugin like Cilium, Flannel and Calico, the desired CNI plugin may be set here. Set to "cni" as a basic implementation, for a deployment in a later stage.

kube_service_addresses:

- Set unique and non-overlapping IPv4 network for Kubernetes services. Example: 10.97.0.0/16.

kube_pods_subnet:

- Required, set an unique and non-overlapping IPv4 network for Kubernetes pods. Shall not overlapp with specified services network as well. Example: 10.7.0.0/16.

enable_dual_stack_networks:

- Default is "false" for environments using IPv4 only. Set to "true" for enhanced networking features such as SRv6.

kube_service_addresses_ipv6:

- Set unique and non-overlapping IPv6 network for Kubernetes services. Example: 2001:db8:7b::/112.

kube_pods_subnet_ipv6:

- Required, set an unique and non-overlapping IPv6 network for Kubernetes pods. Shall not overlapp with specified IPv6services network as well. Example: 001:db8:37::/112.

kube_proxy_remove:

- Default is 'false'. Set to 'true' if CNI plugin, like Cilium, shall overtake more networking functions.

cluster_name:

- Set a name for your new Kubespray cluster environment. This name must be unique to avoid DNS issues.

As Kubespray is based on Ansible and run within an GitLab pipeline, the deployment can be startet with the following command as well. The flag '–skip-tags=multus' is necessary, as there may be difficulties with newer versions of Kubespray. Multus is a CNI plugin which enables attaching multiple networks to a pod. Disabling multus within the ordinary Kubernetes does not seem to work.

```
$ - ansible-playbook -i inventory/bachelorthesis/inventory.ini --become --skip-tags=multus
    pb_main.yml
```

Listing 3.11: Ansible Playbook Command

Handling secrects

Credential stealing and other cybersecurity risks have increased in recent years. Therefore, dealing with secrets is important, especially as secrets are being deployed to Kubernetes nodes. Two main approaches have been discussed at the mid-term presentation. Security is not only necessary on the node, but also within the repositories. One possible approach would be, to store secrets within an encrypted Ansible vault file. Security of secrets within pods and nodes is another topic, whereas produckts like 'bitnami sealed one-way secrets'[4]sealed secrets offer solution for secrets throughout the application lifecycle. This topic is also being covered in the chapter 'discussion' within the section 'Further Work'.

[4]Link to bitnami sealed secrets [VmW23]

3.3. Network

3.3.1. Preparations

To execute and test the multiple deployments, a base network first had to be designed and configured. The designed network simulated a large-scale cloud provider with three different AS regions. It connected the separate Kubernetes clusters, which were built to test various deployments during this thesis. Highly capable routers were needed to use advanced network functionalities like BGP and SRv6. The INS at OST provided the infrastructure based on our design, and it was proficient enough to enable said network protocols. The devices used for this infrastructure were virtualized and connected in LTB, an infrastructure virtualization tool, by INS. Six Ubuntu Linux VMs and one Cisco IOS XR router (further referred to as XR1) were used in the final infrastructure of this thesis. One Ubuntu Linux VM was used as a VPN gateway to exterior public clouds, and two VMs were solely used for testing purposes and will not be mentioned any further. This leaves our main infrastructure with three VMa.
The Ubuntu Linux VM contained the following resources.

```
Operating System: Ubuntu Linux 22.04
Processor: 4 vCore Intel Xeon
Memory size: 16 GB
Disk size: 30 GB
```

Listing 3.12: Resource specification of Ubuntu Linux VM

And the Cisco IOS XR router was of the following version.

```
Cisco IOS XR Software, Version 7.8.1 LNT
Build Information:
    Built By     : ingunawa
    Built On     : Wed Nov 30 13:00:28 UTC 2022
    Build Host   : iox-lnx-108
    Workspace    : /auto/srcarchive13/prod/7.8.1/xrd-control-plane/ws
    Version      : 7.8.1
    Label        : 7.8.1
```

Listing 3.13: Used Cisco IOS XR Version

An SSH access over the LTB was granted to all devices by the INS. As the devices were virtualized in LTB this SSH connection via the LTB gave us, unlike a normal SSH connection, full access to the devices, especially Ubuntu Linux VMs, as if we would have directly plugged in a monitor to the machine. With this connection in place, the configuration of all devices was possible.

3.3.2. Overview

The three regions, defined by different AS numbers 65000, 66000, and 67000, simulated a large-scale cloud provider. The central AS 66000 served as the management region, responsible for controlling and overseeing the two other regions. This region contained the Cisco IOS XR (XR1) router and one Ubuntu Linux VM. XR1 was strategically positioned within and on the edge of the management region and acted as a simple backbone. The usage of a Cisco IOS XR router allowed for the replication of the robustness and scalability found in carrier-grade networks.
The other regions AS 65000 and AS 67000 had to receive updates from the management region AS 66000. Both edge regions contained a single Ubuntu Linux VM, connected by a single link to XR1 in the management region. As we later show, no additional routers on the edge regions were needed to connect the different AS. The Ubuntu Linux VMs in the edge regions were equipped with Kubernetes. This allowed for the installation of later needed tools like Cilium and Nephio. The AS 65000 contained the Kubernetes cluster 'Edge Cluster A' and the AS 67000 contained 'Edge-Cluster-B'. Because all applications are later carried out by the Kubernetes cluster, we no longer refer to the individual VMs, but directly to the Kubernetes cluster 'Edge-Cluster-A' and 'Edge-Cluster-B'.
To communicate between 'Edge-Cluster-A' and 'Edge-Cluster-B' in different AS, a specific protocol called BGP was needed. BGP allows for exchanging routing and reachability information among separate AS. BGP was configured between XR1 and the edge regions.

To be able to route IPv4 traffic over a IPv6 based network and have pod-to-pod connectivity between 'Edge-Cluster-A' and 'Edge-Cluster-B', SRv6 was used. Cilium CNI plugin allows VPN connections between pods in different edge clusters.
All device interfaces were connected according to the network plan shown in figure 3.1.

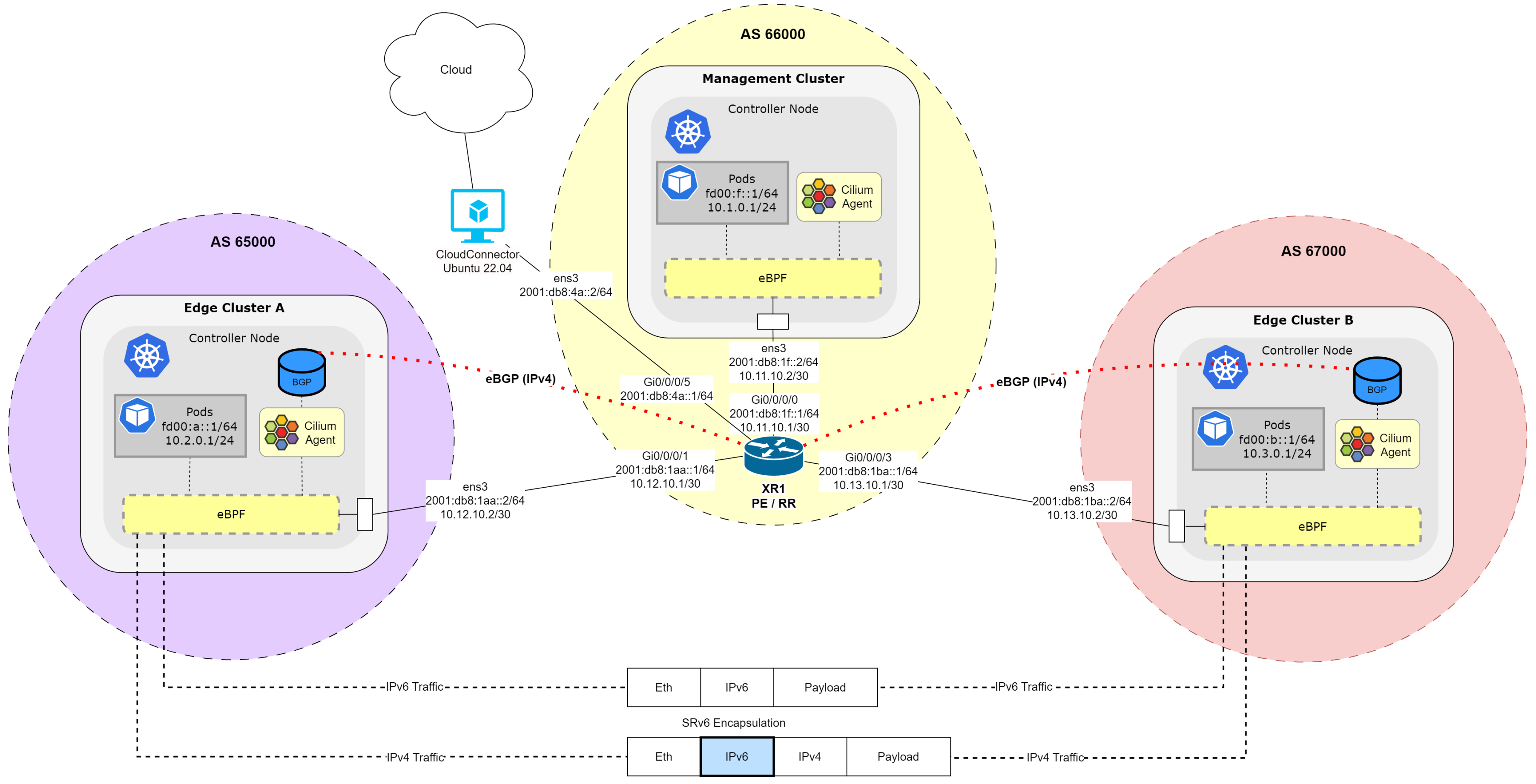


Figure 3.1.: Network Overview

3.3.3. Internet Protocol

In order to establish seamless communication between devices, it was crucial to assign them an IPv4 or IPv6 address. To accommodate sites that had yet to transition to IPv6-only, a dual stack approach was adopted between the two sites, A and B. This approach involved configuring both IPv4 and IPv6 on these sites, allowing them to handle and support communication in both protocols. This ensured seamless connectivity across the network, regardless of the current IPv6 adoption status of individual sites. Consequently, IPv6 configuration was implemented on nearly all devices to ensure their readiness for advanced networking features.
The IPv6 address scheme was based on the RFC3849 standard[5], which defines IPv6 addresses in subnet 2001:DB8::/32 reserved for documentation purposes.
The IPv6 subnets assigned between XR1 and edge clusters A, and B were based on cluster and device names. For example, the edge-cluster-A-controller node and XR1 connection were assigned the IPv6 subnet 2001:db8:1aa::/64. The number '1' in the third quarter '1aa' resembles the router XR1, the first letter 'a' resembles the cluster 'A', and the second letter 'a' resembles the node. The node was given extra lettering to have multiple nodes in the same cluster for future projects.
The IPv4 address scheme was based on the RFC1918 standard[6], which defines IPv4 addresses in subnet 10.0.0.0/8 reserved for private use. Similar to the IPv6 address scheme, the second octet was used to define the connection between 'Edge-Cluster-X' and XR1 in the core. The third octet describes the node, and the fourth octet the device.

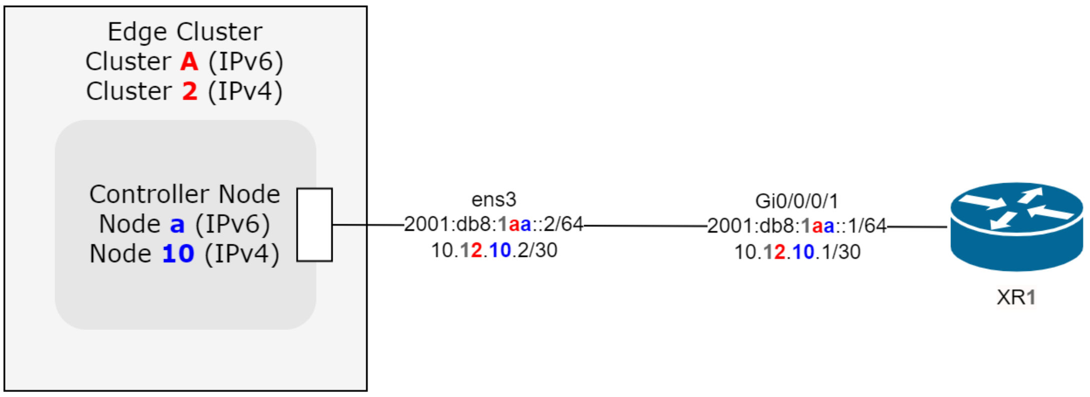


Figure 3.2.: IPv6 Address Schema between Edge Cluster A and XR1

As an example of the network configurations made, XR1 is shown in the listing below.

```
$ show run | begin interface
interface GigabitEthernet0/0/0/0
    description to Management-Cluster
    ipv4 address 10.11.10.1 255.255.255.252
    ipv6 address 2001:db8:1f::1/64
!
interface GigabitEthernet0/0/0/1
    description to Edge-Cluster-A-Controller
    ipv4 address 10.12.10.1 255.255.255.252
    ipv6 address 2001:db8:1aa::1/64
!
interface GigabitEthernet0/0/0/3
    description to Edge-Cluster-B-Controller
    ipv4 address 10.13.10.1 255.255.255.252
    ipv6 address 2001:db8:1ba::1/64
!
```

Listing 3.14: Network Configurations XR1

The following figure 3.3 gives an overview of the Internet Protocol (further referred to as IP) connections only that were implemented.

[5]RFC3849 standard: [TAC04]
[6]RFC1918 standard: [Sys+96]

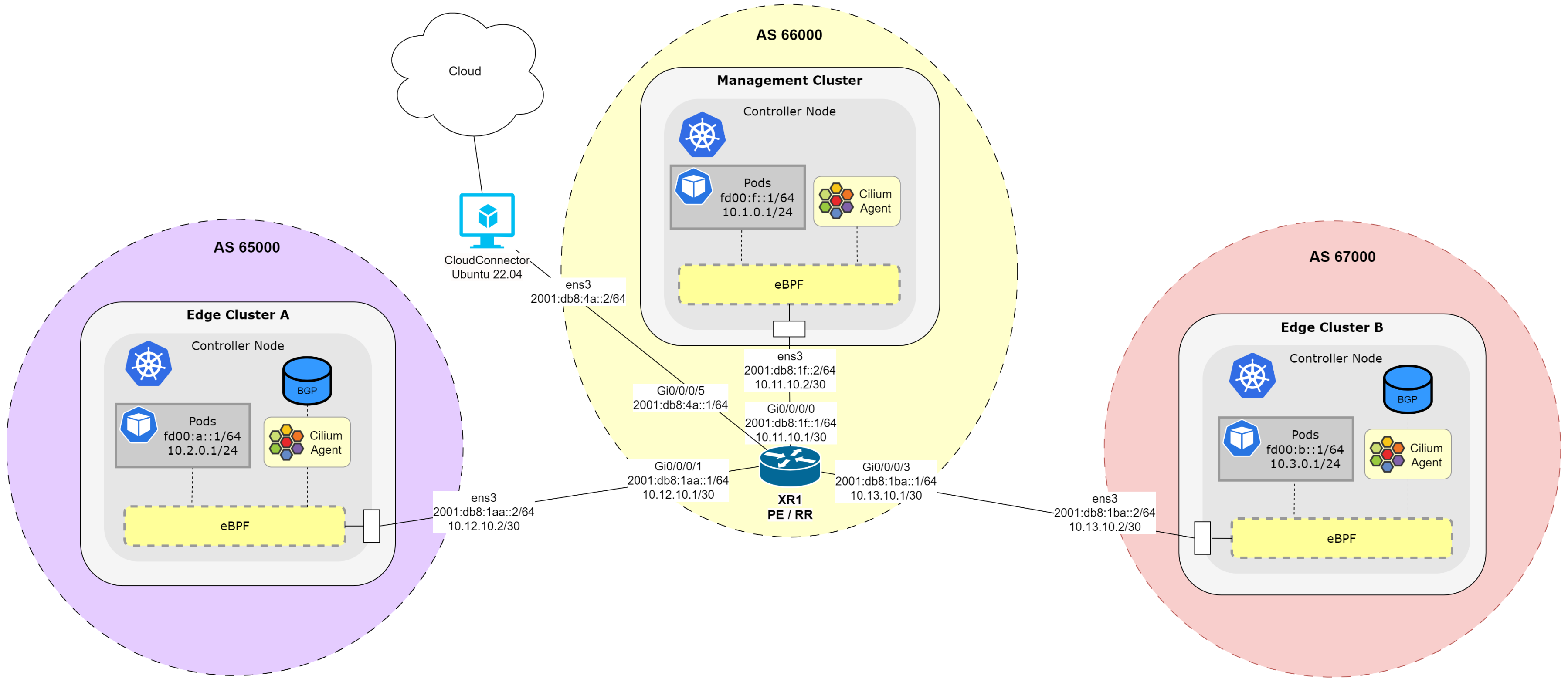


Figure 3.3.: Network Plan of IP Connections

3.4. Nephio

3.4.1. Overview

With Nephio, we were able to introduce our Kubernetes clusters with the cutting-edge GitOps approach. This groundbreaking methodology allowed us to seamlessly manage configurations using the power of GIT and the flexibility of Kubernetes. To work with Nephio, it had to be installed on the management cluster, the 'Edge-Cluster-A' and the 'Edge-Cluster-B'. The installation on the management cluster was done manually, as this only needed to be done once. For Nephio on the 'Edge-Cluster-A' and 'Edge-Cluster-B', an Ansible script was created to automatically install Nephio on targeted clusters.
A common GIT repository was needed to communicate between the different clusters, making all necessary files effortlessly accessible to all clusters. These GIT repositories acted as sources of truth and provided real-time visibility into the desired state for each edge cluster. The repositories later stored the specified BGP and SRv6 workloads, which needed to be implemented in Cilium on 'Edge-Cluster-A' and 'Edge-Cluster-B'.
How these parts interact with each other is shown in figure 'Nephio Setup Overview' 3.4.

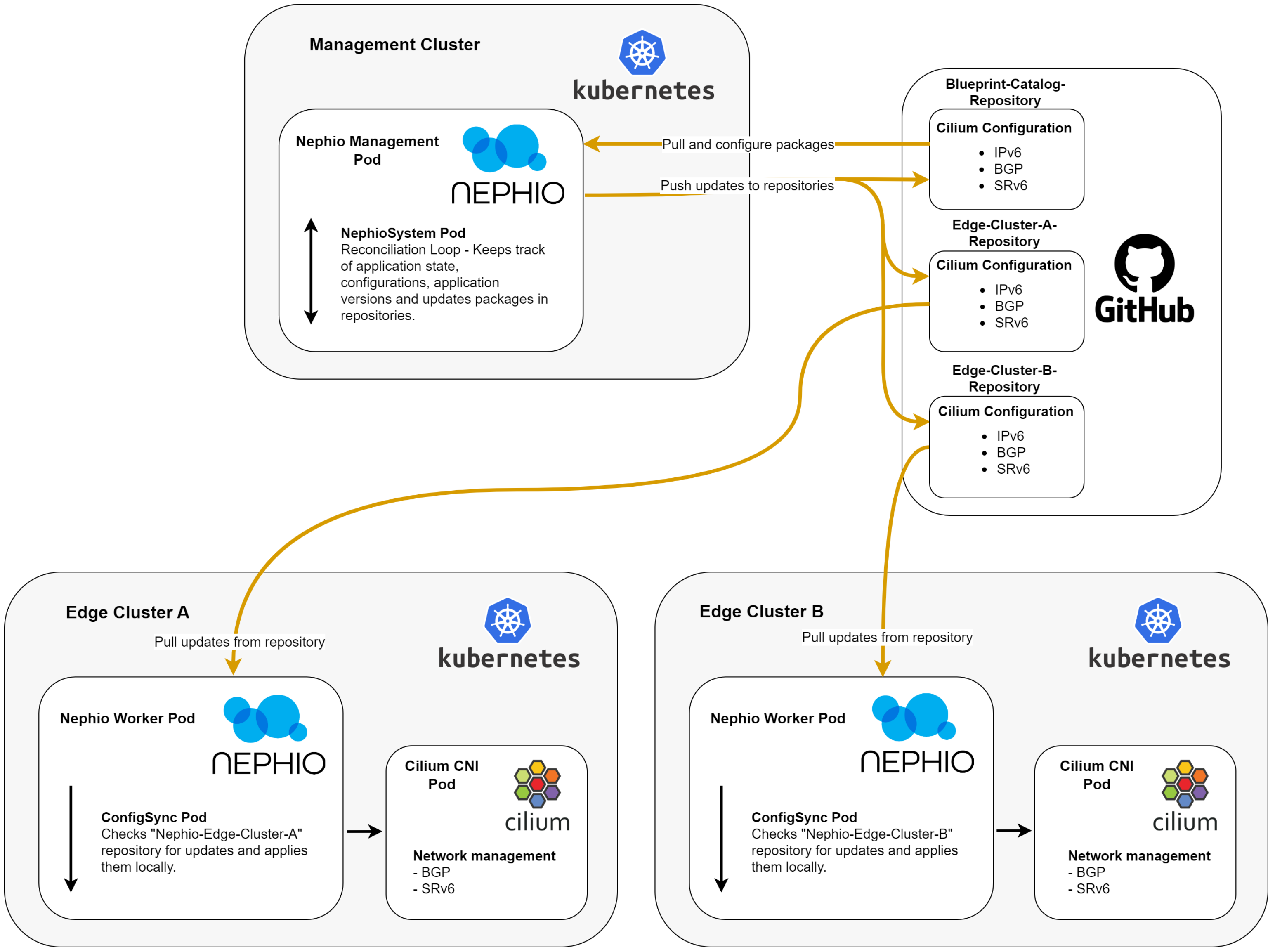
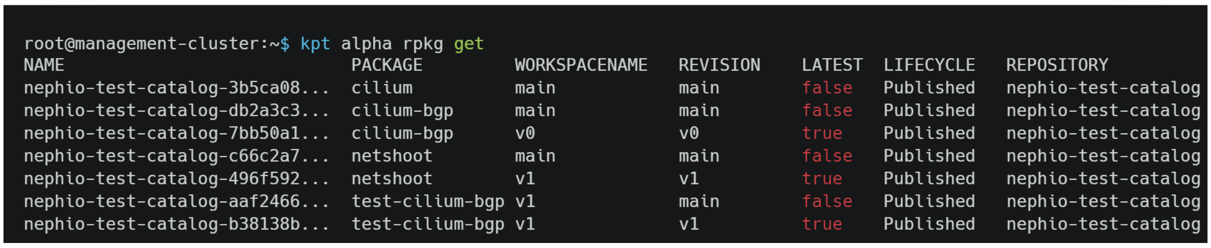

Management Cluster
kubernetes
Nephio Management Pod
NEPHIO
NephioSystem Pod
Reconciliation Loop - Keeps track of application state, configurations, application versions and updates packages in repositories.
Pull and configure packages
Push updates to repositories
Blueprint-Catalog-Repository
Cilium Configuration
IPv6
BGP
SRv6
Edge-Cluster-A-Repository
Cilium Configuration
IPv6
BGP
SRv6
Edge-Cluster-B-Repository
Cilium Configuration
IPv6
BGP
SRv6
GitHub
Edge Cluster A
Pull updates from repository
kubernetes
Nephio Worker Pod
NEPHIO
ConfigSync Pod
Checks "Nephio-Edge-Cluster-A" repository for updates and applies them locally.
Cilium CNI Pod
cilium
Network management
- BGP
- SRv6
Edge Cluster B
Pull updates from repository
kubernetes
Nephio Worker Pod
NEPHIO
ConfigSync Pod
Checks "Nephio-Edge-Cluster-B" repository for updates and applies them locally.
Cilium CNI Pod
cilium
Network management
- BGP
- SRv6

Figure 3.4.: Nephio Setup Overview

3.4.2. Nephio on the Management Cluster

The management cluster is where new edge clusters had to be added, new deployments could be made, and information about configuration files or already made deployments were gathered. It acted as the central management point for all connected edge clusters and provided a single point of contact.
To use Nephio-System on the management cluster, the cluster was configured as described in the chapter A 'Installation Manuals' with the Nephio PoC installation on top. KPT was used to interact with the finished Nephio installation. With KPT, we were able to publish different workloads to our edge clusters. These workloads or packages first had to be created. The created workloads were published to our installation repository, 'nephio-test-catalog'. The published installation packages inside the 'nephio-test-catalog' repository could then be viewed on the management cluster as shown in figure 3.5.



```
root@management-cluster:~$ kpt alpha rpkg get
NAME                           PACKAGE          WORKSPACENAME   REVISION   LATEST  LIFECYCLE   REPOSITORY
nephio-test-catalog-3b5ca08... cilium           main            main       false   Published   nephio-test-catalog
nephio-test-catalog-db2a3c3... cilium-bgp       main            main       false   Published   nephio-test-catalog
nephio-test-catalog-7bb50a1... cilium-bgp       v0              v0         true    Published   nephio-test-catalog
nephio-test-catalog-c66c2a7... netshoot         main            main       false   Published   nephio-test-catalog
nephio-test-catalog-496f592... netshoot         v1              v1         true    Published   nephio-test-catalog
nephio-test-catalog-aaf2466... test-cilium-bgp  v1              main       false   Published   nephio-test-catalog
nephio-test-catalog-b38138b... test-cilium-bgp  v1              v1         true    Published   nephio-test-catalog
```

Figure 3.5.: Installation Packages of 'nephio-test-catalog' on Management Cluster

The shown packages were now ready to be installed on an edge cluster. Deployments could either be done via KPT CLI or with an optional Web UI provided by Nephio. The web UI offered a more accessible and intuitive interaction with Nephio. Therefore, we installed the Web UI on the management cluster as described in the chapter installation manual A.

3.4.3. Nephio on the Edge Cluster

Edge clusters, or workload clusters by Nephio, are clusters that do not contain the Nephio system itself but are intended to run the workloads deployed via Nephio. The edge clusters have to run KPT Config Sync[7] to get the workloads from their assigned deployment repository. KPT Config Sync checks the connected repository for changes in a specified interval and locally applies changes.
The edge clusters with Nephio Config Sync were installed with our Ansible deployment, described in section 3.5.

3.4.4. Repositories for Nephio

Nephio makes use of the GitOps approach and stores packages in repositories. There are two types of repositories. One type is used to store available packages for Nephio to use, and the other is mapped to specific edge clusters and contains the packages installed on them. The Nephio system can have multiple repositories registered to pull packages from, displayed by the 'Public-Nephio-Catalog-Repository', which is a publicly available repository from Nephio and our own 'Nephio-Test-Catalog-Repository', which contains our Cilium packages.
The following diagram showcases the interaction between 'Nephio-System', 'Nephio-ConfigSync', and the different repositories.

[7]KPT ConfigSync [KPT23e]

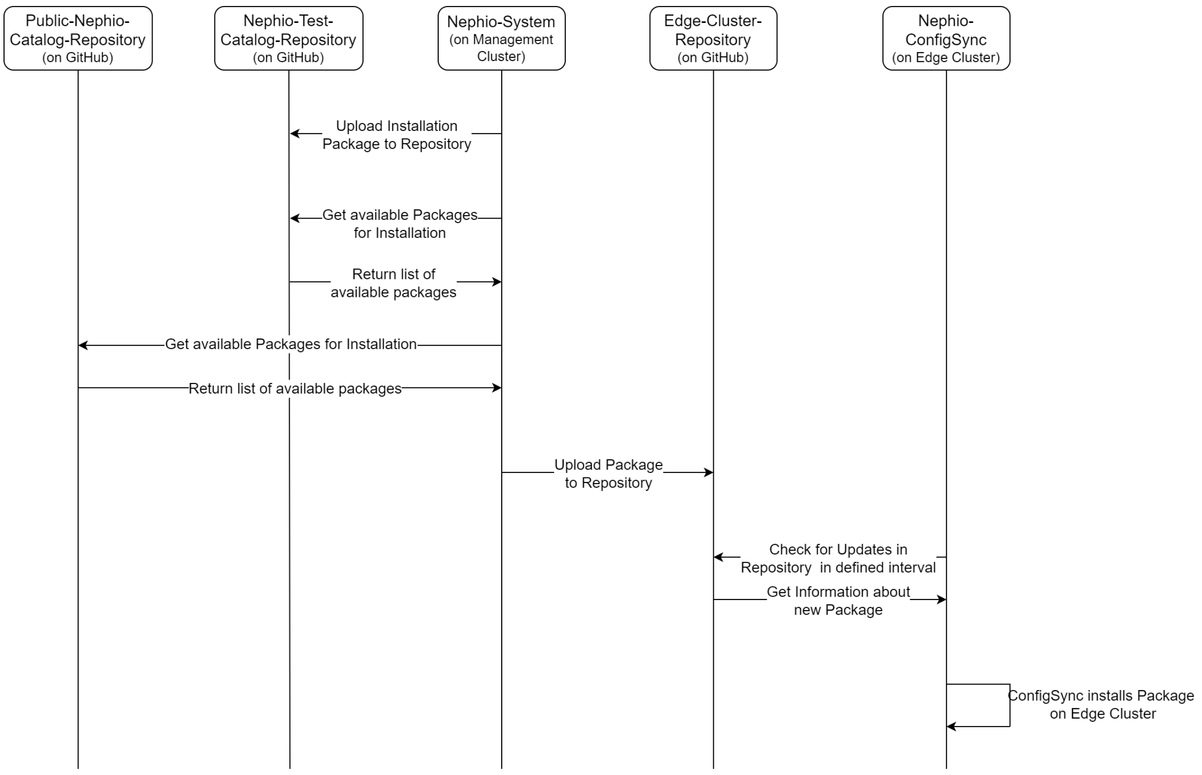

Public-Nephio-Catalog-Repository (on GitHub)
Nephio-Test-Catalog-Repository (on GitHub)
Nephio-System (on Management Cluster)
Edge-Cluster-Repository (on GitHub)
Nephio-ConfigSync (on Edge Cluster)
Upload Installation Package to Repository
Get available Packages for Installation
Return list of available packages
Get available Packages for Installation
Return list of available packages
Upload Package to Repository
Check for Updates in Repository in defined interval
Get Information about new Package
ConfigSync installs Package on Edge Cluster

Figure 3.6.: Nephio Interaction Flow Diagram

3.5. Nephio Deployment

As the management cluster only needs to be installed once, the management cluster is installed according to the installation manual in chapter c. The Nephio Config Sync on the edge clusters was installed with Ansible. Figure 3.7 shows the steps needed to install Nephio on a new edge cluster 'Edge-Cluster-A'. Each step is described in more depth in the next section.

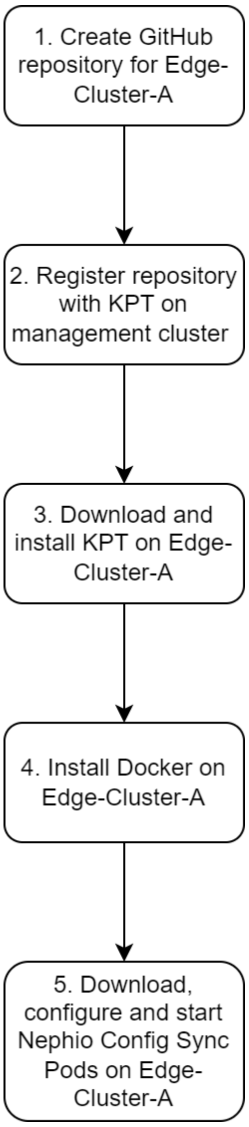

1. Create GitHub repository for Edge-Cluster-A
2. Register repository with KPT on management cluster
3. Download and install KPT on Edge-Cluster-A
4. Install Docker on Edge-Cluster-A
5. Download, configure and start Nephio Config Sync Pods on Edge-Cluster-A

Figure 3.7.: Installation Flow of Nephio ConfigSync on Edge Cluster A

3.5.1. Deployment Steps

Create GitHub Repository

On top of the Kubernetes installation, Nephio needs some additional tools to work correctly. First, a GitHub repository has to be created. At the time of writing this thesis, Nephio was only available as a PoC and only supported GitHub and Google Cloud Source repositories. This repository could not be created with the 'community.general.github_repo' Ansible module as mentioned in chapter 7 Notable Points. Therefore the 'github_configòrganization_api_url' module was used to make a HTTP POST request to the GitHub API to create a GitHub repository with a README file.
To be able to create a GitHub repository with Ansible, a GitHub token has to be created with sufficient permissions to CRUD a repository.

Register Repository on Management Cluster

The previously created GitHub repository must be added to the management cluster. This is for the management cluster to know which repository should be used to communicate with 'Edge-Cluster-A'. The repository needs to be registered with KPT as a deployment repository in order for KPT to recognize this repository as a repository ready for workload deployments.

Download and Install KPT on Edge-Cluster-A

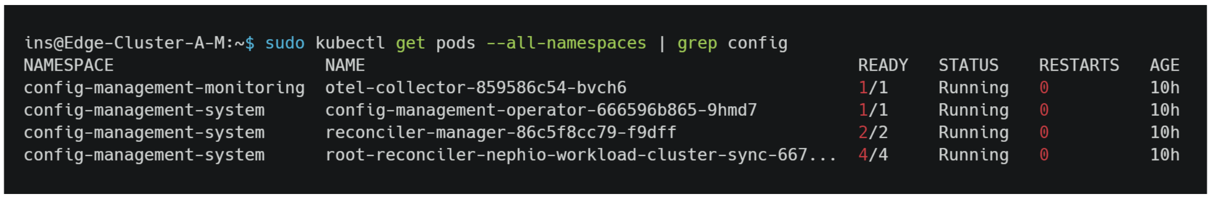
To run Nephio on the edge cluster, an installation of KPT was needed. The KPT binaries could be downloaded from its main GitHub repository[8] and copied to the '/usr/local/bin/kpt' folder to be accessible from the CLI on the 'Edge-Cluster-A'.

Install Docker on Edge-Cluster-A

To run containers in a pod, Kubernetes uses a container runtime.[9] Docker.io[10] was installed on the edge cluster as the container runtime.

Download, configure and start Nephio Config Sync on Edge-Cluster-A

The installation of Nephio Config Sync was done according to the installation manual on the Nephio PoC GitHub page[11].
The edge clusters needed to run Config Sync to get their workloads from their deployment repositories. To install Config Sync, there was a package 'nephio-confisync'[12] available, which could be downloaded with KPT. Because this package was generic, some modifications needed to be done to work the previously created repository for 'Edge-Cluster-A'. The Nephio RootSync resource[13] contains a predefined GitHub repository for testing purposes called `https://github.com/nephio-test/test-edge-01`. This repository entry in the 'rootsync.yaml' file needed to be replaced with our specifically created repository for 'Edge-Cluster-A' and its authentications. To replace this entry, the resource could either be updated manually directly in the file or with KPT functions. To implement our own changes, the KPT functions were used as described on the Nephio PoC repository. Ultimately, these changes had to be applied with KPT, and the pods could be started. Nephio Config Sync is running when the following pods are running.



```
ins@Edge-Cluster-A-M:~$ sudo kubectl get pods --all-namespaces | grep config
NAMESPACE                    NAME                                              READY   STATUS    RESTARTS   AGE
config-management-monitoring otel-collector-859586c54-bvch6                    1/1     Running   0          10h
config-management-system     config-management-operator-666596b865-9hmd7       1/1     Running   0          10h
config-management-system     reconciler-manager-86c5f8cc79-f9dff               2/2     Running   0          10h
config-management-system     root-reconciler-nephio-workload-cluster-sync-667...  4/4  Running   0          10h
```

Figure 3.8.: Nephio Config Sync Running on Edge-Cluster-A

[8] GitHub KPT [KPT23a]
[9] Install Kubeadm on Linux [kub23a]
[10] Ubuntu docker.io package [Dev20]
[11] GitHub Nephio PoC [Nep23b]
[12] Nephio Config Sync Package [Nep22a]
[13] Nephio RootSync Resource GitHub [Nep22b]

3.6. Cilium

The theory part describes how the networking model in Kubernetes works and that Cilium CNI plugin supports the routing technologies BGP and SRv6 to enable L3VPN between clusters. This section explains the setup of Cilium with suitable configuration values needed to enable the control planes of BGP and SRv6. Cilium is installed and configured as Helm release that is deployed through a set of Ansible playbooks in a GitLab CI/CD pipeline. The GitLab CI/CD pipeline installs and configures Cilium as Helm release through a set of Ansible playbooks. The CI/CD pipeline runs these playbooks after the Kubespray playbook that prepares an entire cluster, which is then completed with the installation of Cilium and becomes ready for production. The resources are delivered on every cluster with Nephio.

3.6.1. Cilium Configuration Values

Listing 3.15 shows all required configuration values to enable BGP and SRv6 in Cilium. The configuration values in the map 'helm_values' are injected into the Cilium operator during the Cilium release installation on the cluster. These configurations can be changed afterward in the Cilium ConfigMap on Kubernetes. The values before 'helm_values' are used to install a release from the correct Helm chart version. The initial Kubespray deployment configures the pod CIDRs for IPv4 and IPv6, and Cilium uses them for address allocation. Remark: The listing contains the chart of the open-source Cilium version, but in this thesis, the playbooks contained a chart from a private repository that supports the SRv6 functionality.

```yaml
---
kubeconfig: /root/.kube/config
cilium_config:
  helm_name: cilium
  helm_chart: cilium/cilium
  helm_repo: https://helm.cilium.io/
  helm_version: 1.13.1
  helm_release_name: cilium
  helm_namespace: kube-system
  helm_values:
    kubeProxyReplacement: strict
    operator:
      replicas: 1
    k8sServiceHost: 10.13.10.2 # set internal ip of node.
    k8sServicePort: 6443
    ipv4:
      enabled: true
    ipv6:
      enabled: true
    ipam:
      mode: kubernetes
    k8s:
      requireIPv4PodCIDR: true
      requireIPv6PodCIDR: true
    tunnel : disabled
    ipv4NativeRoutingCIDR: "10.0.0.0/8" # set IPv4 address of local network
    ipv6NativeRoutingCIDR: "2001:db8::/48" # set IPv6 address of local network
    bgpControlPlane:
      enabled: true
    srv6:
      enabled: true
      encapMode: srh
```

Listing 3.15: Cilium Installation Helm Parameters

Some values are explained in more detail:

kubeProxyReplacement:

- kube-proxy is a component in Kubernetes responsible for managing network connectivity and load balancing for services within the cluster. Cilium takes complete control of these functionalities by setting the value to strict.

k8sServiceHost and k8sServicePort:

- k8sServiceHost and k8sServicePort specify the host and the port of the Kubernetes service for which Cilium should handle the network traffic. It is necessary to specify the interface used for routing.

ipam:

- The Kubernetes mode for IP address management enables Cilium to assign IPv4 and IPv6 addresses dynamically to pods based on the Kubernetes API server. Thus, Cilium uses the prefixes defined in the Kubernetes cluster initialization.

requireIPv4PodCIDR and requireIPv6PodCIDR:

- These flags ensure that the cluster has appropriate IPv4 and IPv6 addresses configured for pods, which is a requirement for the SRv6. They are used to validate the configured IPv4 and IPv6 pod CIDRs.

tunnel:

- The tunnel setting is one of the essential flags when enabling SRv6. Cilium uses VXLAN by default as overlay mode to connect pods between nodes. This mode has the disadvantage of using NAT to reach destinations outside the cluster. This "tunnel" mode has to be turned off and use direct (native) routing because SRv6 can not work with NAT.

ipv4NativeRoutingCIDR and ipv6NativeRoutingCIDR:

- These flags define the CIDR ranges of IPv4 and IPv6 addresses that Cilium should use for forwarding and routing traffic within the cluster. If the pod prefixes are not contained in the native routing CIDR blocks, the SRv6 encapsulation mechanism Cilium will not hook the pod's outgoing traffic.

bgpControlPlane:

- This flag enables the BGP control plane to connect to external peers to advertise pod prefixes. It is also used to advertise the SIDs of the L3VPN service.

srv6:

- The Cilium operator enables the SRv6 control plane and data plane on every Cilium agent.

3.6.2. Cilium Playbooks

The Cilium deployment over Ansible consists of three playbooks. The first one installs Helm on the destination node. The second one uses the previously installed Helm to deploy Cilium with the configuration parameters above. The last playbook installs the Cilium CLI tool to check the status of the local Cilium operators and agents. Listing 3.16 shows the playbook that deploys Cilium utilizing the Kubernetes Core module from Ansible. It uses Jinja2 to render the configuration with the parameters stated above.

```yaml
---
- name: Cilium Installation
  hosts: kube_virtualmachines
  gather_facts: no
  tasks:
    - name: Add Cilium repository
      kubernetes.core.helm_repository:
        name: "{{ cilium_config.helm_name }}"
        repo_url: "{{ cilium_config.helm_repo }}"

    - name: Install Cilium CNI with Helm
      kubernetes.core.helm:
        state: present
        kubeconfig: "{{ kubeconfig }}"
        chart_ref: "{{ cilium_config.helm_chart }}"
        release_name: "{{ cilium_config.helm_release_name }}"
        release_namespace: "{{ cilium_config.helm_namespace }}"
        wait: yes
        values:
          "{{ cilium_config.helm_values }}"
      become: true
```

Listing 3.16: Cilium Installation Helm Parameters

The separate ZIP archive contains all Ansible playbooks used in this thesis.

3.6.3. BGP and SRv6 Resources

To create flexible routing with BGP and SRv6, one has to create a few custom resources for Cilium. The first manifest is the BGP Peering Policy, which tells the BGP control plane to peer with the router XR1. For successful peering, the local AS, peer address, and remote must match on both Kubernetes and the router. The key 'exportPodCIDR' tells Cilium to advertise its pod prefixes to XR1. With this information, XR1 can route packets from other hosts destined for a pod because it knows that the pod prefix is reachable at the IPv4 and IPv6 addresses of the node. Figure 3.17 configures the peering over IPv4 and IPv6 to advertise IPv4 and IPv6 prefixes.

```yaml
---
apiVersion: "cilium.io/v2alpha1"
kind: CiliumBGPPeeringPolicy
metadata:
  name: pe1
spec:
  nodeSelector:
    matchLabels:
      kubernetes.io/hostname: edge-cluster-a
  virtualRouters:
  # Peer with XR1 and advertise Pod CIDR
  - localASN: 65000
    exportPodCIDR: true
    mapSRv6VRFs: true
    neighbors:
    - peerAddress: "10.12.10.1/32"
      peerASN: 66000
    - peerAddress: "2001:db8:1aa::1/128"
      peerASN: 66000
```

Listing 3.17: Cilium BGP Peering Policy

This policy allows other peers to route traffic to pods in a Kubernetes cluster. However, Cilium has yet to import received routes from its peer. For this reason, we have to configure static routes on the Kubernetes cluster to reach the SID and pods of other nodes. The first static route is needed to route encapsulated packets to the destination endpoint based on the SRv6 SID. The second static route is used to steer the traffic through the interface connected to the infrastructure, not the internet access interface.

```
$ ip route add 2001:db8:3::/64 via 2001:db8:1aa::1
$ ip route add 10.0.0.0/8 via 10.12.10.1
```

Listing 3.18: Cilium Static Routes

In addition to the correct BGP parameters, both peers have to support the same BGP address families. The default address families are IPv4 unicast and IPv6 unicast. The address families are sufficient to route traffic to the pods in the cluster. In order to provide L3VPN between peers, both have to support the address families Virtual Private Network version 4 (further referred to as VPNv4) unicast and Virtual Private Network version 6 (further referred to as VPNv6) unicast. VPNv4 and VPNv6 Unicast deal with routing information specific to VPN, enabling the distribution of routes between different VPN sites.
The BGP peering advertises prefixes to the entire pod CIDRs in the cluster. In a further step, SRv6 is used to advertise routes in a VRF to create L3VPN. A Cilium SRv6 VRF resource defines to which VRF pods should belong. Listing 3.17 uses selectors to assign pods to a VRF by matching a label. This resource will assign every pod labeled with 'vrf: vrf0' to VRF ID 1. Route targets define the network prefixes with a L3VPN network. Another cluster connected to the PE XR1 exports its prefix of a VRF with a route target, which the current cluster can import to a VRF with BGP. This procedure is also done in the opposite direction so that the other cluster can import the route targets from the current one. The last configuration specifies to what destination CIDR the packets should be encapsulated.

All this information is necessary for Cilium to create an SRv6 SID with the END.DT4 behavior. Cilium uses BGP to advertise L3VPN reachability information. The BGP update contains the reachable pod addresses of the VRF, at what node address the pod is reachable, and the SID to use for encapsulation. Other hosts can use the SID as the destination address in a packet and send it to the cluster, where Cilium will de-capsulate it and forward it to the pods in the corresponding VRF. Notice the key 'mapSRv6VRF' in listing 3.17. This key instructs Cilium to create an SRv6 egress policy by importing the SID and routing information advertised by a BGP peer. When a pod sends traffic to a destination CIDR of the other cluster within L3VPN, Cilium will check if the pod belongs to the correct VRF. If the destination address matches the destination CIDR, the traffic will be encapsulated and forwarded through the PE XR1.

```
---
apiVersion: cilium.io/v2alpha1
kind: CiliumSRv6VRF
metadata:
  name: vrf0
spec:
  vrfID: 1
  importRouteTarget: "65000:1"
  exportRouteTarget: "65000:1"
  rules:
  - selectors:
    - podSelector:
        matchLabels:
          vrf: vrf0
    destinationCIDRs:
    - 10.3.0.0/24
```

Listing 3.19: Cilium SRv6 VRF

The last information for the SRv6 control plane is a Kubernetes node annotation containing the node name, the router ID, and local AS to define which node in a cluster responds to SRv6 de-capsulation. This annotation should only be specified for a single node in a cluster.

```
---
apiVersion: v1
kind: Node
metadata:
  name: edge-cluster-a
  annotations:
    cilium.io/bgp-virtual-router.65000: "router-id=10.12.10.2,srv6-responder=true"
```

Listing 3.20: Node Annotation for SRv6 Responder

We bundle all these YAML manifests in a KPT package, which Nephio can deploy to the clusters.

3.7. Creating BGP and SRv6 Packages

These resources will be configured and deployed with Nephio. The Cilium resources act as network functions similar to the ones in 5G deployments. First, one must create the KPT packages containing the resource definitions as described in the chapter Installation Manual A. For the use cases, we use four different packages:

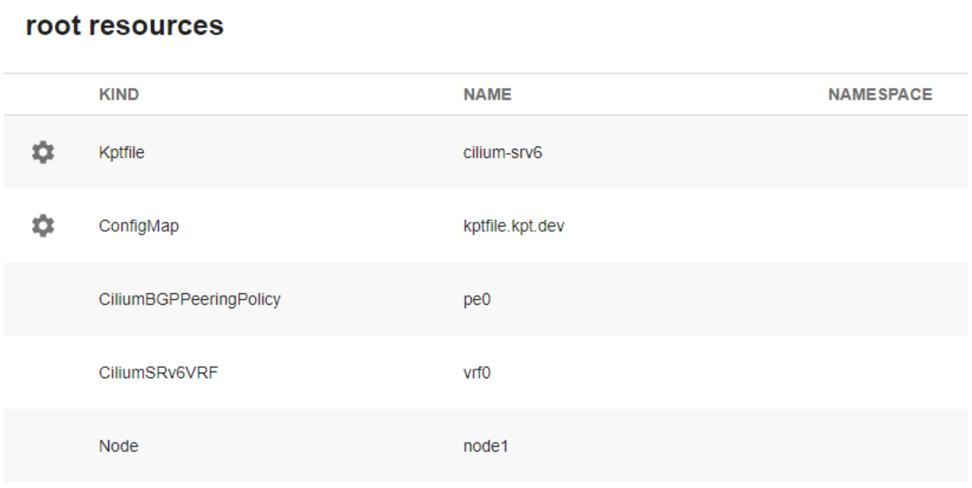
1. Cilium-BGP: A single BGP configuration to enable CE use case where the entire cluster pod prefix is advertised.
2. Cilium-SRv6: A complete SRv6 package containing BGPPeeringPolicy, CiliumSRv6VRF, and the node SRv6 responder annotation to create a complete SRv6 control plane.
3. Cilium-VRF: A single CiliumSRv6VRF resource to create additional VRFs to support L3VPN.
4. Netshoot: A Netshoot daemon set that creates a pod with installed network troubleshooting tools which is used to simulate an application or workload belonging to a VRF.

The Netshoot daemon set ensures that a replica of the Netshoot pod runs on every Kubernetes cluster. The label 'vrf' is used by SRv6 to assign the pods to vrf0.

```
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: netshoot-vrf0
spec:
  selector:
    matchLabels:
      app: netshoot
      vrf: vrf0
  template:
    metadata:
      labels:
        app: netshoot
        vrf: vrf0
    spec:
      tolerations:
      - key: node-role.kubernetes.io/master
        operator: Exists
        effect: NoSchedule
      containers:
      - name: netshoot
        image: nicolaka/netshoot:latest
        command: ["sleep", "infinite"]
```

Listing 3.21: Netshot DaemonSet

We use default configuration values in the resources, which we then modify. Figure 3.9 lists all resources in the KPT package.



root resources

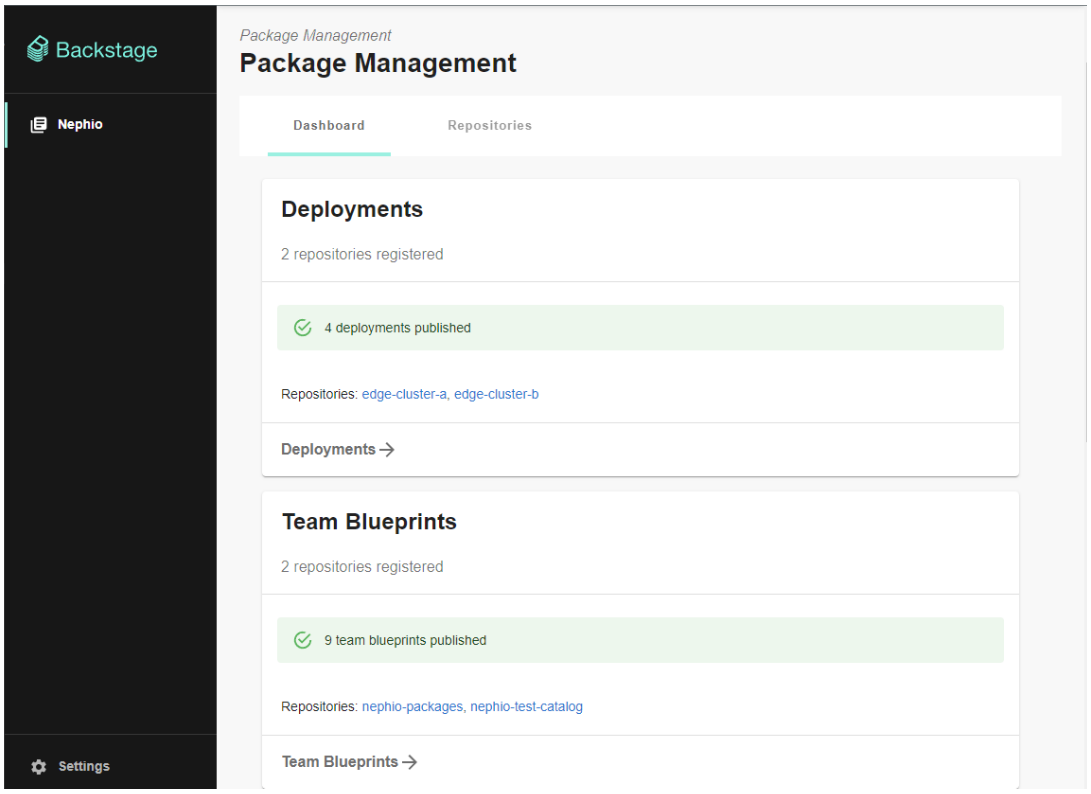
	KIND	NAME	NAMESPACE
⚙	Kptfile	cilium-srv6	
⚙	ConfigMap	kptfile.kpt.dev	
	CiliumBGPPeeringPolicy	pe0	
	CiliumSRv6VRF	vrf0	
	Node	node1	

Figure 3.9.: Resources in Cilium-SRv6 KPT Package

All packages are pushed to the private catalog on GitHub and listed in Nephio.

3.8. Deploying Workloads with Nephio

We have used the Nephio web interface to deploy packages for more convenient usage. The dashboard is straightforward and gives an overview of the deployments and blueprints. One can see the two edge clusters under the deployments.



Package Management

Deployments

2 repositories registered

4 deployments published

Repositories: edge-cluster-a, edge-cluster-b

Team Blueprints

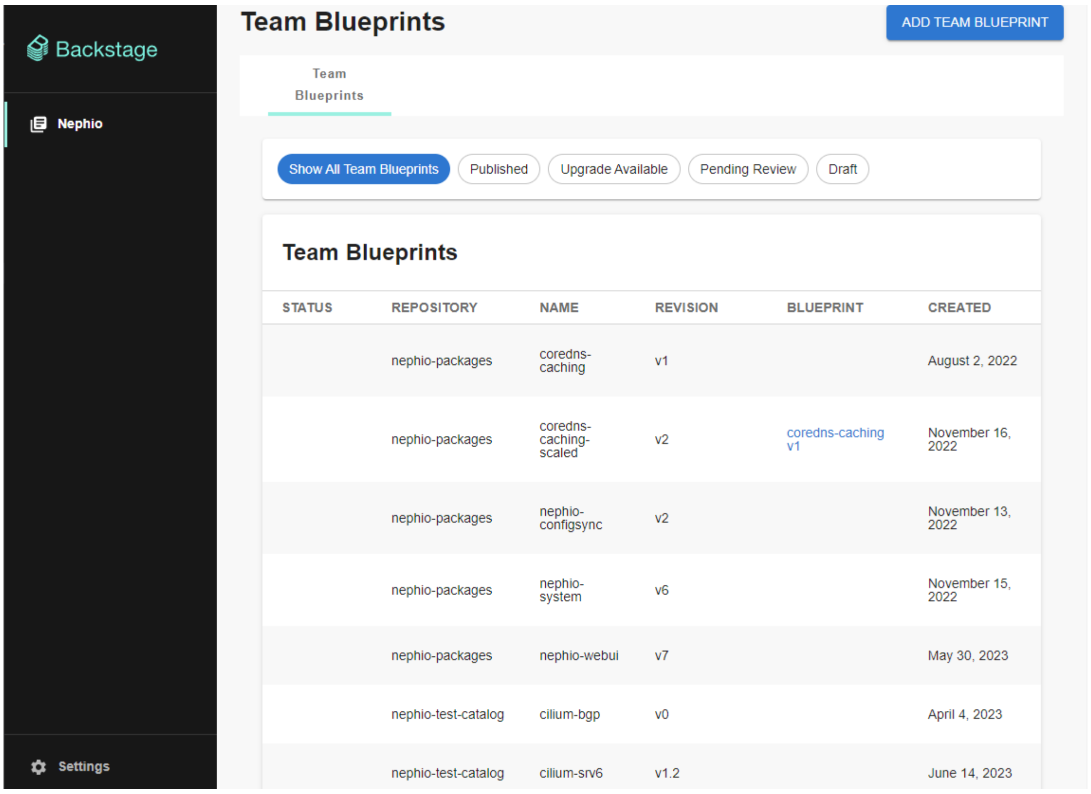
2 repositories registered

9 team blueprints published

Repositories: nephio-packages, nephio-test-catalog

Figure 3.10.: Nephio Dashboard

All team blueprints and resources are also listed. The list includes the blueprint, the repository it comes from, the name, and the current revision. The SRv6 team blueprints are in the 'nephio-test-catalog' repository, while the other packages containing Nephio install components are from a public catalog.

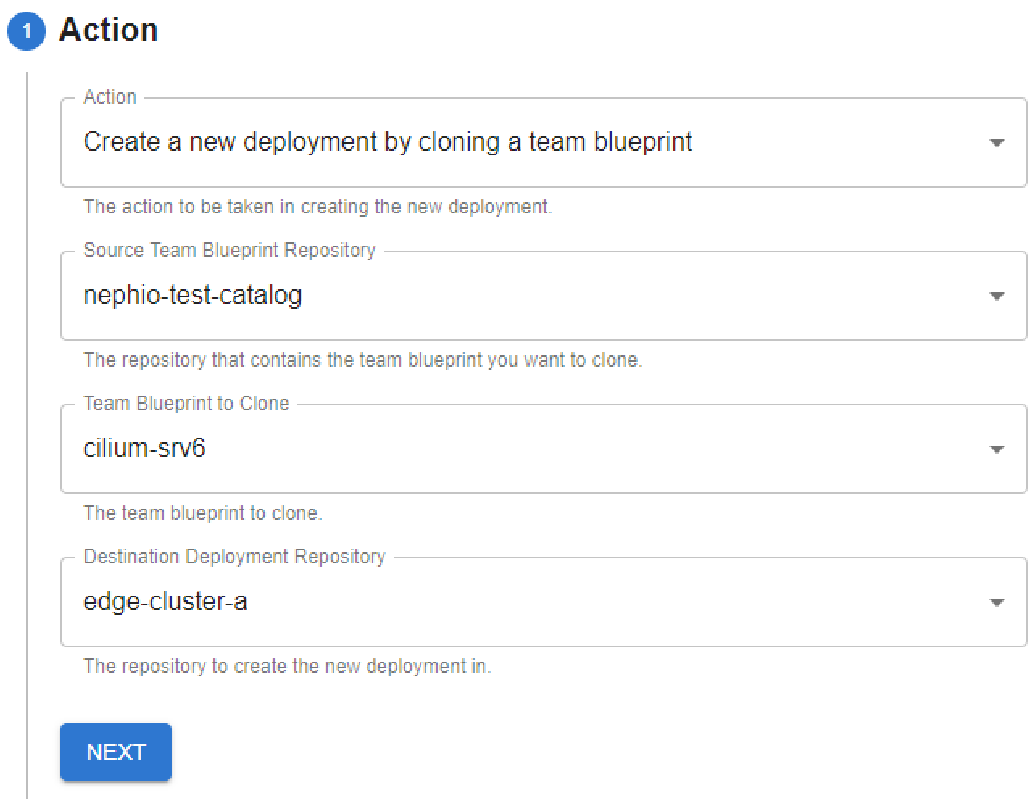


Team Blueprints

STATUS	REPOSITORY	NAME	REVISION	BLUEPRINT	CREATED
	nephio-packages	coredns-caching	v1		August 2, 2022
	nephio-packages	coredns-caching-scaled	v2	coredns-caching v1	November 16, 2022
	nephio-packages	nephio-configsync	v2		November 13, 2022
	nephio-packages	nephio-system	v6		November 15, 2022
	nephio-packages	nephio-webui	v7		May 30, 2023
	nephio-test-catalog	cilium-bgp	v0		April 4, 2023
	nephio-test-catalog	cilium-srv6	v1.2		June 14, 2023

Figure 3.11.: Nephio Team Blueprints

When adding a new deployment, the user is asked what blueprint should be used and where to deploy the package as in figure 3.12. The same menu contains the configuration of the metadata. The package must have a unique name for every deployment repository (cluster).



1 Action

Action: Create a new deployment by cloning a team blueprint

The action to be taken in creating the new deployment.

Source Team Blueprint Repository: nephio-test-catalog

The repository that contains the team blueprint you want to clone.

Team Blueprint to Clone: cilium-srv6

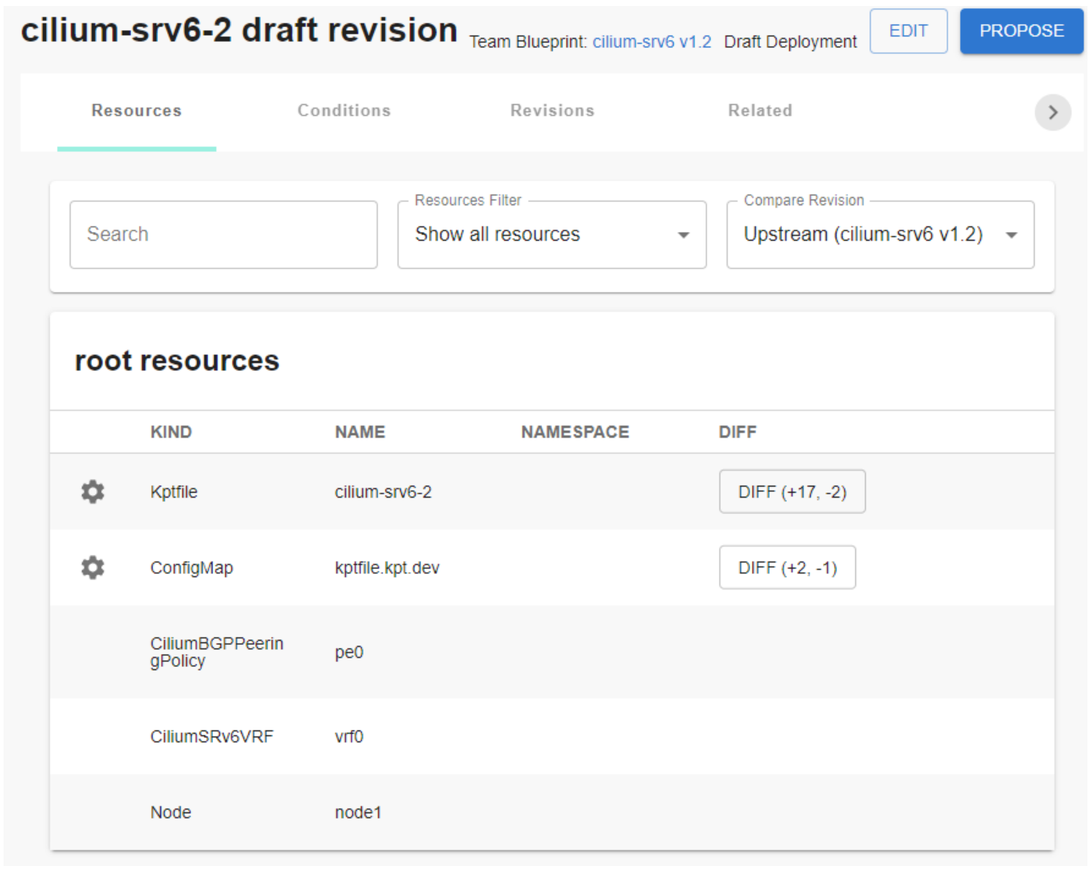
The team blueprint to clone.

Destination Deployment Repository: edge-cluster-a

The repository to create the new deployment in.

Figure 3.12.: Nephio Deployment Action

Nephio will fetch the blueprint configuration and show the draft revision. We must configure and propose the revision since both edge clusters require different configurations.


cilium-srv6-2 draft revision
Team Blueprint: cilium-srv6 v1.2 Draft Deployment
EDIT
PROPOSE
Resources
Conditions
Revisions
Related
Search

Resources Filter: Show all resources

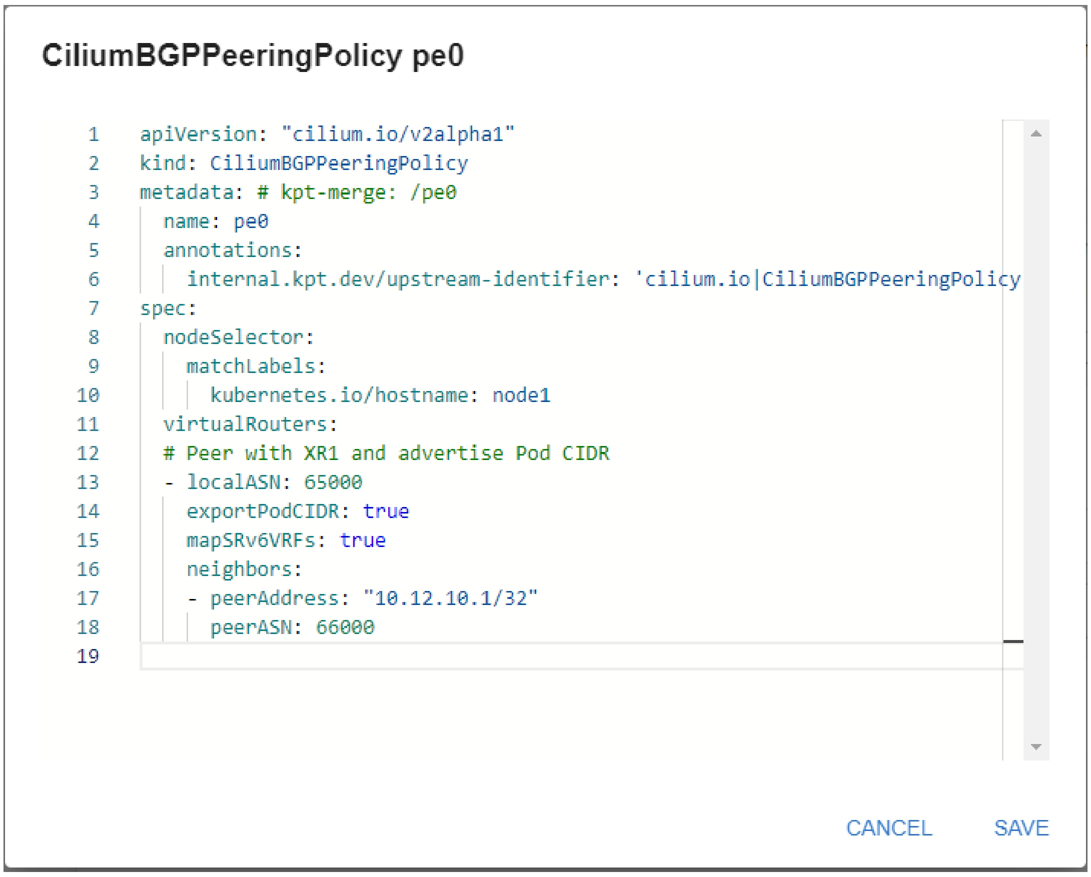
Compare Revision: Upstream (cilium-srv6 v1.2)

root resources

KIND	NAME	NAMESPACE	DIFF
Kptfile	cilium-srv6-2		DIFF (+17, -2)
ConfigMap	kptfile.kpt.dev		DIFF (+2, -1)
CiliumBGPPeeringPolicy	pe0		
CiliumSRv6VRF	vrf0		
Node	node1		

Figure 3.13.: Nephio Base Draft of Cilium SRv6 Deployment

The user can validate or change the resources with validator or mutator functions. The other two approaches are configuring a global ConfigMap or editing the resources directly. Figure 3.14 shows the modification of the CiliumBGPPeeringPolicy that will be applied to 'Edge-Cluster-A'. Once the deployment is ready, it can be proposed for inspection and then approved for provisioning. This procedure triggers the Porch API to copy to push the modified package to the deployment repository associated with the cluster. The Config Sync pod on the edge cluster will fetch the packages from the deployment repository and apply them to the actual state of the cluster.

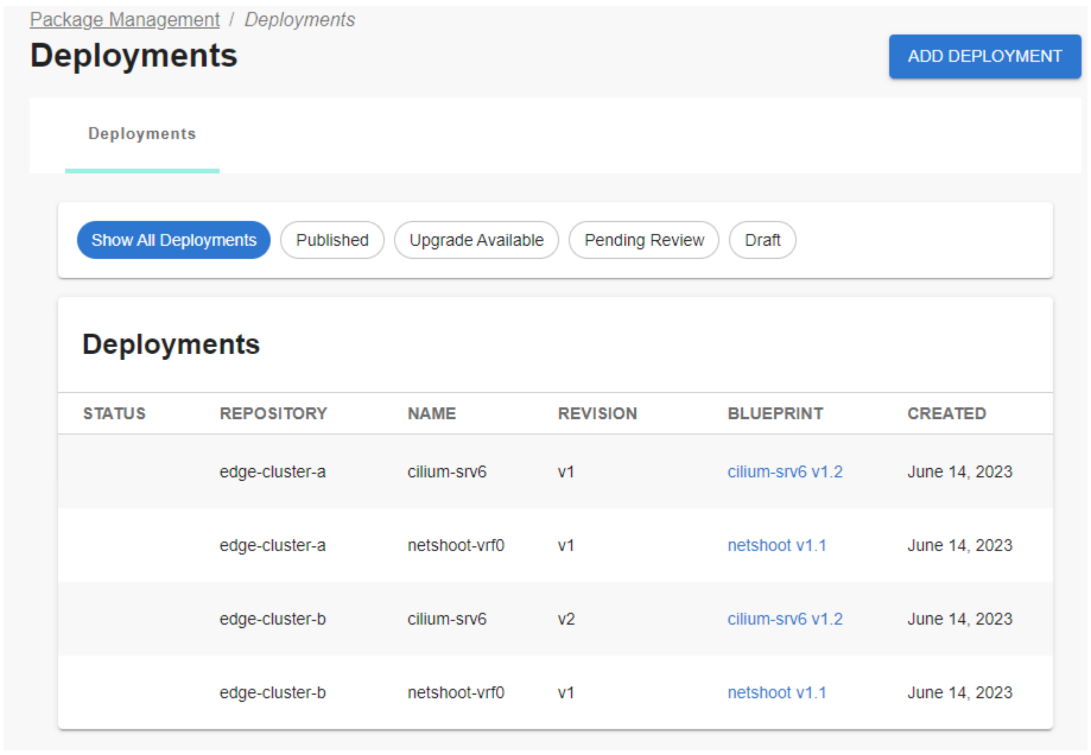

CiliumBGPPeeringPolicy pe0

```
apiVersion: "cilium.io/v2alpha1"
kind: CiliumBGPPeeringPolicy
metadata: # kpt-merge: /pe0
  name: pe0
  annotations:
    internal.kpt.dev/upstream-identifier: 'cilium.io|CiliumBGPPeeringPolicy
spec:
  nodeSelector:
    matchLabels:
      kubernetes.io/hostname: node1
  virtualRouters:
  # Peer with XR1 and advertise Pod CIDR
  - localASN: 65000
    exportPodCIDR: true
    mapSRv6VRFs: true
    neighbors:
    - peerAddress: "10.12.10.1/32"
      peerASN: 66000

```

Figure 3.14.: Nephio SRv6 Package Configuration

For each cluster, an SRv6 package was deployed, including a Netshoot pod to simulate a workload belonging to a VRF as seen figure 3.15.



Deployments

STATUS	REPOSITORY	NAME	REVISION	BLUEPRINT	CREATED
	edge-cluster-a	cilium-srv6	v1	cilium-srv6 v1.2	June 14, 2023
	edge-cluster-a	netshoot-vrf0	v1	netshoot v1.1	June 14, 2023
	edge-cluster-b	cilium-srv6	v2	cilium-srv6 v1.2	June 14, 2023
	edge-cluster-b	netshoot-vrf0	v1	netshoot v1.1	June 14, 2023

Figure 3.15.: Nephio All Deployments

The results of the deployments are described in chapter 4 Results.

3.9. Extending the Infrastructure to the Public Cloud

So far, we have only addressed the infrastructure in the LTB, where we deploy the clusters and workloads. The LTB simulates a local on-premise infrastructure that is usually limited in computing power and storage capacity. Many enterprises implement the hybrid cloud approach to scale their infrastructure. The common practice is to connect the on-premise infrastructure with a Virtual Private Cloud (further referred to as VPC) service at a public cloud provider by a VPN tunnel. All public cloud resources, such as VMs, routers, and firewalls, run in a VPC. A VPC contains subnets to isolate resources.

We also want to apply this practice, deploy clusters locally and in the public cloud, and use SRv6 for flexible routing and L3VPN. However, the infrastructure is based on an IPv6 backbone to route encapsulated packets for the L3VPN service. Some cloud providers do not support the IPv6 protocol in their virtual cloud instances. The biggest challenge is finding a VPN service capable of routing IPv6 packets to a VPC. Most cloud providers offer different VPN services; usually, the most expensive ones support IPv6. Another challenge is using public IP addresses to connect VPN tunnels to the cloud. The LTB has to support yet public IP addresses. Thus, site-to-site VPN is not possible. All cloud provider VPN services require a public IP address for site-to-site tunnels.

To deal with these challenges, we discovered another approach. In this thesis, we often used GCP to test deployments. The GCP marketplace offers an OpenVPN Access Server[14] deployment bundle that allows end-to-site VPN to the resources running in the VPC. One can choose the service from the marketplace and configure the server. The deployment creates a small-sized Ubuntu VM running OpenVPN Access Server. A host can configure the OpenVPN client to connect to the OpenVPN Access Server gateway over TLS and reach the cloud resources running in the same VPC.

A VM called 'CloudConnector' in the LTB is solely used to connect to the VPN and tunnel IPv4 and IPv6 traffic. All hosts in the LTB need to communicate with the Kubernetes nodes running on VMs in GCP. Thus, static routes must be configured on the router XR1 to ensure routing over the 'CloudConnector' to the public cloud instances. Figure 3.16 shows that the VPN tunnel extends the infrastructure to the cloud. The Kubernetes clusters establish BGP peerings with XR1 over the tunnel. Therefore, it is possible to advertise the VPNv4 and VPNv6 prefixes to the on-premise infrastructure and import them into the local Kubernetes clusters with Cilium. This hybrid cloud approach allows the use of SRv6 across the on-premise infrastructure and interconnects pods between separate infrastructures for scalability.

OpenVPN provides a guide[15] on how to configure IPv6 on Open VPN Access Server.

[14]OpenVPN Access Server [Inc23b]
[15]Limited IPv6 support built into the Access Server [Ser23]

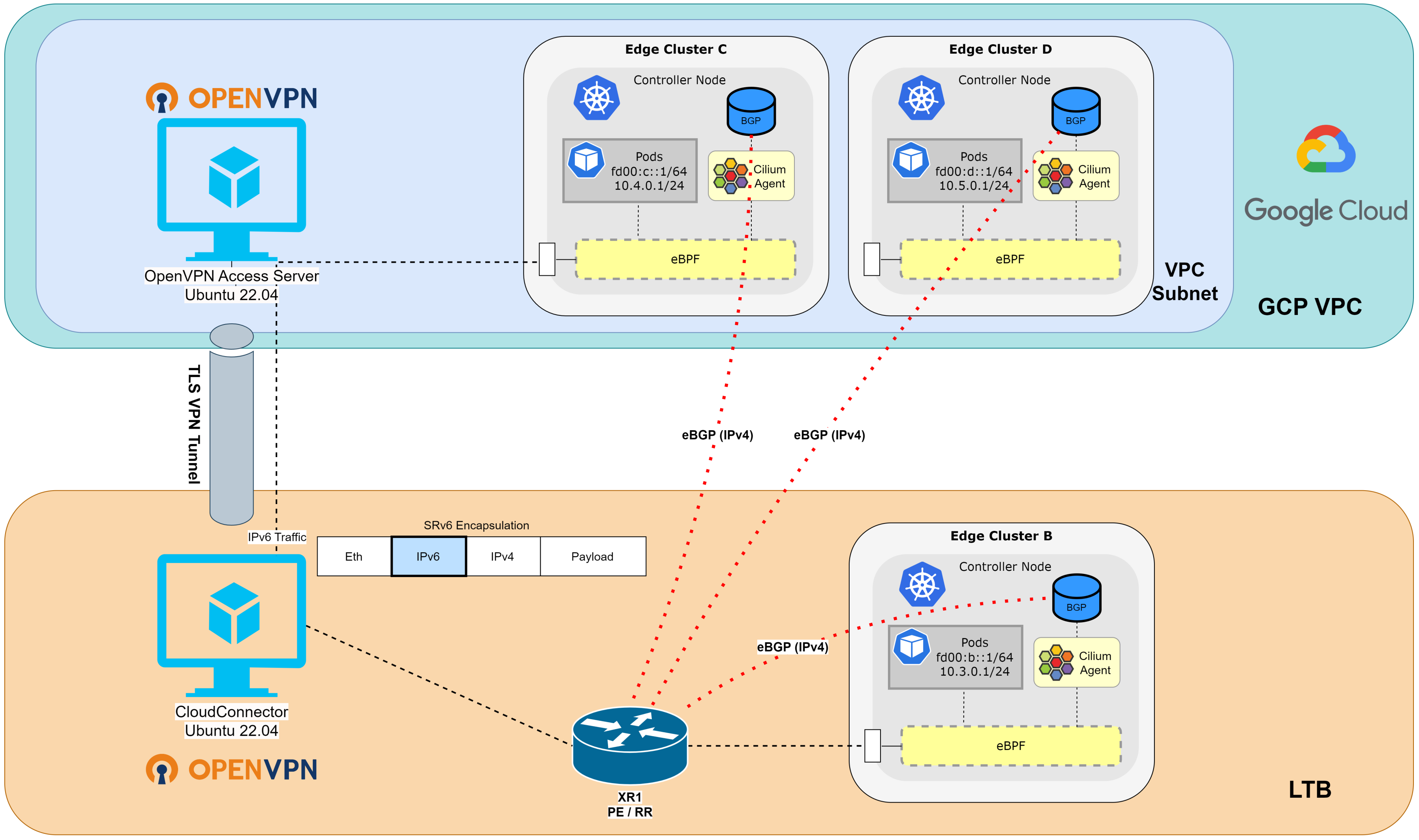


Figure 3.16.: Scaling Infrastructure to the Public Cloud

3.10. Deployment

The deployment is being made from within a GitLab pipeline. In this chapter, details regarding the pipeline are being discussed. As this bachelor thesis was developed as a PoC, the main focus was on getting things to work. This may be seen as the reason for implementing many manual steps within the GitLab pipeline.

3.10.1. Preparations

GitLab needs a configured runner with enough resources, as a slow runner may slow down the deployments remarkably. In addition, the GitLab runner needs to have SSH access to a bastion host or the nodes VMs directly.

A Dockerfile was written to create a container image. This container image consisted of a minimal Linux distribution 'Alpine'. The image was pushed to the GitLab container registry to be used within the GitLab pipeline.

```
FROM alpine:latest

RUN apk add \
--update \
--no-cache ansible && \
rm -rf /tmp/* /var/cache/apk
```

Listing 3.22: Dockerfile for Container Image

3.10.2. Pipeline

In this section, parts of the GitLab CI/CD pipeline are being described.

The pipeline consists of three different stages, each for a specific part of the cloud infrastructure deployment, e.g., Kubernetes, Cilium, and Nephio. To keep the deployment configuration files, of the three different deployments, in a single place because the locations would be too nested and need to be more straightforward. Therefore, single files need to be copied to their correct location, this is as some tools are quite strict about the file locations. Running jobs on specific GitLab runners require the tag 'ins-student' to be set. These runners have access to the internal network infrastructure of the INS and LTB.

The following pipeline snippet shows the uninstallation of Ansible as Kubespray has strict version requirements (see following snippet). Due to using Alpine Linux, most tools need to be added manually, as Alpine has nothing pre-installed. SSHpass, OpenSSH-Server, netaddr-tools, python3 and C interpreter must be installed.

```
stage: deployment_kubernetes
  - apk del ansible
  - apk add py3-pip
  - apk add python3-dev
  - apk add build-base
  - apk add sshpass
  - apk add openssh
  - apk add curl
  - apk add py3-jinja2
  - pip install netaddr
```

Listing 3.23: Software Dependencies for Ansible

As Kubespray throws errors when the working directorys permissions are too restrictive, the permissions of the folder are being altered with 'chmod 750'. This is non-security relevant, as the container will be scrapped after execution, and no data may be accessed. The by Kubespray required software is being installed by executing the txt file. The files are shipped with every release of Kubespray containing necessary packages with mostly up-to-date versions.

```
- chmod -R 750 kubernetes
- pip install -r kubernetes/requirements.txt
```

Listing 3.24: Install Software Dependencies for Ansible and Directory Permissions

As mentioned above, the configuration files need to be copied to the correct folder. This is done with multiple copy commands within the pipeline. The Kubernetes configuration file must be named according to the sample folder structure, which is done before setting the environment variables, ensuring the proper functionality of Ansible. It is mentioned in the section 'further work', that splitting the pipelines and using individual container images would be a clean approach. Kubespray has a specific folder structure which must be addressed by changing the current directory to the 'kubernetes' folder. As runing the 'ansible-playbook' command must be done within this folder.

```
- cp -r deployment_configuration/ansible.cfg kubernetes/
- cp -r deployment_configuration/inventory.ini kubernetes/inventory/bachelorthesis/
- cp -r deployment_configuration/kubernetes-cluster-config.yml
    kubernetes/inventory/bachelorthesis/group_vars/k8s_cluster/
- mv kubernetes/inventory/bachelorthesis/group_vars/k8s_cluster/kubernetes-cluster-config.yml
    kubernetes/inventory/bachelorthesis/group_vars/k8s_cluster/k8s-cluster.yml
- export PATH=/root/.local/bin:$PATH
- cd kubernetes
```

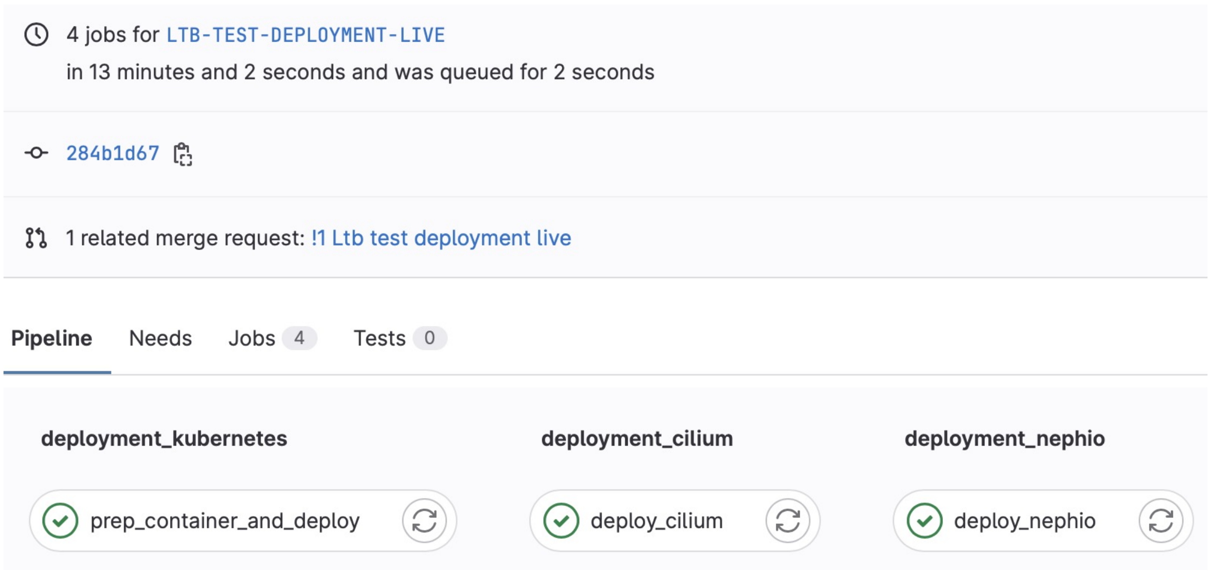
Listing 3.25: File Copying and Renaming within the Pipeline

The following statement defines when the job should be executed. This rule defines that the job only runs when the commit was made within the branch 'bathesis'.

```
- if: '$CI_COMMIT_BRANCH == "bathesis"'
```

Listing 3.26: Branch Exclusion

As the deployments of Cilium and Nephio are pretty similar to the Kubespray, these deployments are not described precisely.


4 jobs for LTB-TEST-DEPLOYMENT-LIVE
in 13 minutes and 2 seconds and was queued for 2 seconds

284b1d67

1 related merge request: !1 Ltb test deployment live

Figure 3.17.: All three Jobs of the GitLab pipeline Deployment

```
####################################################################
### GitLab Pipeline - Automated K8s + Cilium + Nepio Deployment ###
####################################################################

stages:
 - deployment_kubernetes
 - deployment_cilium
 - deployment_nephio

image: registry.gitlab.ost.ch:45023/ins-stud/sa-ba/ba-fs23-cnia/k8s-nodes/ba-ansible:v1

 prep_container_and_deploy:
 stage: deployment_kubernetes
 tags:
  - ins-student
 before_script:
   - apk del ansible
   - apk add py3-pip
   - apk add python3-dev
   - apk add build-base
   - apk add sshpass
   - apk add openssh
   - apk add curl
   - apk add py3-jinja2
   - pip install netaddr
   - chmod -R 750 kubernetes
   - pip install -r kubernetes/requirements.txt
   - cp -r deployment_configuration/ansible.cfg kubernetes/
   - cp -r deployment_configuration/inventory.ini kubernetes/inventory/bachelorthesis/
   - cp -r deployment_configuration/kubernetes-cluster-config.yml
       kubernetes/inventory/bachelorthesis/group_vars/k8s_cluster/
   - mv kubernetes/inventory/bachelorthesis/group_vars/k8s_cluster/kubernetes-cluster-config.yml
       kubernetes/inventory/bachelorthesis/group_vars/k8s_cluster/k8s-cluster.yml
   - export PATH=/root/.local/bin:$PATH
   - cd kubernetes
 script:
   - ansible-playbook -i inventory/bachelorthesis/inventory.ini --become --skip-tags=multus
       pb_main.yml
 rules:
   - if: '$CI_COMMIT_BRANCH == "bathesis"'

deploy_cilium:
 stage: deployment_cilium
 tags:
  - ins-student
 before_script:
   - apk add sshpass
   - apk add openssh
   - chmod 750 cilium
   - cp -r deployment_configuration/inventory.ini cilium/
   - cp -r deployment_configuration/cilium-node1.yml cilium/host_vars
   - mv cilium/host_vars/cilium-node1.yml cilium/host_vars/node1.yml
   - cp -r deployment_configuration/ansible.cfg cilium/
   - chmod 750 cilium
   - cd cilium
 script:
   - ansible-playbook -i inventory.ini --become playbooks/main.yml
 rules:
   - if: '$CI_COMMIT_BRANCH == "bathesis"'

deploy_nephio:
  stage: deployment_nephio
  tags:
    - ins-student
  before_script:
    - apk add sshpass
    - apk add openssh
    - cp -r deployment_configuration/inventory.ini nephio/
    - cp -r deployment_configuration/nephio-config-node1.yml nephio/host_vars/node1.yml
    - cp -r deployment_configuration/nephio-config-global.yml nephio/group_vars/all.yml
    - cp -r deployment_configuration/ansible.cfg nephio/
    - chmod -R 750 nephio
    - cd nephio
  script:
    - ansible-playbook -i inventory.ini --become playbooks/main.yml
  rules:
    - if: '$CI_COMMIT_BRANCH == "bathesis"'
```

Listing 3.27: Full Listing of GitLab Pipeline

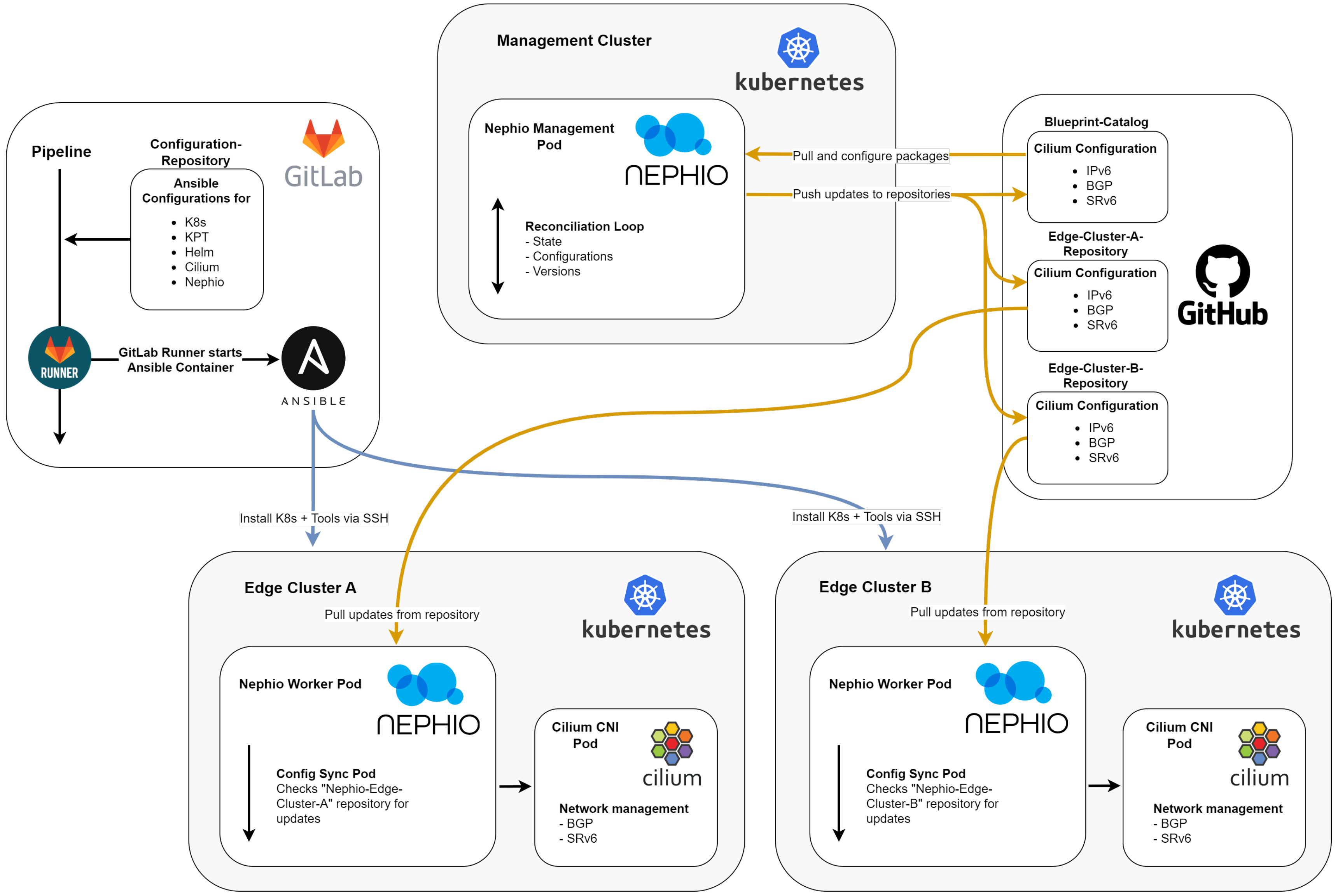


Figure 3.18.: Deployment Workflow with Kubernetes, Cilium and Nephio

CHAPTER 4

Results

4.1. Evaluation of Nephio and FluxCD

We have conducted a comprehensive evaluation of the tools Nephio and FluxCD as cloud-native automation tools to address the challenges in managing a large fleet of Kubernetes clusters and containerized applications on it.

Nephio currently offers a PoC version for simple KPT package orchestration but still needs the ability to spin up clusters. While FluxCD offers a current solution to do so, it is not available as an open-source project.
We have found Nephio to be the suitable tool to solve this challenge, as it was developed with the intent to manage thousands of 5G remote sites and their network functions, what also applies to regular applications in distributed Kubernetes environments.
The key point is the process regarding source integration. Nephio and FluxCD use the concept of GitOps and Single-Source-of-Truth to store their artifacts. Registering new sources and artifacts in FluxCD requires copying folders, creating new repositories, configuring source paths for every application on the cluster, and perform merge requests for configuration changes.
Nephio, replaces this Git workflow by choosing and modifying blueprints from a catalog repository and pushing them to multiple deployment repositories. The Config Sync pods on the edge clusters do not require a specific source like in FluxCD. They fetch all resources in their associated deployment repository and compare them with the actual state of the cluster. If Config Sync detects any drifts, it will apply the desired state. This procedure scales well because the entire intelligence relies on the centralized management cluster, and the worker clusters need little to no configuration.

One drawback of Nephio is its installation procedure and unstable components. We had difficulties figuring out how to get it running; its components required many resources of the underlying VM.

Nevertheless, we could quickly deploy network configurations to edge clusters, and the automatic Nephio integration of edge clusters through our Kubespray modification enabled us a quick setup of a cloud-native infrastructure.

4.2. Provisioning of Kubernetes and Nephio

Provisioning Kubernetes started with having difficulties with our self-engineered Kubernetes deployment. Challenges arose due to running low on resources, having multiple interfaces, and inexplicable behavior with non-responding services were points to tackled among others. The team spent much time researching and troubleshooting the situation. Most challenges were not unknown to the community, as most issues have been discussed online already. The team had to decide what to do, as most solutions were practicable, but there were better ways to go.

The team agreed to switch to Kubespray. An efficient way of deploying production-ready Kubernetes clusters was found, customizable to deploy from multiple node clusters to different CNI plugins. Nevertheless, the team also faced some further challenges with Kubespray. Having multiple interfaces attached to the VM was also a challenge with Kubespray. It was agreed to deploy single node clusters within the LTB, enabling us to run and test the infrastructure needed for other use cases. Running the deployments in public clouds like GCP and DigitalOcean, the deployment was also successful with a multi-cluster setup. Being based on Ansible, the solution embedded well into the GitLab pipelines with the following Ansible based deployments of the CNI plugin Cilium and Nephio. Eventually, a capable solution was found and adapted, and the Kubernetes deployment was accordingly successfully completed.

The GitLab pipeline successfully deploys the necessary steps to run Nephio like creating the edge cluster repositories on GitHub, registering this repository on the management-cluster, and installing KPT and Nephio Config Sync onto the edge cluster. After the deployment is finished, the created edge cluster can be managed from the management cluster either through the local KPT CLI or the Nephio web UI. The web UI efficiently modifies blueprints and deploys them as workloads to registered edge clusters.

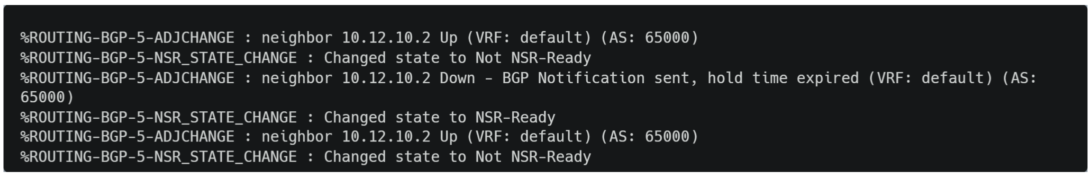
4.3. Provisioning of Cilium CNI Plugin

Our GitLab CI/CD pipeline successfully installs Helm and deploys the Cilium CNI plugin Helm chart with the parameters defined in the YAML configuration file. In addition, it installs the Cilium CLI tool, which we used to verify the status of Cilium. A few configuration values must be changed for every deployment because the hosts have different IP addresses and pod prefixes.

4.4. Deployment of Cilium BGP and SRv6 Workloads for L3VPN

The deployment of Cilium configuration resources and Netshoot daemon sets is successful, and all resources are created on the destination cluster. However, L3VPN does currently not work in the LTB. When the 'CilliumBGPPeeringPolicy' is deployed, Cilium tries to establish a peering with the XRd router, which may take minutes. After successfully establishing a BGP peering, the router will close the connection after 90 seconds and log that the hold timer has expired. The hold timer expires when a peer does not receive a BGP keepalive message for a negotiated interval. Cisco XR has a default hold timer of 180 seconds, while Cilium uses the open-source BGP implementation goBGP and a hold timer of 90 seconds. The lower one is negotiated and used on both peers. The interval for the keepalive messages is usually a third of the hold timer. This means both peers expect a keepalive message every 30 seconds from each other. If a peer does not receive a keepalive message in 90 seconds, it will declare its neighbor dead, close the peering, and send a BGP notification message to its neighbor informing that the hold timer expired.

When we experienced this for the first time, we thought the router was misconfigured. After analyzing the link between the peers with Wireshark, we saw a flooding of TCP Retransmission and some TCP DUP ACK packets sent by both sides. All packets used TCP port number 179, which is BGP. We investigated the Kubernetes cluster with Cilium but found no misconfiguration. We had a suspicion about the Kubernetes cluster and tried a standalone BGP software. goBGP was installed on another VM connected to the same Cisco XRd Router. The BGP peering was configured similarly to the Cilium configuration, but the TCP Retransmissions appeared again. Next, the tool IPerf was installed to simulate TCP traffic between the Kubernetes cluster and the additional VM, where the packets had to go through the Cisco XRd router. We could observe the same behavior, even without BGP. Additional troubleshooting included setting the same MTU size and collecting Linux logs without success.



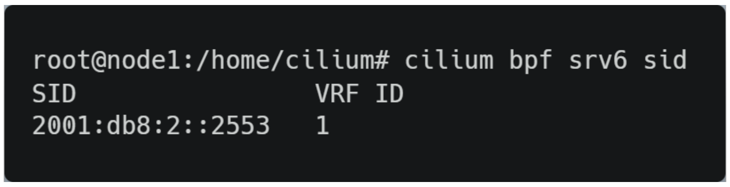
```
%ROUTING-BGP-5-ADJCHANGE : neighbor 10.12.10.2 Up (VRF: default) (AS: 65000)
%ROUTING-BGP-5-NSR_STATE_CHANGE : Changed state to Not NSR-Ready
%ROUTING-BGP-5-ADJCHANGE : neighbor 10.12.10.2 Down - BGP Notification sent, hold time expired (VRF: default) (AS:
65000)
%ROUTING-BGP-5-NSR_STATE_CHANGE : Changed state to NSR-Ready
%ROUTING-BGP-5-ADJCHANGE : neighbor 10.12.10.2 Up (VRF: default) (AS: 65000)
%ROUTING-BGP-5-NSR_STATE_CHANGE : Changed state to Not NSR-Ready
```

Figure 4.1.: BGP Holder Timer Expired Logs

When deploying a smaller lab, we could not observe the same behavior. The smaller lab was only temporary, and right after deploying a new one with only one router, the TCP transmission appeared again. The behavior could only be replicated sometimes. After losing time with the re-configuration, we focused on the Cilium configuration of L3VPN in a local lab and the cloud. The advisor was informed about this decision.

We use the virtual routing software FRRouting, like in the demo lab that was provided to us. It is open-source and runs on Ubuntu 22.04. FRRouting and Cilium SRv6 successfully provide a L3VPN on pod basis. We will either use the local or cloud setup for demo purposes.

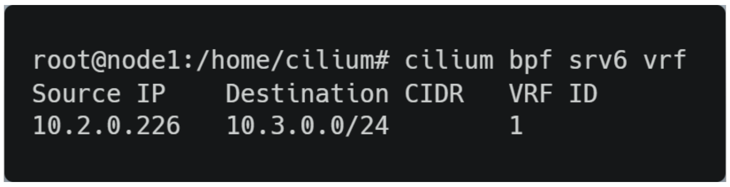
The state of Cilium SRv6 can be checked by connecting to the Cilium agent and entering a set of commands. The command in figure 4.2 shows the SID Cilium has generated to advertise its L3VPN reachability. This SID is used by other Kubernetes clusters or routers to reach the pod network over VPNv4.



```
root@node1:/home/cilium# cilium bpf srv6 sid
SID                 VRF ID
2001:db8:2::2553    1
```

Figure 4.2.: Cilium SRv6 SID

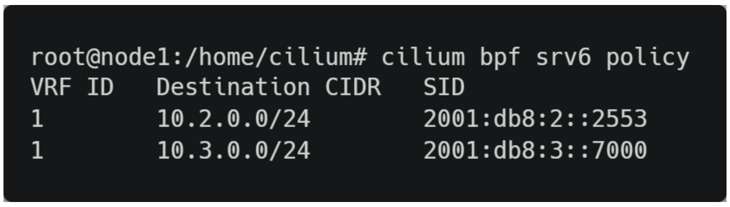
Figure 4.3 shows the VRF and destination CIDR mapping to the IP address of the Netshoot pod. The pod is allowed to route to the destination CIDR in VRF 1.



```
root@node1:/home/cilium# cilium bpf srv6 vrf
Source IP    Destination CIDR   VRF ID
10.2.0.226   10.3.0.0/24        1
```

Figure 4.3.: Cilium SRv6 VRF

Figure 4.4 shows that Cilium received network reachability information from the other Kubernetes cluster through the router over BGP VPNv4 peering. Cilium creates an egress policy, and the prefix is injected into VRF with the SID to be used to reach a pod in the other cluster. The figure contains two egress policies because Cilium also receives the network reachability information, which is advertised to the router.



```
root@node1:/home/cilium# cilium bpf srv6 policy
VRF ID   Destination CIDR   SID
1        10.2.0.0/24        2001:db8:2::2553
1        10.3.0.0/24        2001:db8:3::7000
```

Figure 4.4.: Cilium SRv6 Egress Policy

The egress policy is responsible for the encapsulation. When the Netshoot pods sends traffic to a pod in the destination CIDR, Cilium will encapsulate it with the SID, which was advertised together with the prefix. The destination node will decapsulate it and forward it to the pod in the associated VRF. Cilium only supports a single SID in packets. This allows a VPN tunnel between pods in Kubernetes clusters. The peer does not have to be a Kubernetes cluster as well and could be a CE connected to the PE.

4.5. Extending the Infrastructure to the Public Cloud

The on-premise infrastructure (LTB) was successfully connected to the VPC with the subnets where the Kubernetes clusters reside. This extension was possible due to OpenVPN's Access Server that supports end-to-site VPN with IPv6 traffic inside the tunnel. Despite the challenges in the LTB regarding TCP retransmission, this concept was proven to be feasible. This concept extends the SRv6 domain to the public cloud, allowing more flexibility and scalability when routing applications.

CHAPTER 5

Discussion

5.1. Research Question and Findings

The primary goal of this thesis was to research Nephio and create a PoC to deploy network configurations. We started to compare Nephio with FluxCD along the way and later created an infrastructure to automatically deploy Nephio onto new clusters. In use case 1, 'Provisioning of Kubernetes and Nephio', a Kubespray deployment was written to fully automatize the deployment of new Kubernetes clusters. To deploy Nephio on newly initiated clusters, an Ansible script was written to create a new GitHub repository and deploy Nephio Config Sync on the created edge cluster. For use cases 2 and 3, another Ansible deployment was written to install Cilium onto the edge clusters and add the BGP and SRv6 workloads. These Cilium workloads could later be distributed using Nephio. The deployment of Kubernetes, Nephio and Cilium was tested on Google Cloud and in LTB.

5.2. Limitations

When writing this thesis, Nephio was only available as a PoC. The full version release of Nephio, scheduled for May 2023, has been delayed to the third quarter of 2023 during the thesis. Some functions regarding Nephio, like the web UI, required manual configurations as the official documentation did not work as intended. In addition to that, we could only deploy workloads to single clusters while the full version supports multiple deployments at a time using Kubernetes selectors.

5.3. Recommendation

The deployment of Kubernetes with Kubespray minimizes much of the manual labor required and should be considered by operators working with large fleet clusters and ISPs with extensive 5G networks. Nephio in its PoC state looks promising for future workloads deployments, especially for 5G network functions. The ability to quickly deploy specific network functions like Cilium BGP and SRv6 makes it interesting to orchestrate high amounts of clusters. As promising as Nephio looks, it still needs some time to reach its maturity state. The segment routing technology SRv6 will play a significant role when using flexible routing and optimizing traffic between Kubernetes clusters. While already being established in operating systems for network devices, SRv6 is slowly adapted to Kubernetes. We recommend operators of large Kubernetes fleets to follow the current integration of SRv6 into Kubernetes by various CNI plugins such as Cilium.

5.4. Further Work

As with most engineering topics in computer science, there are different ideas for developing new or better features. As this thesis was conducted as a PoC, the focus was on getting the assigned tasks to work. Some solutions have yet to be engineered with 'beauty' in mind and not just functionality. On the other hand, meaningful work in relevant subjects, such as security, should be considered.
Some of the following ideas for further development have been (partially) engineered and tested but have yet to be added to updated versions. Features were not added because of the maturity of the solutions, as this thesis is a PoC with other goals. The ideas are described from an objective viewpoint with no prioritization. This list is not completed, and other ideas are possible.

5.4.1. Security

In this section, ideas to improve the security level are described. No negligent security levels were used during the project. Nevertheless, security has been, especially during recent years, an ever-growing concern and needs to be addressed with appropriate measures and countermeasures.

Bastion Host

In network environments, especially with network providers, segmentation of networks and even single hosts has recently been adopted. Establishing boundaries between servers, clients, and devices increases security substantially, as traffic can only pass via firewalls. Strong security means that VMs can not be accessed from WAN zones. Therefore there needs to be a 'friendly' proxy. A so-called 'bastion host' VM within a special DMZ, accessible from specific ranges outside the network. This machine can connect to the cluster nodes and replays the commands via SSH. Perimeter security can be persisted using bastion hosts

Public Private Key

Establishing public-private-key authentication and encryption is a best practice in productive environments. This requires secure handling and storing of these keys.

Ansible Vault

Ansible Vault is a well-known official feature. Secrets such as credentials, tokens, and classified information stored in a YAML are AES256 encrypted. The encrypted vault file is specified by entering the run command for Ansible playbooks, and the password needs to be entered. When running a playbook with Ansible Vault within a GitLab pipeline, the password needs to be extracted from a GitLab secret into a .txt file within the container working directory. The password is stored securely as the containers are being dropped after exiting the pipeline job. The copy job had to be placed within the pipeline, as the GitLab secrets are accessible. This feature has been partially implemented. The feature was not implemented in the PoC as the secrets are stored within the GitLab instance. Unfortunately, Kubespray did not work with storing usernames within these secure locations.

On Host Secrets Security

Credentials and tokens distributed to the nodes during deployment may be protected better. As in typical environments, Kubernetes nodes do not run modern security tools, such as XDR software, which would allow in-depth multi-level security. Specific software should be used to protect keys and tokens. Bitnami256[1] is a tool that brings security for secrets at rest and in transit. This tool is established in the industry and was recommended for investigation by Philip Schmid.

5.4.2. New features

Ideas for new features and ideas are written down within this chapter.

[1]GitHub Bitnami Secrets [Lab23]

Integration of IaC Concepts

IaC is a trend that has gained traction with the adaption of Terraform, an innovative platform with APIs for public cloud service providers. The thesis is deploying Kubernetes, Cilium, and Nephio on existing VMs. Infrastructure may be described within configuration files and deployed before the Kubernetes deployment. Using Kubernetes services from the cloud provider may be interesting, as the integration with Cilium and Nephio may improve in the future.

Extraction of Kubeconfig Files

After deploying a Kubernetes cluster, the kube config files may be extracted with Ansible allowing for remote access and management using Kubectl.

Improvements of Existing Work

Configuration files and pipelines may be restructured, and unnecessary items may be deleted. This would increase the ambiguity of the configuration files.

Pipeline within Pipeline

GitLab pipelines allow only one image to be used. Usually, this is no problem as only one task, 'of a kind', is being deployed. Using Ansible for deploying Nephio, Cilium, and Kubespray with strict software version restrictions may be challenging to deal with. Running pipelines within a pipeline enables the use of different container images.

Parallel Deployment

Running multiple deployment jobs may be achieved by initializing them one by one or by using pipelines in the pipeline. The advantage of running pipelines in pipelines is that jobs run parallel and, therefore, are more efficient and punctual. These types of pipelines are called 'parent-child pipelines'[2].

5.4.3. Preparing for Production Readiness

Before bringing the development to a minimal stage ready for production, some enhancements must be made.

Testing Kubespray Scaling with Cilium and Nephio

As Kubespray can scale in and out, the behavior of Cilium and Nephio should be tested accurately. Removing, updating, or changing the control plane nodes has yet to be tested, but the possibility of success regarding Cilium and Nephio seems relatively low.

Remove Interfaces for Deployment

As the deployments have shown, initializing Kubernetes clusters with multiple interfaces may be challenging. Removing interfaces before deployments and binding them later is expected to be successful.

Generate Report

Ansible is set to gather facts while executing Kubespray deployment jobs. Artifacts and data describing the state and the internal inventory of the nodes could be automatically exported into a JSON. These files and information could be used to ensure quality standards and prove the achievement of internal compliance.

[2]GitLab Docs[Inc23a]

Configure Cisco IOS XR with Ansible

As our optional 'Provisioning of L3VPN Configurations on Provider Routers' use case showed, automatic Cisco IOS XR router configuration was generally possible but did not work with the intended Ansible modules. We must first fix the error with the Ansible module and our devices to use this method in production. With a working setup, deploying Cisco configurations would be much faster and reproducible.

Fix TCP Retransmissions in LTB

The INS is keen on the SRv6 technology, but due to the TCP retransmissions that break up the BGP peering in the LTB, they must currently use another virtualization platform to use Cilium SRv6. LTB is developed by INS and used for practical lab exercises where students can learn to operate network devices such as more comprehensive routers. Additional work could be addressing and solving this issue in cooperation with the developers of the LTB platform.

CHAPTER 6

Challenges and Decisions

In this part, all relevant technical challenges and decisions are documented. Some decisions were based of challenges that occurred during the thesis, while others where planned from the beginning. The why and how of the discussion, if necessary, will be addressed in the according sections.

6.1. IS-IS as IGP

Date: 10.03.2023
Decided by: Group
Brief description: Decision why Intermediate System to Intermediate System (further referred to as IS-IS) was used as Interior Gateway Protocol (further referred to as IGP). The decision was based on the findings in the student research project of the previous semester and is added for completeness.[1]

- Cisco documentation for SRv6 with Open Shortest Path First (further referred to as OSPF) did not work with the given router operating system (IOS XR)
  - Manual: Configuring SRv6 with OSPFv3 Protocol2[2]
  - Cisco Version: IOS XR Configuration 7.5.1
- SRv6 exercise in module "Software Defined Networks" at OST was done with IS-IS.
- The main reason is the use of IS-IS Type Length Value (further referred to as TLV) - allowing to extend header for SRv6 information which OSPF does not support.
- IS-IS sees more usage with big internet service providers.
- IS-IS is less Central Processing Unit (further referred to as CPU) intensive compared to OSPF.
- IS-IS convergences faster on default timers.
- IS-IS runs on layer 2 and is more secure compared to OSPF which runs on layer 3 (IP spoofing attacks).
- Update 29.05.2023: IS-IS is not needed anymore in the core due to the downsizing of one router (see technical decision "Downsizing Backbone Infrastructure").

[1]Student Research Project: Cilium CNI plugin Plugins for Kubernetes Cluster Networking [SDE22]
[2]2PDF Document "Configuring SRv6" from cisco [Cis23]

6.2. Network Infrastructure

Date: 10.03.2023
Decided by: Group
Brief description: A network plan containing two PE routers, one Provider Router (further referred to as P) router, one CE router, and five virtual machines was created to configure the infrastructure in the LTB.

- In the context of network infrastructure planning, facing the need to create a network plan for the creation of the infrastructure in the LTB, we decided to keep the provider core as simple as possible with three routers to focus on realizing the use cases. We neglected the more comprehensive network infrastructure design, accepting that we would have less flexibility when routing packets in the core network.

6.3. Additional CE

Date: 10.03.2023
Decided by: Group
Brief description: For testing purposes, an additional CE running Cisco IOS XE is connected to a PE.

- In the context of network infrastructure planning, facing the need to test and validate SRv6 workloads for L3VPN on Kubernetes, we decided to connect a single CE to a PE to easily validate single L3VPN configurations of a Kubernetes cluster instead of having to setup up two clusters before being able to test and validate L3VPN. We neglected to leave out the CE, accepting that we would have an additional router to configure and manage.
- Update 29.05.2023: The CE router is not needed anymore and is removed from the topology.

6.4. Nephio as Continuous Delivery Tool

Date: 10.03.2023
Decided by: Advisors
Brief description: Kubernetes workloads are deployed using Nephio instead of the graduated tool FluxCD.

- In the context of the evaluation of the continuous delivery tools Nephio and FluxCD, facing the need to choose a tool for the implementation of the use cases, the advisors decided to use Nephio instead of FluxCD as the initial stakeholders demanded it and as it is more interesting for future use. We neglected the use of the Cloud Native Computing Foundation graduated and established tool FluxCD, accepting that we would have to use the incomplete PoC version of Nephio and wait for an official release until May 2023.
- Update 19.05.2023: The official release of Nephio was postponed to mid 2023.

6.5. Using GitHub instead of GitLab for Nephio Deployments

Date: 24.04.2023
Decided by: Group
Brief description: Nephio was unsuitable for running with GitLab.

- Because OST provided us with GitLab infrastructure, the idea was to only use GitLab repositories to keep the tasks simple. On the Nephio PoC GitHub site, it was written that Nephio will only work with GitHub and Google Cloud Source Repository services. *"Nephio can work with repositories in GitHub or the Google Cloud Source Repository service. This example will use GitHub."*[3]. Nevertheless, we still wanted to try if it is possible to run 'nephio-system' on the management cluster with GitLab repositories. Unfortunately, we were not able to start the 'nephio-system' on the management cluster with GitLab repositories, and therefore we used GitHub repositories in further work containing Nephio. Our attempts

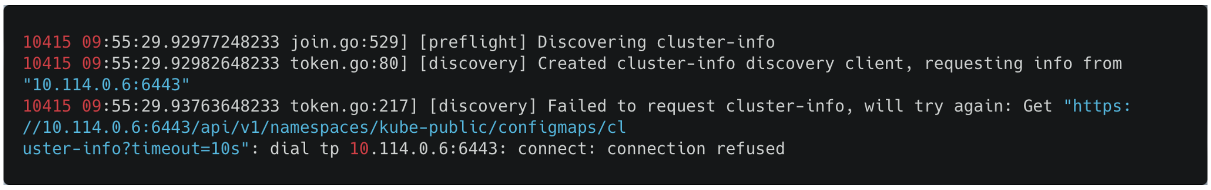
[3]GitHub 'nephio-poc' [Nep23b]

to run 'nephio-system' on the management cluster with GitLab repositories are documented in the chapter installation manuals A.

6.6. Kubernetes Deployment Technology Switch

Date: 24.04.2023
Decided by: Group
Brief description: Switch from self-developed Kubernetes Ansible-backed deployment to Kubespray.

- During the deployment of self-engineered Kubernetes deployments, multiple problems occurred. The problems were discussed and troubleshooted within the group and, at a later stage, discussed with our advisor. Errors with errors within certificates and connection problems while joining. Most things were fixed during the development, but not everything (see chapter "Methods"). These symptoms occurred not only on LTB infrastructure but also on the public cloud deployments.
- As the project schedule was dependent, working Kubernetes development was a basis, the group decided to halt their own Kubernetes development in favor of Kubespray.
- Some minor tests with Kubespray were conducted before the decision to use Kubespray.



```
10415 09:55:29.92977248233 join.go:529] [preflight] Discovering cluster-info
10415 09:55:29.92982648233 token.go:80] [discovery] Created cluster-info discovery client, requesting info from
"10.114.0.6:6443"
10415 09:55:29.93763648233 token.go:217] [discovery] Failed to request cluster-info, will try again: Get "https:
//10.114.0.6:6443/api/v1/namespaces/kube-public/configmaps/cl
uster-info?timeout=10s": dial tp 10.114.0.6:6443: connect: connection refused
```

Figure 6.1.: Kubernetes Connection Joining Problem

6.7. Google Cloud for Deployment Testings

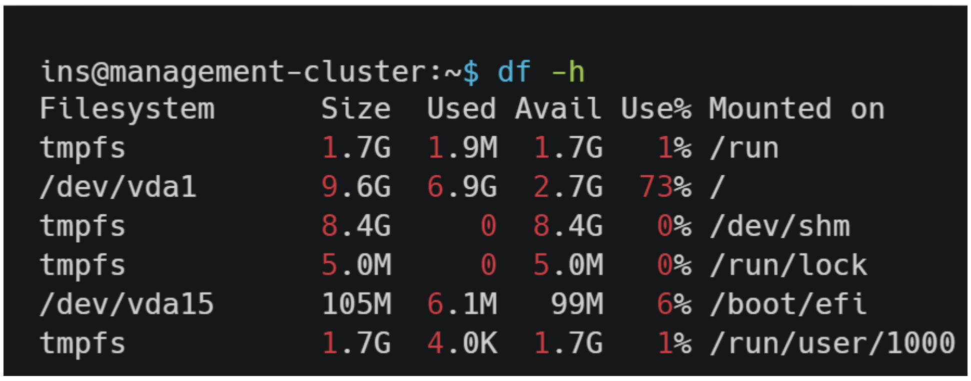
Date: 26.04.2023
Decided by: Group
Brief description: Google Cloud Compute Engines were used to test the deployments.

- Because at the start of the thesis, we only had one LTB Lab and did not want to ruin our freshly installed VM with a faulty deployment, we extensively tested the deployments with Google Cloud Compute Engines. In the first few weeks, we even used Kamatera[4] for testing but later switched to Google Cloud because of the generous testing period. All cloud environments were activated with our private accounts, and we made use of the free testing periods. Especially Google Cloud made it much easier to create new compute engines after faulty deployments and test different equipped machines. Without Google Cloud, we would have had to manually delete installation packages and files on the LTB machines and ensure everything was as new as on a fresh installation. Furthermore, a simple redeployment of a single VM in LTB was neither possible. Luckily our Google Cloud testing periods almost lasted till the end of our thesis and were very helpful.

6.8. New Management Cluster outside of LTB

Date: 19.05.2023
Decided by: Group and Advisors
Brief description: Storage size of management cluster needed to be extended.
The initial disk storage size of our VM in LTB were only 10 GB.

[4]Kamatera Website [Kam23]



```
ins@management-cluster:~$ df -h
Filesystem      Size  Used Avail Use% Mounted on
tmpfs           1.7G  1.9M  1.7G   1% /run
/dev/vda1       9.6G  6.9G  2.7G  73% /
tmpfs           8.4G     0  8.4G   0% /dev/shm
tmpfs           5.0M     0  5.0M   0% /run/lock
/dev/vda15      105M  6.1M   99M   6% /boot/efi
tmpfs           1.7G  4.0K  1.7G   1% /run/user/1000
```

Figure 6.2.: Management Cluster Disk Size

- After installing the necessary tools like Kubernetes, Cilium and Nephio free disk size was around 10 %. This resulted in Kubernetes trying to shut down multiple pods and trying to free up resources. During this cleanup process, almost 100 pods ended in the pod state 'Evicted' and did not function properly. After troubleshooting with Marco Martinez and Yannick Zwicker, we deployed a new VM outside of our LTB lab to function as the management cluster. If we wanted to increase the resources on the existing VM, we had to redeploy the whole LTB lab completely. We would have had to reconfigure all Cisco routers and the existing machines. Because the management cluster only communicates via the provided repositories with the edge clusters, it did not need to be placed in the LTB lab. To cause as little interruption to the thesis as possible, Yannick Zwicker deployed a new Hyper-V based VM on the 19. May 2023.

6.9. Increased Resources of LTB VMs

Date: Mon. 25.05.2023
Decided by: Group and Advisors
Brief description: Due to deployment difficulties, the resources of all Ubuntu VM were increased.

- The original devices had 2 Cores CPU, 2 GB RAM, and 10 GB hard disk. These defined resources were too small to be suitable for our Kubernetes deployments with Kubespray. We run into strange problems with pods ending in the state 'Evicted' and the deployment of Kubernetes, Cilium and Nephio taking around 30 to 40 minutes. After resizing the virtual machines in edge cluster A, edge cluster B, and management cluster to 4 Cores CPU, 4 GB RAM, and 50 GB hard disk, the deployments only took about 10 to 15 minutes, and the pods did not change to state 'Evicted'.

6.10. Single Node Clusters

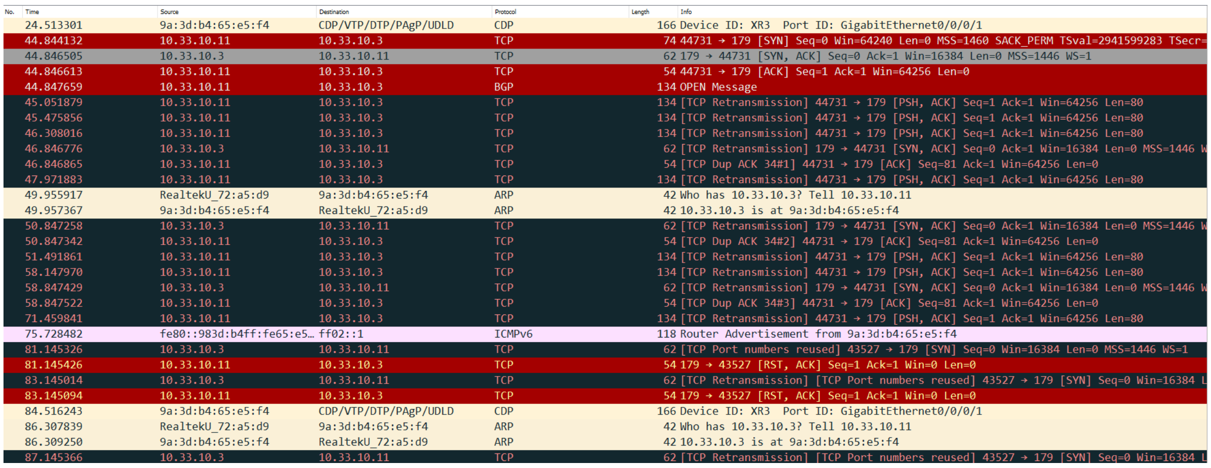
Date: 26.05.2023
Decided by: Group (and Advisor)
Brief description: Decision to deploy only single-node clusters to the LTB.

- Usually, when deploying Kubernetes, multi-node cluster are deployed. As the VM within the LTB has multiple networking interfaces attached, the binding to join and communicate between the nodes did not work with the proposed flags (see meeting minutes and section "Methods"). Even though Kubespray offers a possibility to specify the interface to establish the binding by defining the IP address of the interface within the inventory, the cluster-join command did not work. After many hours of debugging, the group decided to use single-node clusters, which would no longer block the project schedule.

- The group already deployed single-node clusters before the meeting, as this was the only appropriate way to have clusters up and running. We discussed this issue at the weekly meeting with our supervisor and advisor.

6.11. Downsizing Backbone Infrastructure

Date: 05.06.2023
Decided by: Group
Brief description: The backbone was downsized to only one router due to TCP retransmissions.



```
No. Time       Source                  Destination            Protocol  Length Info
    24.513301  9a:3d:b4:65:e5:f4       CDP/VTP/DTP/PAgP/UDLD  CDP          166 Device ID: XR3  Port ID: GigabitEthernet0/0/0/1
    44.844132  10.33.10.11             10.33.10.3             TCP           74 44731 → 179 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=2941599283 TSecr=
    44.846505  10.33.10.3              10.33.10.11            TCP           62 179 → 44731 [SYN, ACK] Seq=0 Ack=1 Win=16384 Len=0 MSS=1446 WS=1
    44.846613  10.33.10.11             10.33.10.3             TCP           54 44731 → 179 [ACK] Seq=1 Ack=1 Win=64256 Len=0
    44.847659  10.33.10.11             10.33.10.3             BGP          134 OPEN Message
    45.051879  10.33.10.11             10.33.10.3             TCP          134 [TCP Retransmission] 44731 → 179 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=80
    45.475856  10.33.10.11             10.33.10.3             TCP          134 [TCP Retransmission] 44731 → 179 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=80
    46.308016  10.33.10.11             10.33.10.3             TCP          134 [TCP Retransmission] 44731 → 179 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=80
    46.846776  10.33.10.3              10.33.10.11            TCP           62 [TCP Retransmission] 179 → 44731 [SYN, ACK] Seq=0 Ack=1 Win=16384 Len=0 MSS=1446 W
    46.846865  10.33.10.11             10.33.10.3             TCP           54 [TCP Dup ACK 34#1] 44731 → 179 [ACK] Seq=81 Ack=1 Win=64256 Len=0
    47.971883  10.33.10.11             10.33.10.3             TCP          134 [TCP Retransmission] 44731 → 179 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=80
    49.955917  RealtekU_72:a5:d9       9a:3d:b4:65:e5:f4      ARP           42 Who has 10.33.10.3? Tell 10.33.10.11
    49.957367  9a:3d:b4:65:e5:f4       RealtekU_72:a5:d9      ARP           42 10.33.10.3 is at 9a:3d:b4:65:e5:f4
    50.847258  10.33.10.3              10.33.10.11            TCP           62 [TCP Retransmission] 179 → 44731 [SYN, ACK] Seq=0 Ack=1 Win=16384 Len=0 MSS=1446 W
    50.847342  10.33.10.11             10.33.10.3             TCP           54 [TCP Dup ACK 34#2] 44731 → 179 [ACK] Seq=81 Ack=1 Win=64256 Len=0
    51.491861  10.33.10.11             10.33.10.3             TCP          134 [TCP Retransmission] 44731 → 179 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=80
    58.147970  10.33.10.11             10.33.10.3             TCP          134 [TCP Retransmission] 44731 → 179 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=80
    58.847429  10.33.10.3              10.33.10.11            TCP           62 [TCP Retransmission] 179 → 44731 [SYN, ACK] Seq=0 Ack=1 Win=16384 Len=0 MSS=1446 W
    58.847522  10.33.10.11             10.33.10.3             TCP           54 [TCP Dup ACK 34#3] 44731 → 179 [ACK] Seq=81 Ack=1 Win=64256 Len=0
    71.459841  10.33.10.11             10.33.10.3             TCP          134 [TCP Retransmission] 44731 → 179 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=80
    75.728482  fe80::983d:b4ff:fe65:e5… ff02::1               ICMPv6       118 Router Advertisement from 9a:3d:b4:65:e5:f4
    81.145326  10.33.10.3              10.33.10.11            TCP           62 [TCP Port numbers reused] 43527 → 179 [SYN] Seq=0 Win=16384 Len=0 MSS=1446 WS=1
    81.145426  10.33.10.11             10.33.10.3             TCP           54 179 → 43527 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
    83.145014  10.33.10.3              10.33.10.11            TCP           62 [TCP Retransmission] [TCP Port numbers reused] 43527 → 179 [SYN] Seq=0 Win=16384 L
    83.145094  10.33.10.11             10.33.10.3             TCP           54 179 → 43527 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
    84.516243  9a:3d:b4:65:e5:f4       CDP/VTP/DTP/PAgP/UDLD  CDP          166 Device ID: XR3  Port ID: GigabitEthernet0/0/0/1
    86.307839  RealtekU_72:a5:d9       9a:3d:b4:65:e5:f4      ARP           42 Who has 10.33.10.3? Tell 10.33.10.11
    86.309250  9a:3d:b4:65:e5:f4       RealtekU_72:a5:d9      ARP           42 10.33.10.3 is at 9a:3d:b4:65:e5:f4
    87.145366  10.33.10.3              10.33.10.11            TCP           62 [TCP Retransmission] [TCP Port numbers reused] 43527 → 179 [SYN] Seq=0 Win=16384 L
```

Figure 6.3.: TCP Retransmissions

- The backbone was downsized to only one router due to TCP retransmission problems and for simplicity. The original infrastructure figure B.1, which is displayed in the chapter appendix B contained three Cisco IOS XR devices in the core. Nevertheless, we encountered multiple TCP retransmission errors after configuring BGP on the PE devices and on Kubernetes. This retransmission caused the BGP peering to reset. We used a program to simulate TCP traffic where we observed the same behavior. These errors were discussed with our advisor, and we decided to downsize the backbone infrastructure to only one router. However, the TCP transmission still occurred, and we decided to switch to cloud and local environment for demo purposes.

6.12. Cilium SRv6 Configuration Issues

Date: 17.05.2023
Decided by: Group
Brief description: Cilium SRv6 issues were solved after receiving a lab example.

- We had difficulties setting the SRv6 control plane up because the encapsulation mechanism did not work. We sent debug information to Philip Schmid, who offered us a debugging session. The appointment gave us more insights into Cilium SRv6, but we could not solve the failing encapsulation mechanism. Philip Schmid sent us an example of a development lab that we used to replicate some configurations. We found out that we had some missing configurations in Cilium and on the VM. Thanks to Philip Schmid, we could enable and use SRv6 and observe the L3VPN service between pods.

CHAPTER 7

Notable Points

Any notable and vital aspects of this bachelor thesis, regarding the technical part, will be documented in this section. These difficulties, under certain circumstances, may cause changes to our thesis.

7.1. Nephio POC Version

During the project, only a PoC version of Nephio was available. This version possesses a continuous delivery mechanism for deploying KRM resources in the form of KPT packages to multiple Kubernetes clusters. The team has asked the Linux Foundation for the availability of a full version, who responded that Google would release a code drop to the community in May 2023. The PoC version was used as a demo for telcos to show the provisioning of 5G workloads, which was the primary use case and motivation for the Nephio project. It was sufficient for the thesis to successfully deploy workloads to other clusters and support source control for all packages and deployments. There is little information available on what functionalities the full version will support. Despite being incomplete, the team will work with the PoC in this project as it supports the provisioning of resources to multiple clusters. If there is enough time for an evaluation in May 2023, the team will evaluate the full version.

7.2. Nephio in Combination with GitLab

The PoC repository of Nephio states that Nephio can work with repositories in GitHub or Google Cloud Source Repository service. The example is shown with GitHub.[1]
Because we are using the provided GitLab service by OST, we tried to run Nephio with GitLab. The configurations used to initialize Nephio with GitHub were changed to match configurations with GitLab.
The changes are in the appendix chapter "Installation Manuals" A.
The Nephio management cluster could push packages to the repository in GitLab. Unfortunately, we could not deploy these packages on the Nephio edge cluster with GitLab. The Nephio edge cluster could not handle credentials provided for GitLab, and the following error occurred in the Nephio config sync pod, which is responsible for pulling packages from the repository to the cluster.

```
$ kubectl logs root-reconciler-nephio-workload-cluster-sync-6f468bbbcc-5vdbj -n
    config-management-system
For more information, see https://g.co/cloud/acm-errors#knv2004
E0328 12:40:22.432514       1 controller.go:89] failed to get the commit hash and sync
    directory from the source directory /repo/source/rev: KNV2004: unable to sync repo
Error in the git-sync container: {"Msg":"unexpected error syncing repo, will
    retry","Err":"Run(git clone -v --no-checkout -b main --depth 1
    https://gitlab.ost.ch/ba-cloud-native-intent-automation/nephio-edge-cluster-01
    /repo/source): exit status 128: { stdout: \"\", stderr: \"Cloning into
```

[1]Nephio PoC [Nep23b]

```
        '/repo/source'...\\nfatal: could not read Username for 'https://gitlab.ost.ch': No such
        device or address\\n\" }","Args":{}}
```

Listing 7.1: Error in Nephio Workload Cluster Sync pod with GitLab configurations

Therefore GitHub was used to communicate with Nephio.

7.3. Cilium Provisioning with Nephio

At the beginning of the project, the idea was to deploy Cilium CNI plugin on Kubernetes nodes with Nephio, which would bring the same benefits for the CNI plugin as for application and configuration packages. The Cilium configuration parameters, such as the network CIDR and amount of operators, would be managed by Nephio, which would enable tight coupling between the CNI plugin and its YAML configurations. However, after the evaluation of Nephio, the team had to use another method to deploy the CNI plugin as Nephio itself needs to run on Kubernetes containers and without a preinstalled CNI plugin it could not communicate over the network to fulfill its tasks.
The team had to utilize the fallback variant to use Ansible to deploy the CNI plugin before Nephio.

7.4. Automatically Create GitHub Repositories with Ansible

Ansible offers a community module 'community.general.github_repo' to specifically create GitHub repositories.

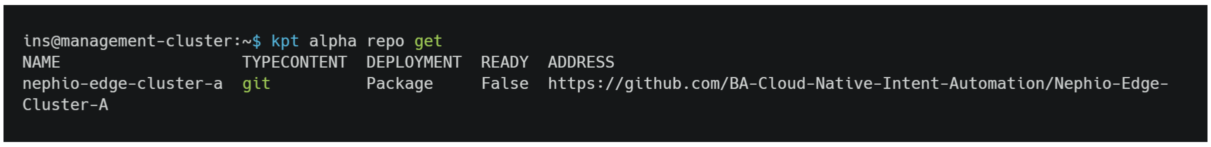
```
- name: Create a Github repository
community.general.github_repo:
    access_token: {placeholder for access token}
    organization: BA-Cloud-Native-Intent-Automation
    name: Nephio-Edge-Cluster-A
    description: "Repository for Edge-Cluster-A"
    private: true
    state: present
    force_defaults: false
register: result
```

Listing 7.2: Ansible Module github_repo create Repository

This method initializes an empty repository with the name 'Nephio-Edge-Cluster-A'. When the repository was registered with KPT on the management cluster, KPT did not mark the repository as ready.

```
kpt alpha repo register \
    --deployment \
    --namespace default \
    --repo-basic-username={GITHUB_USERNAME} \
    --repo-basic-password={GITHUB_TOKEN} \
    https://github.com/BA-Cloud-Native-Intent-Automation/Nephio-Edge-Cluster-A.git
```

Listing 7.3: Register Repository Nephio-Edge-Cluster-A with KPT



```
ins@management-cluster:~$ kpt alpha repo get
NAME                   TYPECONTENT  DEPLOYMENT  READY  ADDRESS
nephio-edge-cluster-a  git          Package     False  https://github.com/BA-Cloud-Native-Intent-Automation/Nephio-Edge-
Cluster-A
```

Figure 7.1.: Empty Repository not ready

It was later discovered that KPT did not work with empty repositories. The Ansible 'community.general.github_repo' module did not offer an option to create a standard ReadMe file with the repository. We, therefore, had to use the Ansible 'ansible.builtin.uri' module to create an HTTP POST request and set the 'aut_init' flag to true to create the ReadMe file automatically.

```
- name: Create repositories Nephio-Edge-Cluster-A on GitHub
    ansible.builtin.uri:
        url: "https://api.github.com/orgs/BA-Cloud-Native-Intent-Automation/repos"
        headers:
            Accept: application/vnd.github+json
            Authorization: Bearer {GITHUB_TOKEN}
            X-GitHub-Api-Version: "2022-11-28"
        method: POST
        body: {'auto_init': true, 'name': "Nephio-Edge-Cluster-A}"}
        status_code: 201
        body_format: json
    register: result
    ignore_errors: true
```

Listing 7.4: Manually create README file with auto_init Flag true

7.5. Optional Use Case 4 - Provisioning of L3VPN Configurations on Provider Routers

In the optional use case 4, the VRF configurations for each edge cluster, which were previously done manually, should now be implemented in the Ansible deployment. When a new site is deployed, VRF configurations must be implemented to connect this site with existing ones.

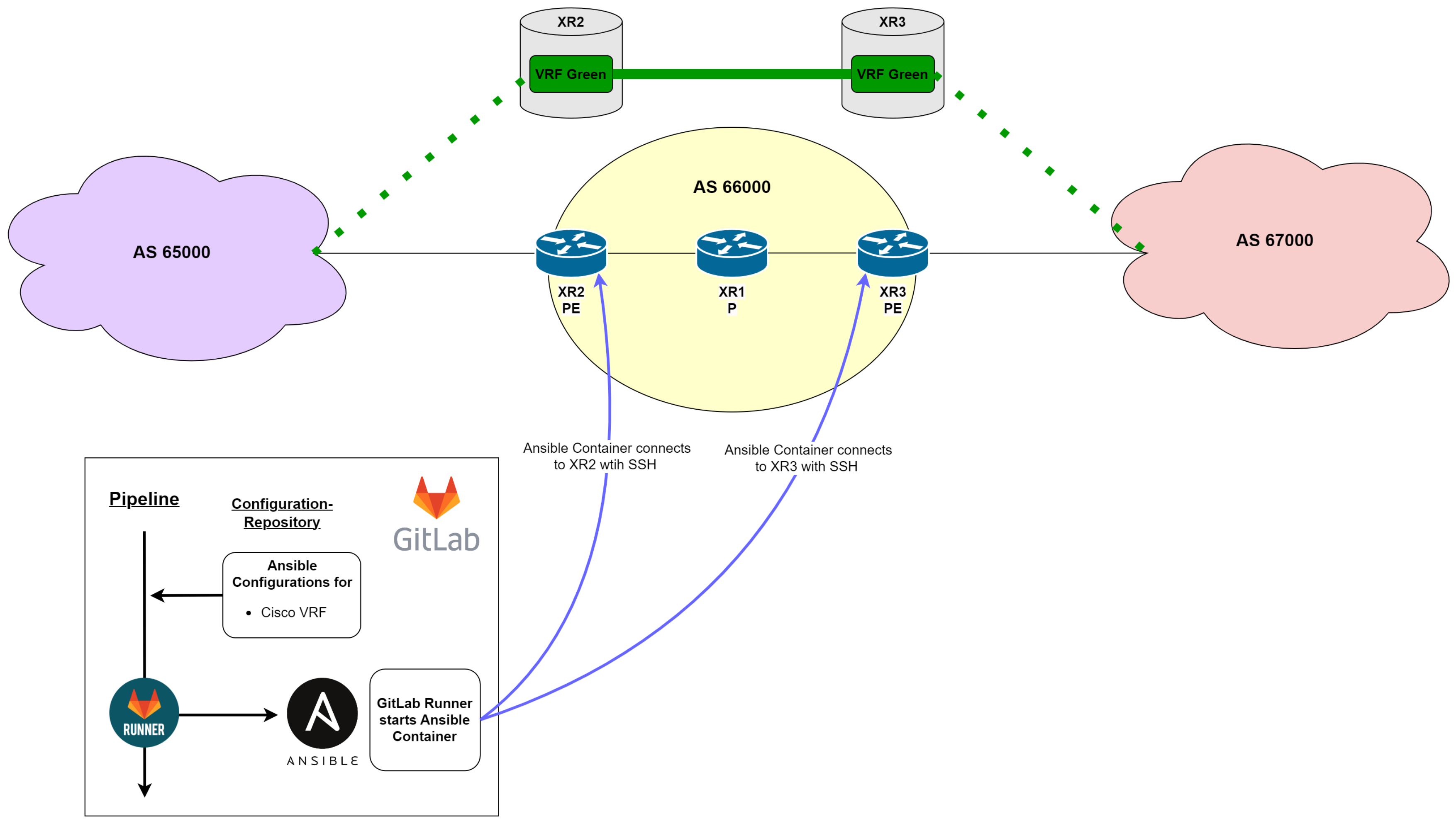

XR2
VRF Green
XR3
VRF Green
AS 65000
AS 66000
AS 67000
XR2 PE
XR1 P
XR3 PE
Ansible Container connects to XR2 wtih SSH
Ansible Container connects to XR3 with SSH
Pipeline
Configuration-Repository
GitLab
Ansible Configurations for
• Cisco VRF
RUNNER
ANSIBLE
GitLab Runner starts Ansible Container

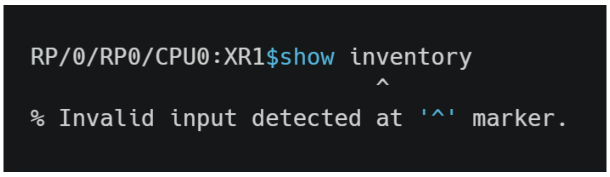
Figure 7.2.: Use Case 4 Diagram

Ansible provides specific modules to configure Cisco devices, which must be installed in the Ansible container. For Cisco IOS XR devices, ansible provides the 'Cisco.Iosxr' collection[2].
The 'Cisco.Iosxr' Ansible collection tries to run the command 'show inventory' on the routers.

```
    def get_device_info(self):
        data = self.get_command_output("show inventory")
```

Listing 7.5: 'show inventory' Command being run in Cisco.Iosxr Collection - File 'iosxr.py'

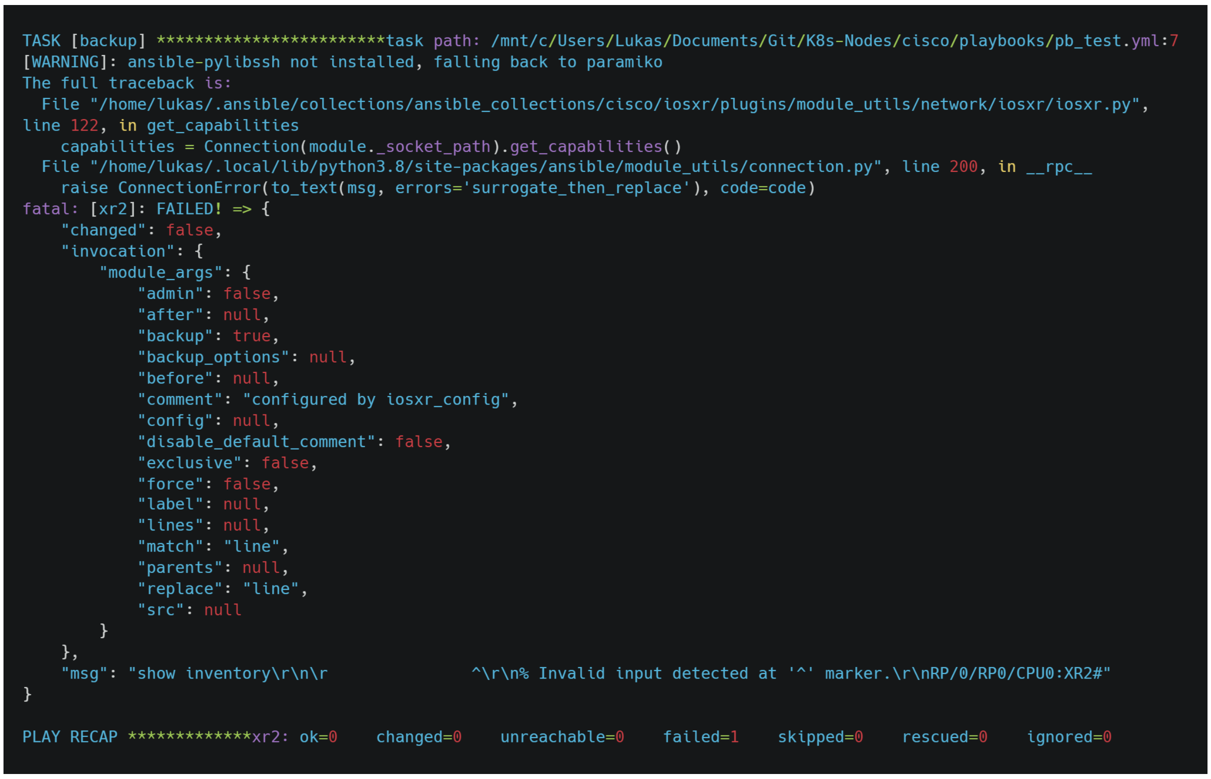
Due to unknown reasons, this command does not exist on our Cisco IOS XR routers in the LTB lab.



```
RP/0/RP0/CPU0:XR1$show inventory
                       ^
% Invalid input detected at '^' marker.
```

Figure 7.3.: Manual Entry of 'show inventory' Command on Cisco IOS XR

The same error also occurred when running the Ansible deployment with any module of the 'Cisco.Iosxr' collection.



```
TASK [backup] ************************task path: /mnt/c/Users/Lukas/Documents/Git/K8s-Nodes/cisco/playbooks/pb_test.yml:7
[WARNING]: ansible-pylibssh not installed, falling back to paramiko
The full traceback is:
  File "/home/lukas/.ansible/collections/ansible_collections/cisco/iosxr/plugins/module_utils/network/iosxr/iosxr.py",
line 122, in get_capabilities
    capabilities = Connection(module._socket_path).get_capabilities()
  File "/home/lukas/.local/lib/python3.8/site-packages/ansible/module_utils/connection.py", line 200, in __rpc__
    raise ConnectionError(to_text(msg, errors='surrogate_then_replace'), code=code)
fatal: [xr2]: FAILED! => {
    "changed": false,
    "invocation": {
        "module_args": {
            "admin": false,
            "after": null,
            "backup": true,
            "backup_options": null,
            "before": null,
            "comment": "configured by iosxr_config",
            "config": null,
            "disable_default_comment": false,
            "exclusive": false,
            "force": false,
            "label": null,
            "lines": null,
            "match": "line",
            "parents": null,
            "replace": "line",
            "src": null
        }
    },
    "msg": "show inventory\r\n\r               ^\r\n% Invalid input detected at '^' marker.\r\nRP/0/RP0/CPU0:XR2#"
}

PLAY RECAP *************xr2: ok=0    changed=0    unreachable=0    failed=1    skipped=0    rescued=0    ignored=0
```

Figure 7.4.: Ansible 'Cisco.Iosxr' Error with 'iosxr-config' Module

With the 'Cisco.Ios' collection[3], which was not explicitly designed for Cisco IOS XR devices, we could perform basic operations like creating a backup of the configuration on the devices. However, we failed to implement more complex configurations like VRF as this collection was not intended to be run on Cisco IOS XR devices. The last option was to use the 'ansible-netcommon'[4] collection. The 'ansible.netcommon' collection uses the

[2]Ansible Cisco.Iosxr Collection [Com23b]
[3]Ansible Cisco.Ios Collection [Com23a]
[4]Ansible Netcommon Collection [PN23]

netconf[5] protocol to send XML commands to a device and detects configuration changes.
As this is no official approach to configure Cisco IOS XR devices with Ansible, little to no documentation was available. Nevertheless, we created an XML backup of the active configuration with the 'netcommon.netconf-config' module. We made the backup file more readable with an XML to JSON converter, reverse-engineered a backup of an already made VRF configuration, and created our own deployment.

```
tasks:
- name: Globally configure VRF Green on Cisco IOS XR
  ansible.netcommon.netconf_config:
    format: json
    content: {
                "config": {
                      "vrfs": {
                        "@xmlns": "http://cisco.com/ns/yang/Cisco-IOS-XR-infra-rsi-cfg",
                          "vrf": {
                            "vrf-name": "VRF Green",
                            "create": "",
                            "afs": {
                              "af": [
                                {
                                  "af-name": "ipv4",
                                  "saf-name": "unicast",
                                  "topology-name": "default",
                                  "create": "",
                                  "bgp": {
                                    "@xmlns":
                                         "http://cisco.com/ns/yang/Cisco-IOS-XR-ipv4-bgp-cfg",
                                    "import-route-targets": {
                                      "route-targets": {
                                        "route-target": {
                                          "type": "as",
                                          "as-or-four-byte-as": {
                                            "as-xx": 0,
                                            "as": 1,
                                            "as-index": 1,
                                            "stitching-rt": 0
                                          }
```

Listing 7.6: Example of Cisco VRF netconf-config Deployment in JSON

Due to missing documentation, this method required lots of trial and error.
Ultimately, we decided against this method because it was not practical for professional usage. These challenges were discussed with the advisor in the meeting of week 12.
Further tests on this issue with real Cisco IOS XR routers were not possible, as such devices were not accessible to the group.

[5] IETF 6241 Netconf [R E+11]

Part II.

Appendix

CHAPTER A

Installation Manuals

a. Running a deployment within GitLab CI/CD Pipeline

The whole thesis is designed to run the deployments within a GitLab-Pipeline.
A quick-start guide and essential information may be found inside the deployment-repositorys' ReadMe-file.

b. Nephio with GitLab on Management Cluster

The following configurations from the Nephio manual were changed to match the configurations for GitLab.[1]
Unfortunately, as mentioned in the technical chapter "Notable Points", we were not able to run Nephio with GitLab.
Shell variables for GitLab were set.

```
$ GITLAB_USERNAME=lukas.schlunegger
$ GITLAB_TOKEN='not shown for security reasons'
```

Listing A.1: Error in Nephio Workload Cluster Sync pod with GitLab configurations

The repositories in GitLab were added to the Nephio management cluster:

```
kpt alpha repo register \
    --namespace default \
    --repo-basic-username=${GITLAB_USERNAME} \
    --repo-basic-password=${GITLAB_TOKEN} \
    https://gitlab.ost.ch/ba-cloud-native-intent-automation/nephio-test-catalog.git

kpt alpha repo register \
    --deployment \
    --namespace default \
    --repo-basic-username=${GITLAB_USERNAME} \
    --repo-basic-password=${GITLAB_TOKEN} \
    https://gitlab.ost.ch/ba-cloud-native-intent-automation/nephio-edge-cluster-01.git

kpt alpha repo register \
    --namespace default \
    https://github.com/nephio-project/nephio-packages.git
```

Listing A.2: Added GitLab repositories to nephio management cluster

Nephio needs to know which repository should be used to communicate whit the specific cluster. The value containing 'https://github.com/[a-zA-Z0-9-]+/(.*)' in the standard configuration has to be replaced with a link to our repository in GitLab.

[1]Nephio PoC [Nep23b]

```
kpt pkg get --for-deployment
     https://github.com/nephio-project/nephio-packages.git/nephio-configsync
     nephio-edge-cluster-01
kpt fn eval nephio-edge-cluster-01 \
    --save \
    --type mutator \
    --image gcr.io/kpt-fn/search-replace:v0.2.0 \
    -- by-path=spec.git.repo by-value-regex='https://github.com/[a-zA-Z0-9-]+/(.*)' \
    put-value="https://gitlab.ost.ch/ba-cloud-native-intent-automation/\${1}"
kpt fn render nephio-edge-cluster-01
kpt live init nephio-edge-cluster-01
kpt live apply nephio-edge-cluster-01 --reconcile-timeout=15m --output=table
```

Listing A.3: Error in Nephio Workload Cluster Sync pod with GitLab configurations

After applying the changes, the resources are created with 'kpt live apply nephio-edge-cluster-01' with the new GitLab configurations.

```
kpt live apply nephio-edge-cluster-01 --reconcile-timeout=15m --output=table
[RUNNING] "gcr.io/kpt-fn/search-replace:v0.2.0"
[PASS] "gcr.io/kpt-fn/search-replace:v0.2.0" in 600ms
Results:
    [info] spec.git.repo: Mutated field value to
         "https://gitlab.ost.ch/ba-cloud-native-intent-automation/test-edge-01"
Added "gcr.io/kpt-fn/search-replace:v0.2.0" as mutator in the Kptfile.
Package "nephio-edge-cluster-01":
[RUNNING] "gcr.io/kpt-fn/apply-replacements:v0.1.1"
[PASS] "gcr.io/kpt-fn/apply-replacements:v0.1.1" in 600ms
[RUNNING] "gcr.io/kpt-fn/search-replace:v0.2.0"
[PASS] "gcr.io/kpt-fn/search-replace:v0.2.0" in 500ms
Results:
    [info]: no matches

Successfully executed 2 function(s) in 1 package(s).
initializing "resourcegroup.yaml" data (namespace: default)...success
installing inventory ResourceGroup CRD.
```

Listing A.4: Ouput of 'kpt live apply'

The line '[Info]: no matches' was not seen when this command was run with GitHub configurations. There was a message similar to the message 'spec.git.repo: Mutated field value to' above, which is most likely why we are having an error when running Nephio with GitLab configurations on the nephio edge cluster. The replacement of GitHub with GitLab repositories, did not seem to work entirely.
After applying the configurations the edge-workload-cluster-sync pod has following log entries:

```
$ kubectl logs root-reconciler-nephio-workload-cluster-sync-6f468bbbcc-5vdbj -n
     config-management-system
For more information, see https://g.co/cloud/acm-errors#knv2004
E0328 12:40:22.432514       1 controller.go:89] failed to get the commit hash and sync
     directory from the source directory /repo/source/rev: KNV2004: unable to sync repo
Error in the git-sync container: {"Msg":"unexpected error syncing repo, will
     retry","Err":"Run(git clone -v --no-checkout -b main --depth 1
     https://gitlab.ost.ch/ba-cloud-native-intent-automation/nephio-edge-cluster-01
     /repo/source): exit status 128: { stdout: \"\", stderr: \"Cloning into
     '/repo/source'...\\nfatal: could not read Username for 'https://gitlab.ost.ch': No such
     device or address\\n\" }","Args":{}}
```

Listing A.5: Error in Nephio Workload Cluster Sync pod with GitLab configurations

Which showed an unsuccessful connection between the nephio edge cluster sync pod and GitLab repositories.

c. Installation of Management Cluster

c.a. Kubernetes

To install the Kubernetes cluster on the management VM, an installation guide was used.[2]

Set Hostname and Add Entries in Hosts File

Set hostname of control plane using hostnamectl command.

```
$ sudo hostnamectl set-hostname "management-cluster.example.net"
$ exec bash
```

Listing A.6: Set Hostname of Cluster

Added IPv4 and IPv6 addresses to /etc/hosts for name resolving.

```
$ sudo vim /etc/hosts
10.11.10.2 management-cluster.example.net management-cluster
2001:db8:1f::2 management-cluster.example.net management-cluster
```

Listing A.7: Add Entries in Host File

Disable Swap and Add Kernel Settings

Kubernetes requires a disabled swap on the host system.

```
$ sudo swapoff -a
$ sudo sed -i '/ swap / s/^\(.*\)$/#\1/g' /etc/fstab
```

Listing A.8: Disable Swap on Host System

Load following kernel modules on node.

```
$ sudo tee /etc/modules-load.d/containerd.conf <<EOF
overlay
br_netfilter
EOF
$ sudo modprobe overlay
$ sudo modprobe br_netfilter
```

Listing A.9: Add Kernel Settings

Set following Kernel parameters for Kubernetes.

```
$ sudo tee /etc/sysctl.d/kubernetes.conf <<EOF
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
EOF
```

Listing A.10: Set Kernel Parameters for Kubernetes

Enable IPv6 forwarding.

```
$ sudo sysctl -w net.ipv6.conf.all.forwarding=1
```

Listing A.11: Enable IPv6 Forwarding

Reload changes above and persist across reboots.

[2] Install Kubernetes Cluster on Ubuntu [Kum22]

```
$ sudo sysctl --system
```

Listing A.12: Reload Kernel

Install containerd run time

Install dependencies to install containerd.

```
$ sudo apt install -y curl gnupg2 software-properties-common apt-transport-https ca-certificates
```

Listing A.13: Install containerd Dependencies

Enable docker repository.

```
$ sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmour -o
    /etc/apt/trusted.gpg.d/docker.gpg
$ sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu
    $(lsb_release -cs) stable"
```

Listing A.14: Enable Docker Repository

Retrieve information on latest packages and install containerd.

```
$ sudo apt update
$ sudo apt install -y containerd.io
```

Listing A.15: Install containerd

Configure containerd to start using systemd as cgroup for Ubuntu version 22.04.

```
$ containerd config default | sudo tee /etc/containerd/config.toml >/dev/null 2>&1
$ sudo sed -i 's/SystemdCgroup \= false/SystemdCgroup \= true/g' /etc/containerd/config.toml
```

Listing A.16: Configure containerd to use systemd as cgroup

Restart and enable the service.

```
$ sudo systemctl restart containerd
$ sudo systemctl enable containerd
```

Listing A.17: Enable containerd service

Add apt repository for Kubernetes

Add repositories for Kubernetes.

```
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
$ sudo apt-add-repository "deb http://apt.kubernetes.io/ kubernetes-xenial main"
```

Listing A.18: Add Repositories for Kubernetes

At the time of this writing, Xenial was still the latest Kubernetes repository. When the Kubernetes repository for Ubuntu 22.04 (Jammy Jellfish) is available, replace xenial with 'jammy' in apt-add repository command.

Install Kubernetes components Kubectl, kubeadm and kubelet

Install Kubectl, kubeadm and kubelet on the node, but first update packages.

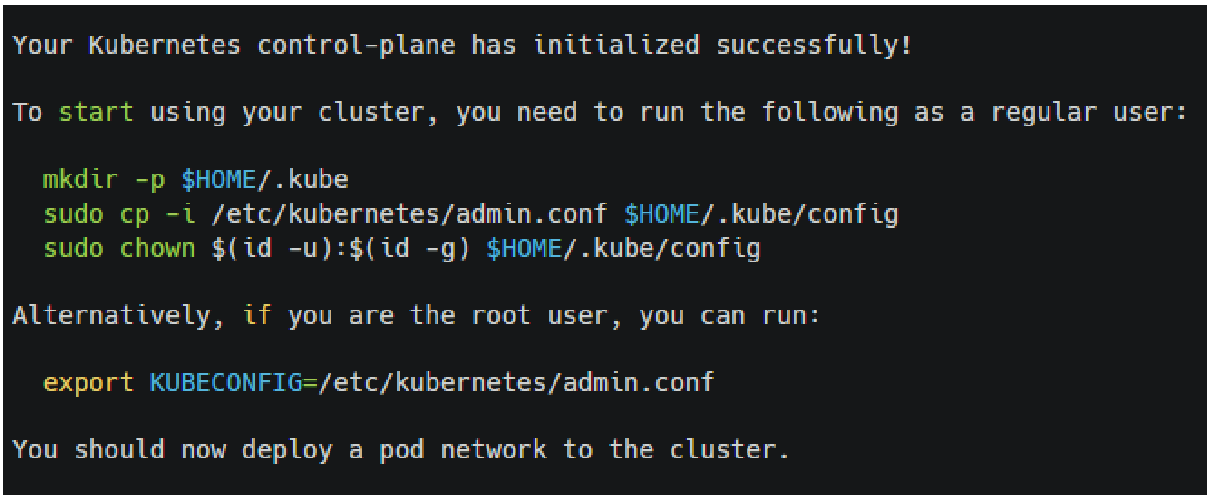
```
$ sudo apt update
$ sudo apt install -y kubelet kubeadm kubectl
$ sudo apt-mark hold kubelet kubeadm kubectl
```

Listing A.19: Install Kubernetes Components

Initialize Kubernetes cluster

```
sudo kubeadm init --control-plane-endpoint=10.11.10.2 \
    --pod-network-cidr=10.1.0.0/16,fd00:f::/64 --service-cidr=10.96.0.0/24,2001:db8:a::/112 \
    --skip-phases=addon/kube-proxy
```

Listing A.20: Initialize Kubeadm with IPv4 and IPv6 Dual-Stack



```
Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

Alternatively, if you are the root user, you can run:

  export KUBECONFIG=/etc/kubernetes/admin.conf

You should now deploy a pod network to the cluster.
```

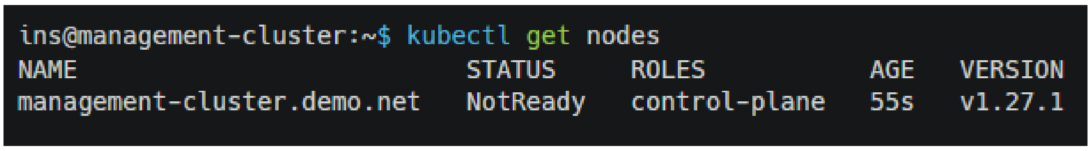
Figure A.1.: Kubernetes Ready

To interact with cluster, following commands need to be run.

```
$ mkdir -p $HOME/.kube
$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Listing A.21: Configuration for Kubectl Access

Kubernetes is running but the node is not ready because a CNI plugin is missing.



```
ins@management-cluster:~$ kubectl get nodes
NAME                           STATUS     ROLES           AGE   VERSION
management-cluster.demo.net    NotReady   control-plane   55s   v1.27.1
```

Figure A.2.: Nodes not Ready because of missing CNI Plugin

Management Cluster needed only one node, therefore the label "Control-plane:NoSchedule" was removed.

```
$ kubectl taint node management-cluster.example.net
    node-role.kubernetes.io/control-plane:NoSchedule-
```

Listing A.22: Remove NoSchedule Label from Master Node

c.b. Helm Installation

Helm was needed to install Cilium.
Followed installation manual on helm website.[3]
First download installation files.

```
$ curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
```

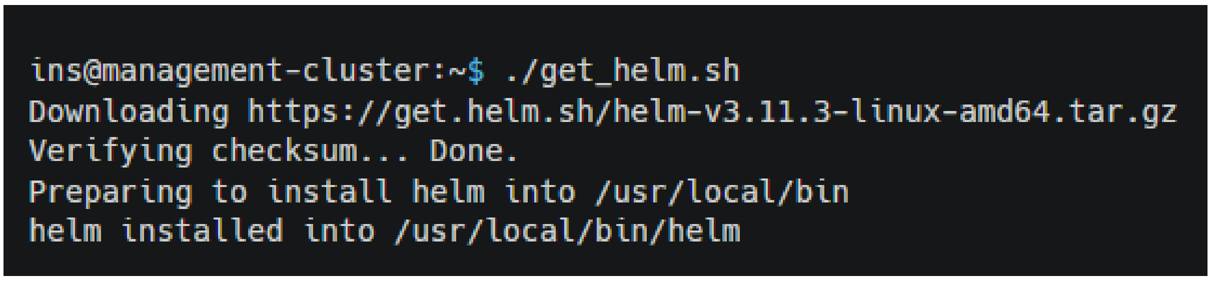
Listing A.23: Download Helm

Make files executable and run start installation.

```
$ chmod 700 get_helm.sh
$ ./get_helm.sh
```

Listing A.24: Install Helm

Helm was now ready.



```
ins@management-cluster:~$ ./get_helm.sh
Downloading https://get.helm.sh/helm-v3.11.3-linux-amd64.tar.gz
Verifying checksum... Done.
Preparing to install helm into /usr/local/bin
helm installed into /usr/local/bin/helm
```

Figure A.3.: Helm Installed

c.c. Cilium Installation

Add community repository of Cilium.

```
$ helm repo add cilium https://helm.cilium.io/
```

Listing A.25: Add Community Repository of Helm

Configure Cilium release with following Helm flags.

```
$ helm install cilium cilium/cilium \
    --namespace kube-system \
    --set kubeProxyReplacement=strict \
    --set operator.replicas=1 \
    --set k8sServiceHost=10.11.10.2 \
    --set k8sServicePort=6443 \
    --set ipv4.enabled=true \
    --set ipv6.enabled=true \
    --set ipam.mode=cluster-pool \
    --set ipam.operator.clusterPoolIPv4PodCIDRList="10.1.0.0/16" \
    --set ipam.operator.clusterPoolIPv6PodCIDRList="fd00:f::/112" \
    --set ipam.operator.clusterPoolIPv4MaskSize=24 \
    --set ipam.operator.clusterPoolIPv6MaskSize=112 \
    --set bgpControlPlane.enabled=true \
    --set bpf.masquerade=true \
    --set enableIPv6Masquerade=false
```

Listing A.26: Install and Configure Cilium

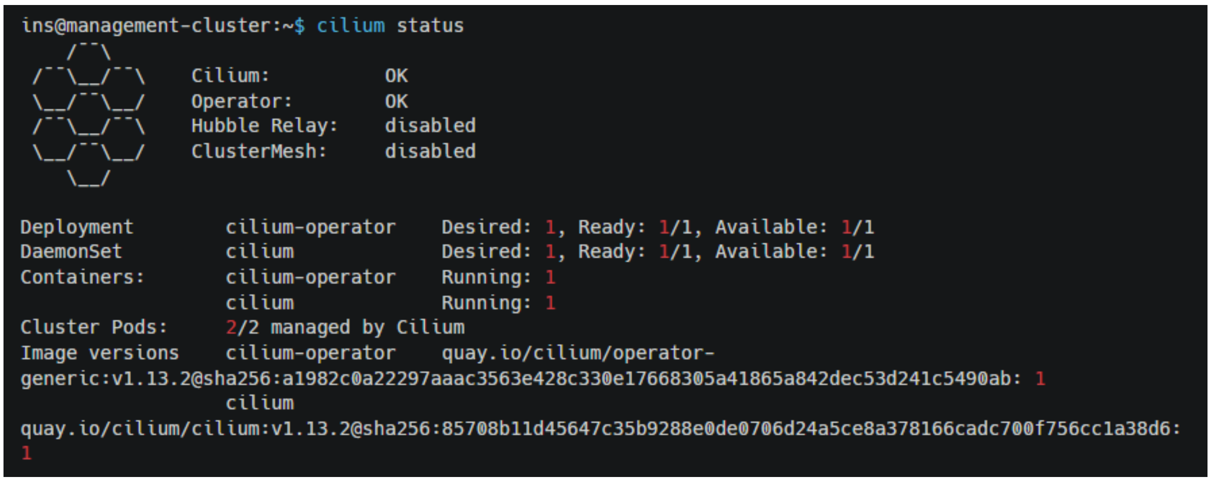
Install Cilium CLI.

[3]Installing Helm [Aut23]

```
$ CILIUM_CLI_VERSION=$(curl -s
    https://raw.githubusercontent.com/cilium/cilium-cli/master/stable.txt)
$ CLI_ARCH=amd64
$ if [ "$(uname -m)" = "aarch64" ]; then CLI_ARCH=arm64; fi
$ curl -L --fail --remote-name-all
   https://github.com/cilium/cilium-cli/releases/download/${CILIUM_CLI_VERSION}/
   cilium-linux-${CLI_ARCH}.tar.gz{,.sha256sum}
$ sha256sum --check cilium-linux-${CLI_ARCH}.tar.gz.sha256sum
$ sudo tar xzvfC cilium-linux-${CLI_ARCH}.tar.gz /usr/local/bin
$ rm cilium-linux-${CLI_ARCH}.tar.gz{,.sha256sum}
```

Listing A.27: Install CIlium CLI

The installation was successful and all cilium pods were operational.

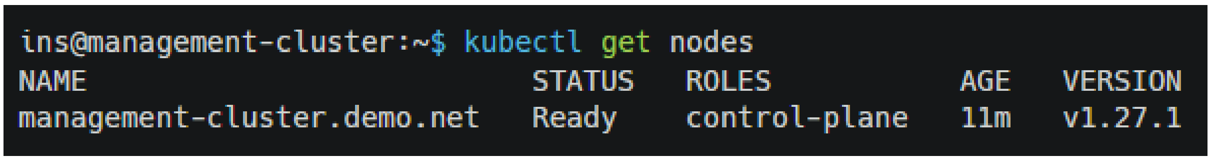


```
ins@management-cluster:~$ cilium status
    /¯¯\
 /¯¯\__/¯¯\    Cilium:         OK
 \__/¯¯\__/    Operator:       OK
 /¯¯\__/¯¯\    Hubble Relay:   disabled
 \__/¯¯\__/    ClusterMesh:    disabled
    \__/

Deployment        cilium-operator    Desired: 1, Ready: 1/1, Available: 1/1
DaemonSet         cilium             Desired: 1, Ready: 1/1, Available: 1/1
Containers:       cilium-operator    Running: 1
                  cilium             Running: 1
Cluster Pods:     2/2 managed by Cilium
Image versions    cilium-operator    quay.io/cilium/operator-
generic:v1.13.2@sha256:a1982c0a22297aaac3563e428c330e17668305a41865a842dec53d241c5490ab: 1
                  cilium
quay.io/cilium/cilium:v1.13.2@sha256:85708b11d45647c35b9288e0de0706d24a5ce8a378166cadc700f756cc1a38d6:
1
```

Figure A.4.: Successful Cilium Installation

The node management-cluster.demo.net was now ready as well.



```
ins@management-cluster:~$ kubectl get nodes
NAME                         STATUS   ROLES           AGE   VERSION
management-cluster.demo.net  Ready    control-plane   11m   v1.27.1
```

Figure A.5.: Nodes Ready after Cilium Installation

c.d. KPT Installation

To Install KPT the official installation manual of KPT was consulted.[4]

```
$ wget https://github.com/GoogleContainerTools/kpt/releases/download/ \
   v1.0.0-beta.25/kpt_linux_amd64
$ chmod +x kpt_linux_amd64
$ sudo mv kpt_linux_amd64 /usr/local/bin/kpt
```

Listing A.28: Install KPT

c.e. Nephio Installation

To install Nephio the installation guide on the 'nephio-poc' GitHub page was used.[5] At the time of writing this thesis only a PoC version was available and the repository has been archived as of March 23, 2023.

[4]KPT Installation [KPT23c]
[5]GitHub Nephio PoC [Nep23b]

```
$ mkdir nephio-install
$ cd nephio-install

$ kpt pkg get --for-deployment
    https://github.com/nephio-project/nephio-packages.git/nephio-system
$ kpt fn render nephio-system

$ kpt live init nephio-system
$ kpt live apply nephio-system --reconcile-timeout=15m --output=table
```

Listing A.29: Install Nephio

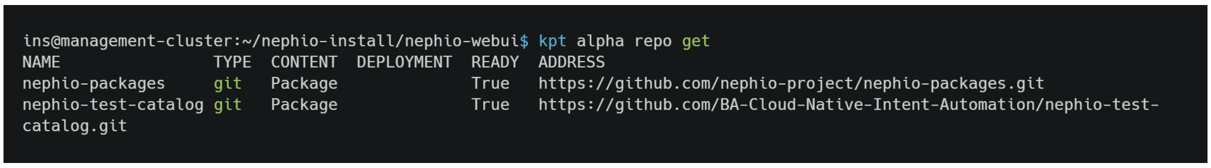
In order for Nephio to know which repository contains installation packages, the repositories have to be added with KPT. Our own 'nephio-test-catalog' and Nephio's 'nephio-package' repositories were added. These repositories can later be used to deploy installation packages.
Replace GITHUB USERNAME and GITHUB TOKEN with the username and access token for your personal repository.

```
$ kpt alpha repo register \
    --namespace default \
    --repo-basic-username={GITHUB USERNAME} \
    --repo-basic-password={GITHUB TOKEN} \
    https://github.com/BA-Cloud-Native-Intent-Automation/nephio-test-catalog.git

$ kpt alpha repo register \
    --namespace default \
    https://github.com/nephio-project/nephio-packages.git
```

Listing A.30: Add Repositories to KPT

The registered repositories were shown with the following command.



```
ins@management-cluster:~/nephio-install/nephio-webui$ kpt alpha repo get
NAME                TYPE  CONTENT  DEPLOYMENT  READY  ADDRESS
nephio-packages     git   Package              True   https://github.com/nephio-project/nephio-packages.git
nephio-test-catalog git   Package              True   https://github.com/BA-Cloud-Native-Intent-Automation/nephio-test-
catalog.git
```

Figure A.6.: Show KPT Registered Repositories

c.f. Web-UI Installation

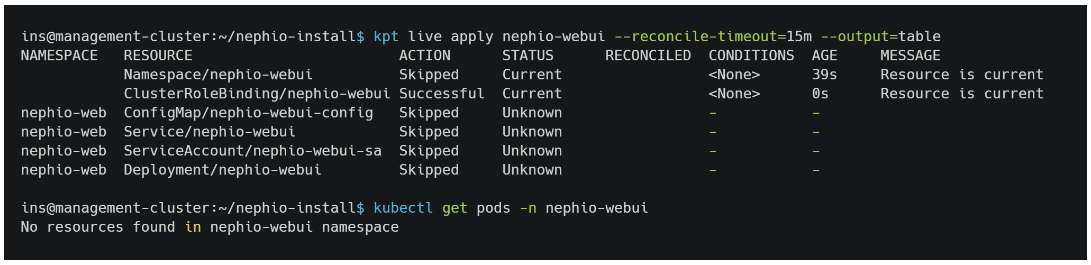
```
$ kubectl create ns nephio-webui

$ kpt pkg get --for-deployment
    https://github.com/nephio-project/nephio-packages.git/nephio-webui
$ kpt fn render nephio-webui

$ kpt live init nephio-webui
$ kpt live apply nephio-webui --reconcile-timeout=15m --output=table
```

Listing A.31: Install Nephio Web-UI KPT

After running 'kpt live apply' unlike with the 'nephio-system' pods, the 'nephio-webui' pods did not get created.



```
ins@management-cluster:~/nephio-install$ kpt live apply nephio-webui --reconcile-timeout=15m --output=table
NAMESPACE   RESOURCE                          ACTION      STATUS      RECONCILED  CONDITIONS  AGE     MESSAGE
            Namespace/nephio-webui            Skipped     Current                 <None>      39s     Resource is current
            ClusterRoleBinding/nephio-webui   Successful  Current                 <None>      0s      Resource is current
nephio-web  ConfigMap/nephio-webui-config     Skipped     Unknown                 -           -
nephio-web  Service/nephio-webui              Skipped     Unknown                 -           -
nephio-web  ServiceAccount/nephio-webui-sa    Skipped     Unknown                 -           -
nephio-web  Deployment/nephio-webui           Skipped     Unknown                 -           -

ins@management-cluster:~/nephio-install$ kubectl get pods -n nephio-webui
No resources found in nephio-webui namespace
```

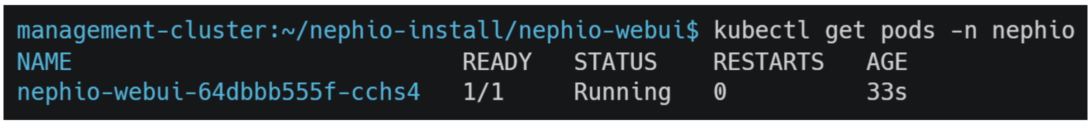
Figure A.7.: Nephio-Webui Pods not being created

All resources in the folder 'nephio-webui' had to be manually created.

```
$ kubectl apply -f 0-namespace.yaml
$ kubectl apply -f cluster-role-binding.yaml
$ kubectl apply -f config-map.yaml
$ kubectl apply -f deployment.yaml
$ kubectl apply -f package-context.yaml
$ kubectl apply -f resourcegroup.yaml
$ kubectl apply -f service-account.yaml
$ kubectl apply -f service.yaml
```

Listing A.32: Manually deploy Nephio-Webui Resources

Afterwards the 'nephio-webui' pod was running.



```
management-cluster:~/nephio-install/nephio-webui$ kubectl get pods -n nephio
NAME                            READY   STATUS    RESTARTS   AGE
nephio-webui-64dbbb555f-cchs4   1/1     Running   0          33s
```

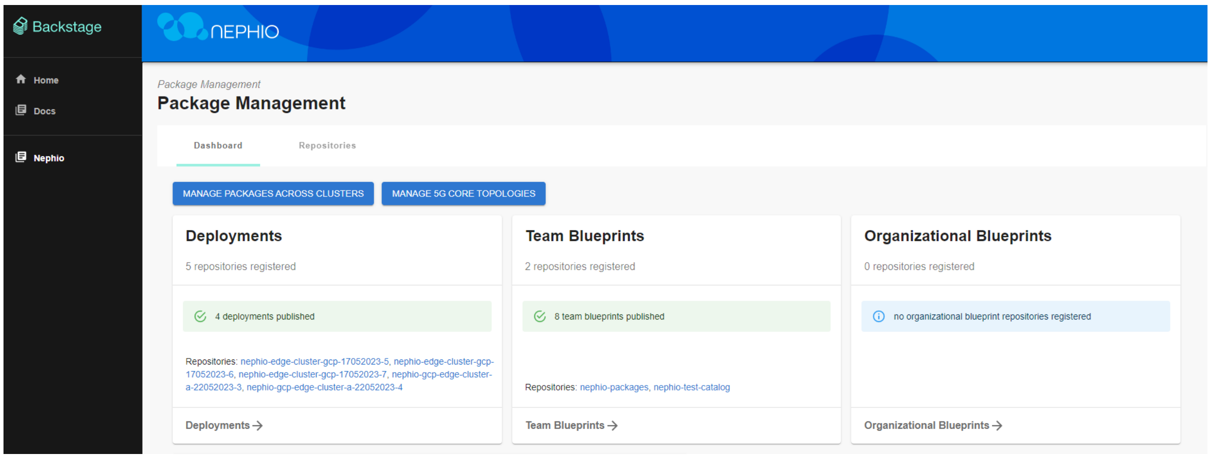
Figure A.8.: Nephio-Webui Pod Ready

Because the LTB lab was behind a gateway, were only SSH connections would be forwarded, we were not able to directly access the 'nephio-webui'. To be able to access the webui on the management-cluster over our local systems, we had to forward port 7007 via the SSH connection.

```
# login from your workstation, forwarding 7007 -> localhost:7007 on the remote VM.
$ ssh -L7007:localhost:7007 ins@10.18.10.25 -p 20002
# in the remote VM run to forward nephio-webui to port 7007
$ kubectl port-forward --namespace=nephio-webui svc/nephio-webui 7007
```

Listing A.33: nephio-webui portforwarding

Afterwards the 'nephio-webui' was locally accessible through a browser with `http://localhost:7007/`.


Backstage
NEPHIO
Home
Docs
Nephio
Dashboard
Repositories
MANAGE PACKAGES ACROSS CLUSTERS
MANAGE 5G CORE TOPOLOGIES
Deployments →
Team Blueprints →
Organizational Blueprints →

Package Management

Package Management

Deployments

5 repositories registered

4 deployments published

Repositories: nephio-edge-cluster-gcp-17052023-5, nephio-edge-cluster-gcp-17052023-6, nephio-edge-cluster-gcp-17052023-7, nephio-gcp-edge-cluster-a-22052023-3, nephio-gcp-edge-cluster-a-22052023-4

Team Blueprints

2 repositories registered

8 team blueprints published

Repositories: nephio-packages, nephio-test-catalog

Organizational Blueprints

0 repositories registered

no organizational blueprint repositories registered

Figure A.9.: Nephio-Webui

The management cluster was now ready and the edge clusters had to be configured next.

d. KPT Package Creation

Nephio uses KPT packages as format for resource deployments. The following manual shows how to create a simple package with only one Kubernetes manifest.
Create a folder with all the resources to be configured and deployed together. It can also be the content of an entire Helm chart. This example uses a package with the folder name 'Cilium-VRF' and a Cilium SRv6VRF resource within. In addition to the actual resources, every package needs a KPTfile:

```
---
apiVersion: kpt.dev/v1
kind: Kptfile
metadata:
  name: netshoot
  annotations:
    config.kubernetes.io/local-config: "true"
info:
  description: sample description
```

Listing A.34: KPTfile example

That example has no specific important configuration. The KPTfile is used to define mutators and validators. In this thesis, we focus on the deployment of a package to clusters. Programming functions to manipulate or validate configurations according to some rules is not part of the scope.
Once all files are added, create a Git Commit for the package.

```
$ git add .
$ git commit -m "add Cilium VRF package"
```

Listing A.35: Commit KPT Package Creation

Create a Git Revision tag according to the example in the snippet. The management cluster will not fetch packages that do not have a revision tag in the form of a Git tag.

```
$ git tag cilium-vrf/v1
```

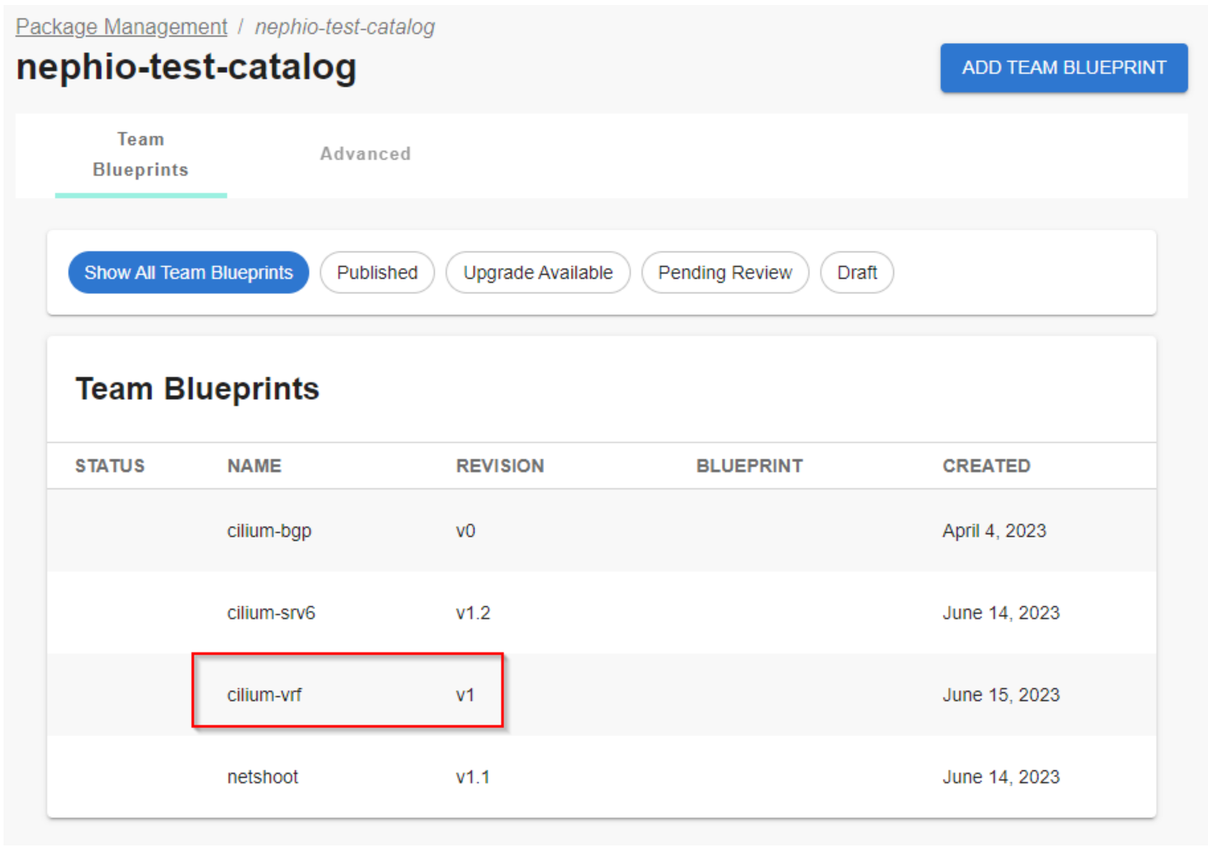
Listing A.36: Commit KPT Package Creation

Push the package together with the revision to the private package catalog.

```
$ git tag cilium-vrf/v1
```

Listing A.37: Push KPT Package to Private Package Catalog

The package will be available on the management cluster as in figure A.10 if KPTfile and the revision were correctly created



STATUS	NAME	REVISION	BLUEPRINT	CREATED
	cilium-bgp	v0		April 4, 2023
	cilium-srv6	v1.2		June 14, 2023
	cilium-vrf	v1		June 15, 2023
	netshoot	v1.1		June 14, 2023

Figure A.10.: Private Package Catalog in Nephio UI

CHAPTER B

Documents

a. Old Network Plan

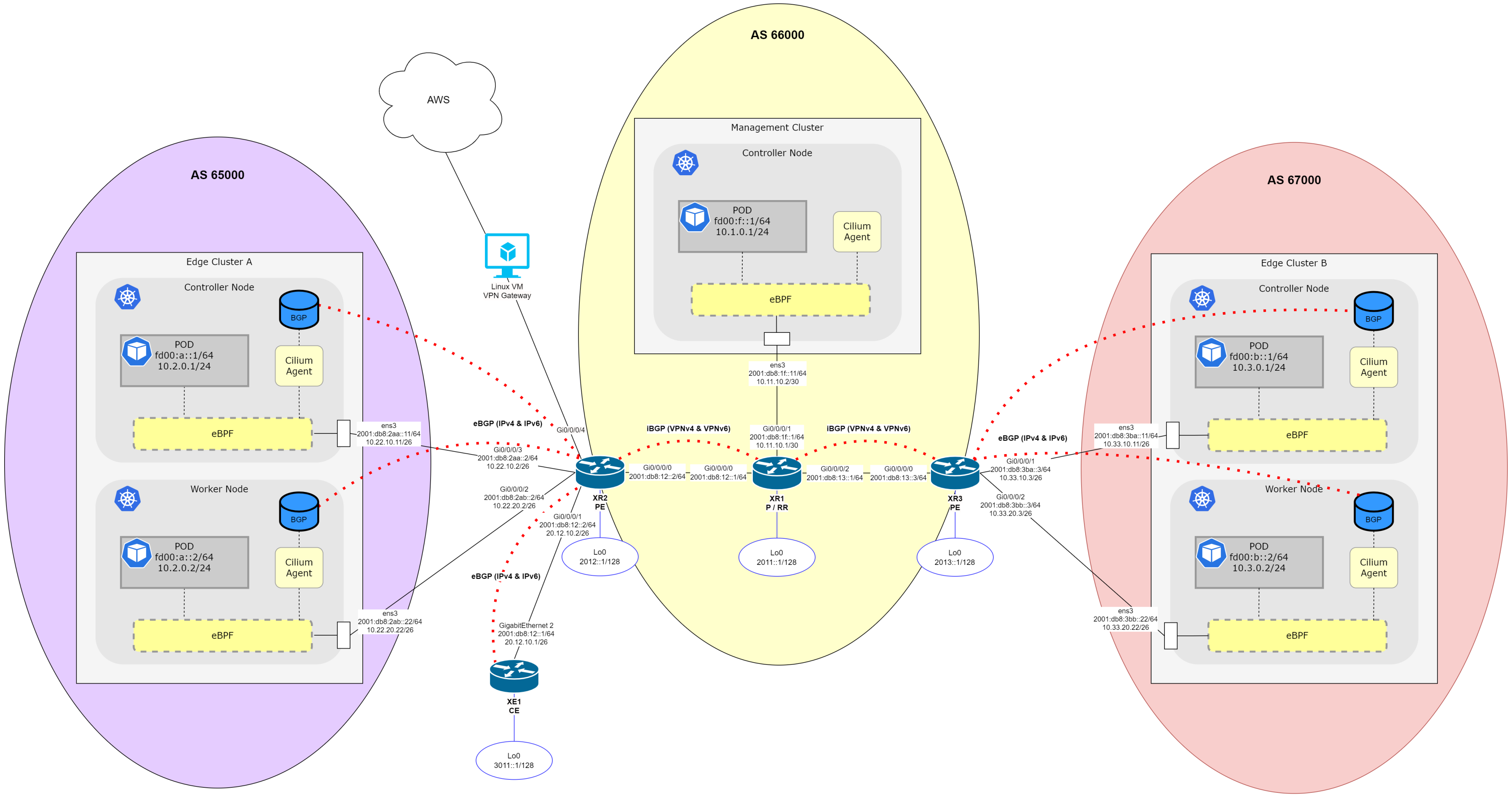


Figure B.1.: Old Network Overview

CHAPTER C

Listings

List of Figures

Listings

CHAPTER D

Bibliography

[Sys+96] Y. Rekhter Cisco Systems et al. *Address Allocation for Private Internets*. Feb. 1, 1996. URL: https://www.rfc-editor.org/rfc/rfc1918.

[TAC04] G. Huston Telstra, A. Lord APNIC, and P. Smith Cisco. *IPv6 Address Prefix Reserved for Documentation*. July 1, 2004. URL: https://datatracker.ietf.org/doc/html/rfc3849.

[R E+11] Ed. R. Enns et al. *Network Configuration Protocol (NETCONF)*. June 1, 2011. URL: https://datatracker.ietf.org/doc/html/rfc6241.

[Com18] Redaktion Computerweekly.de. *WYSIWYG (What You See Is What You Get)*. June 1, 2018. URL: https://www.computerweekly.com/de/definition/WYSIWYG-What-You-See-Is-What-You-Get.

[Dev20] Ubuntu Developers. *docker.io package in Ubuntu*. Oct. 24, 2020. URL: https://launchpad.net/ubuntu/+source/docker.io.

[Hua20] Huawei. *Understanding Segment Routing IPv6*. Oct. 10, 2020. URL: https://support.huawei.com/enterprise/en/doc/EDOC1000173015/d169625f/understanding-segment-routing-ipv6.

[Por+20] Rafael Portela et al. *FluxCD, ArgoCD or Jenkins X: Which Is the Right GitOps Tool for You?* Apr. 30, 2020. URL: https://blog.container-solutions.com/fluxcd-argocd-jenkins-x-gitops-tools.

[Wea20] Weaveworks. *Manage Thousands of Clusters with GitOps and the Cluster API*. Sept. 19, 2020. URL: https://www.weave.works/blog/manage-thousands-of-clusters-with-gitops-and-the-cluster-api.

[Wer20] Max Jonas Werner. *Managing Thousands of Clusters and Their Workloads with Flux*. May 19, 2020. URL: https://gitopsconeu22.sched.com/event/zrqu/managing-thousands-of-clusters-and-their-workloads-with-flux-max-jonas-werner-d2iq.

[Alt22] Altan Altundemir. *Demystifying GitOps-Bootstrapping Flux*. Apr. 19, 2022. URL: https://medium.com/@aaltundemir/demystifying-gitops-bootstrapping-flux-fbfee94f8188.

[BW22] John Belamaric and Stephen Wong. *Nephio Intro and Pre-Release Demo*. Dec. 15, 2022. URL: https://www.youtube.com/watch?v=Reu03R1BwcE.

[Car22] Scott Carey. *Google gets serious about Gitops*. May 25, 2022. URL: https://www.infoworld.com/article/3661366/google-gets-serious-about-gitops.html.

[Cil22] Cilium. *Cilium GitHub*. Dec. 19, 2022. URL: https://github.com/cilium.

[dho22] dholbach. *Flux is a CNCF Graduated project*. Nov. 30, 2022. URL: https://fluxcd.io/blog/2022/11/flux-is-a-cncf-graduated-project.

[Fou22] The Linux Foundation. *The Linux Foundation and Google Cloud Launch Nephio to Enable and Simplify Cloud Native Automation of Telecom Network Functions*. Apr. 12, 2022. URL: https://www.linuxfoundation.org/press/press-release/the-linux-foundation-and-google-cloud-launch-nephio-to-enable-and-simplify-cloud-native-automation-of-telecom-network-functions.

[Har22] Linda Hardesty. *Linux Foundation, Google Cloud launch Nephio to automate 5G edge sites*. Apr. 12, 2022. URL: https://www.fiercewireless.com/5g/linux-foundation-google-cloud-launch-nephio-automate-5g-edge-sites.

[Iso22] Isovalent. *eBPF-based networking, security, and observability*. Dec. 19, 2022. URL: https://isovalent.com/.

[Kum22] Pradeep Kumar. *How to Install Kubernetes Cluster on Ubuntu 22.04*. July 25, 2022. URL: https://www.linuxtechi.com/install-kubernetes-on-ubuntu-22-04/.

[May22] Dan Mayer. *Google, Linux Foundation Launch Nephio to Automate 5G*. Apr. 12, 2022. URL: https://www.sdxcentral.com/articles/news/google-linux-foundation-launch-nephio-to-automate-5g/2022/04/.

[Nep22a] Nephio. *nephio-configsync*. Nov. 1, 2022. URL: https://github.com/nephio-project/nephio-packages/tree/main/nephio-configsync.

[Nep22b] Nephio. *nephio-project*. Nov. 1, 2022. URL: https://github.com/nephio-project/nephio-packages/blob/main/nephio-configsync/rootsync.yaml.

[SDE22] L. Schlunegger, L. Dina, and M. Eberhard. "Cilium CNI Plugins for Kubernetes Cluster Networking". unpublished. 2022.

[Sha22] Sandeep Sharma. *Nephio Technical Overview Video*. Sept. 20, 2022. URL: https://www.aarnanetworks.com/post/nephio-technical-overview-video.

[Wer22] Max Jonas Werner. *Flux multi-cluster example repository*. May 17, 2022. URL: https://github.com/makkes/flux-mc-control-plane.

[Aut23] Helm Authors. *Installing Helm*. Jan. 1, 2023. URL: https://helm.sh/docs/intro/install/.

[Bel23] Bell-Canada. *Network Coverage Map*. Apr. 18, 2023. URL: https://www.bell.ca/Mobility/Our_network_coverage.

[Cis23] Cisco. *Configuring SRv6*. May 6, 2023. URL: https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus9000/sw/93x/srv6/Configuration/guide/cisco-nexus-9000-series-nx-os-srv6-configuration-guide-93x/b-cisco-nexus-9000-series-nx-os-srv6-configuration-guide-93x_chapter_002.pdf.

[Com23a] Ansible Network Community. *Ansible Cisco.Ios Collection*. June 1, 2023. URL: https://docs.ansible.com/ansible/latest/collections/cisco/ios/index.html.

[Com23b] Ansible Network Community. *Ansible Cisco.Iosxr Collection*. June 1, 2023. URL: https://docs.ansible.com/ansible/latest/collections/cisco/iosxr/index.html.

[Com23c] Cloud Native Cloud Computing. *Projects - Flux*. Mar. 15, 2023. URL: https://www.cncf.io/projects/flux.

[Flu23] FluxCD. *Flux v2*. May 2, 2023. URL: https://github.com/fluxcd/flux2.

[git23] gitlab.com. *What is GitOps?* Mar. 1, 2023. URL: https://about.gitlab.com/topics/gitops/.

[Inc23a] GitLab Inc. *GitLab Pipeline architecture*. June 14, 2023. URL: https://docs.gitlab.com/ee/ci/pipelines/downstream_pipelines.html#parent-child-pipelines.

[Inc23b] OpenVPN Inc. *OpenVPN Access Server*. Feb. 23, 2023. URL: https://console.cloud.google.com/marketplace/product/openvpn-access-server-200800/openvpn-access-server.

[Kam23] Kamatera. *Kamatera.com*. June 15, 2023. URL: https://www.kamatera.com/.

[KPT23a] KPT. *KPT Automate Kubernetes Configuration Editing*. June 15, 2023. URL: https://github.com/GoogleContainerTools/kpt/.

[KPT23b] KPT. *KPT Backstage Plugins*. June 15, 2023. URL: https://github.com/GoogleContainerTools/kpt-backstage-plugins.

[KPT23c] KPT. *KPT CLI Installation*. June 15, 2023. URL: https://kpt.dev/installation/kpt-cli.

[KPT23d] KPT. *KPT Concepts*. Apr. 9, 2023. URL: https://kpt.dev/book/02-concepts/.

[KPT23e] KPT. *KPT Config Sync*. June 15, 2023. URL: https://kpt.dev/gitops/configsync/.

[KPT23f] KPT. *KPT Declarative Function Execution*. Apr. 9, 2023. URL: https://kpt.dev/book/04-using-functions/01-declarative-function-execution.

[kpt23] kpt. *The Rationale behind kpt*. Mar. 15, 2023. URL: https://kpt.dev/guides/rationale.

[kub23a] kubernetes. *Installing Kubeadm*. May 24, 2023. URL: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/.

[kub23b] kubespray-opensource-community. *eBPF-based networking, security, and observability*. June 14, 2023. URL: https://kubespray.io/logo/logo-clear.png.

[Lab23] Bitnami Labs. *Bitnami Labs - sealed secrets repository*. June 14, 2023. URL: https://github.com/bitnami-labs/sealed-secrets.

[lin23] Author by linuxize.com. *TCPdump command in linux*. June 14, 2023. URL: https://linuxize.com/post/tcpdump-command-in-linux/.

[Met23] Laurent Metzger. "SRv6 Part 1". Mar. 16, 2023.

[Nep23a] Nephio. *About Nephio*. Mar. 15, 2023. URL: https://nephio.org/about.

[Nep23b] Nephio. *Nephio Proof-of-Conecpt*. Mar. 23, 2023. URL: https://github.com/nephio-project/nephio-poc.

[Nep23c] Nephio. *Nephio: Cloud Native Network Automation*. Mar. 15, 2023. URL: https://nephio.org.

[PN23] Leandro Lisboa Penz and Ganesh Nalawade. *ansible.netcommon.netconfconfig module - netconf device configuration*. June 1, 2023. URL: https://docs.ansible.com/ansible/latest/collections/ansible/netcommon/netconf_config_module.html.

[Sah23] Sahid. *[ERROR CRI]: container runtime is not running [Solved]*. May 11, 2023. URL: https://k21academy.com/docker-kubernetes/container-runtime-is-not-running/.

[Ser23] OpenVPN Access Server. *Limited IPv6 support built into the Access Server*. June 4, 2023. URL: https://openvpn.net/vpn-server-resources/limited-ipv6-support-built-into-the-access-server/.

[SIG23] Kubespray SIG. *Website Kubespray*. June 16, 2023. URL: https://kubespray.io/#/.

[Viv23] Author of online article Vivek Gite. *SSHPASS instructions*. June 14, 2023. URL: https://www.cyberciti.biz/faq/noninteractive-shell-script-ssh-password-provider/.

[VmW23] Bitnami by VmWare. *Sealed Secrets*. June 15, 2023. URL: https://bitnami.com/stack/sealed-secrets.

[Wea23] Weaveworks. *GitHub Weave GitOps*. June 15, 2023. URL: https://github.com/weaveworks/weave-gitops.

CHAPTER E

Glossary

ACK AWS Controllers for Kubernetes. 20, 24, 26, 27, *Glossary:* ACK

ACK AWS Controllers for Kubernetes (ACK) is a tool that lets operators directly manage AWS services from Kubernetes using YAML files. 20

Ansible Ansible is an open source automation platform that enables automation of a whole network of computers at once. 15, 16, 19, 20, 22, 27, 38–41, 44, 50, 52–54, 56, 57, 69, 70, 74, 77–80, 87, 88, 91

API Application Programming Interface. 19, 23, 25–28, 54, *Glossary:* API

API An application programming interface (API) is a way for two or more computer programs to communicate with each other. 19

AS An Autonomous System (AS) is a collection of connected Internet Protocol (IP) routing prefixes under the control of one or more network operators on behalf of a single administrative entity or domain that presents a common and clearly defined routing policy to the Internet. 39

BGP Border Gateway Protocol (BGP) is a standardized exterior gateway protocol designed to exchange routing and reachability information among autonomous systems (AS) on the Internet. 16

CE Customer Edge Router (CE) connect the directly attached local network of a customer to the provider backbones PE router. 11, 16

CI/CD Continuous Integration and Continuous Delivery(CI/CD). Nowadays used to automate release and testing workflows during software development using pipelines. 56, 74

CIDR Classless Inter-Domain Routing (CIDR) Network address space without class rules used to dimension and plan networks. 33

Cilium Cilium is open software to provide cloud-native connectivity between containers with eBPF. 8, 16, 18, 37, 38, 43–46, 50, 52, 56–59, 67, 69, 70, 74–77, 79–81, 84, 85, 87

CldOps Cloud-Operations(module at university). *Glossary:* CldOps

CLI Command Line Interface. 23, 26, 28, 29, 52, 55, *Glossary:* CLI

CLI Command Line Interface (CLI). A CLI is a text-based user interface (UI) used to run programs, manage computer files and interact with the computer. 23

CNCF Cloud Native Computing Foundation. 15, 28, 31, 37, *Glossary:* CNCF

CNCF Cloud Native Computing Foundation (CNCF). CNCF is the open source, vendor-neutral hub of cloud native computing, hosting projects like Kubernetes and Prometheus to make cloud native universal and sustainable. 15

CNI plugin The Container Network Interface (CNI) plugin consists of a specification and libraries for writing plugins to configure network interfaces in Linux containers. 15

Config Sync Config Sync is a GitOps service that lets cluster operators deploy configurations from a source of truth such as Git repositories. 24, 26, 27, 52, 53, 55, 64, 73, 74, 77

Containerd Container runtime daemon (ContainerD) is a process that manages the lifecycle of containers, acting as a container runtime. Containerd does not offer. 38, 40

CPU Central Processing Unit. 81, *Glossary:* CPU

CPU A Central Processing Unit (CPU), is the most important processor in a given computer. 81

CRUD Create Read Update Delete. 26, 54, *Glossary:* CRUD

CRUD Create Read Update Delete (CRUD) describe the four essential operations for creating and managing persistent data elements. 26

Docker Docker is a platform allowing for OS-Level virtualization. 19

dual stack An approach to support both IPv4 and IPv6.. 48

eBPF extended Berkeley Packet Filter. 37, *Glossary:* eBPF

eBPF Extended Berkeley Packet Filter (eBPF), based on Linux Kernels. Improves Security, enables for better monitoring, tracing, and networking. 37

EC2 Elastic Compute Cloud. *Glossary:* EC2

EKS Elastic Kubernetes Service. *Glossary:* Elastic Kubernetes Service

Elastic Kubernetes Service Amazon EKS is a managed service to run Kubernetes on the cloud. 27

FluxCD FluxCD is a tool for keeping Kubernetes clusters in sync with sources of configuration (like Git repositories), and automating updates to configuration when there is new code to deploy. 15, 17, 21, 23, 28–32, 73, 77, 82

GCP Google Cloud Platform. 16, 67, 74

GIT GIT is an open-source version control tool. Used for branching and tracking the history of the source code or data changed. 23, 26, 28–31, 50

GitHub GitHub Inc. is an Internet hosting service for software development and version control using Git. It provides the distributed version control of Git plus access control, bug tracking, software feature requests, task management, continuous integration, and wikis for every project. 15, 28, 54, 55, 60, 74, 77, 82, 86, 87, 94, 99

GitLab The Git Lab Platform for Version Control with Git. Provided by GitLab or the University itself. 15, 38, 41, 44, 56, 69, 74, 78, 79, 82, 83, 86, 93, 94

GitOps GitOps is a delivery approach that uses Git as single source of truth for infrastructure and application deployment. 15, 16, 23, 28, 30, 50, 52, 73

Helm Helm is a package manager for Kubernetes. Helps manage Kubernetes applications by using charts instead of single manifests. 15, 23, 25, 28–31, 37, 56, 57, 74, 102

HTTP Hypertext Transfer Protocol. *Glossary:* HTTP

HTTP POST POST is a HTTP request used to send data to a server to create or update a resource. 54

IaC Infrastructure as Code. 23, 24, 79, *Glossary:* IaC

IaC Infrastructure as Code (IaC) is the provisioning and managing of infrastructure through code instead of through manual processes. 23

IGP An interior gateway protocol (IGP) or Interior routing protocol is a type of routing protocol used for exchanging routing table information between gateways (commonly routers) within an autonomous system (for example, a system of corporate local area networks). 81

INS Institute for Networked Solutions (INS) in Rapperswil at OST within the department of Computer Science. 8

IP The Internet Protocol (IP) is the communications protocol on the network layer in the Internet protocol suite to transfer data across networks. 48

IPSec IPSec is a widely used technology to protect the payload of IP packages while in transport across public (or sometimes internal) networks. 37

IPv4 Internet Protocol version 4 (IPv4) is the fourth version of the Internet Protocol (IP). It is one of the core protocols of standards-based internet working methods in the Internet and other packet-switched networks. 33

IPv6 Internet Protocol version 6 (IPv6) is the most recent version of the Internet Protocol (IP), the communications protocol that provides an identification and location system for computers on networks and routes traffic across the Internet. 33

IS-IS The Intermediate System to Intermediate System (IS-IS) routing protocol is an Interior Gateway Protocol (IGP) standardized by the Internet Engineering Task Force (IETF) and commonly used in large Service Provider networks. 81

Isovalent Company behind Cilium, developing in Switzerland and the US. 8, 37

ISP An Internet service provider (ISP) is an organization that provides services for accessing, using, or participating in the Internet. 13

Jinja2 Jinja2 is a templating engine for Python that generates dynamic content in YAML files. 40, 57

KPT Kubernetes Package Tool. 24–27, 31, 32, 52, 54, 55, 59, 60, 73, 74, 86, 87, 99, 100, 102, *Glossary:* KPT

KPT Kubernetes Package Tool (KPT) is a toolchain for deploying Kubernetes Resource Models in the manner of WYSIWYG and supports configuration authoring and automation. 24

KRM Kubernetes Resource Model. 15, 25, 27, 28, 86, *Glossary:* KRM

KRM Kubernetes Resource Model (KRM) is a practice of defining and managing resources using declarative configuration files to enable efficient and automated management of Kubernetes resources. 15

Kubeadm Kubernetes Admin (Kubeadm) is a Kubernetes installation and setup tool. Performs setup for a cluster creation without a complex configuration. 38

Kubectl Kubernetes Control (Kubectl) is a command-line interface for cluster interaction. Allows running commands against Kubernetes clusters to view and manage resources. 23, 79

Kubelet The Kubernetes node agent registers the node with the API server of the control plane and manages pods for the given node when instructed to do so. Kubelet is a connecting interface. 20

Kubernetes Kubernetes, also known as K8s, is an orchestration platform for managing deployment and scaling of containers. 13–16, 18, 20–24, 26–34, 37–41, 43–45, 50, 54–59, 67, 69, 70, 73–79, 82–87, 102

Kubespray Kubespray is a tool that automates the deployment and management of Kubernetes clusters on-premises or in the cloud using Ansible. 15, 20, 22, 27, 38–41, 44, 56, 69, 70, 73, 74, 77–79, 83, 84

Kustomization Kustomization is the concept of defining and managing customizations and overlays applied to a base Kubernetes configuration using a kustomization.yaml file. 28, 30

Kustomize Kustomize is a Kubernetes-native tool that allows for easy and repeatable customization of Kubernetes resources by applying patches to base configurations. 15, 28–31

L3VPN Layer 3 VPN (L3VPN) provide "private" package transport infrastructure through a dedicated routing infrastructure only used by this customer / company. 16

LTB Lab topology builder (LTB) is a software to provide virtualized networking/infrastructure for training and project purposes to students. 15

manifest Kubernetes Manifest are YAML files that describe each component or resource of the deployment and the state wanted in the cluster to be in once applied. 15, 26, 29, 30, 37, 58

NAT Network address translation (NAT) is a method of mapping an IP address space into another by modifying network address information in the IP header of packets while they are in transit across a traffic routing device. 33

Nephio Nephio is a Kubernetes-based intent-driven automation of network functions and the underlying infrastructure that supports those functions. 15–17, 21, 23–27, 31, 32, 37, 38, 42, 45, 50, 52–56, 59–61, 63, 69, 70, 73, 74, 77, 79, 82, 84, 86, 87, 93, 94, 99, 100, 102

OpenVPN OpenVPN is an open-source VPN software that provides secure remote access and encrypted communication over the internet. 67

OSPF Open Shortest Path First (OSPF). Link State IGP routing protocol. 81

OST OST is the University of Applied Sciences in Eastern Switzerland. 8, 45, 81, 82, 86

P Provider Router (P) are used only within the providers backbone network and sometimes connect to the PE routers. 82

PE Provider Edge Router (PE) connect customer edge routers with the provider backbone network. 36

PoC Proof of Concept (PoC) defines the practical feasability of a given matter. Mostly acquired during a project. 13, 86

Porch Package Orchestration Server (Porch) extends the Kubernetes API server and manages the lifecycle of configuration KRM packages. 24, 26, 27, 64

Rancher Rancher is a cloud-native management platform that allows users to deploy and manage containerized applications across multiple Kubernetes clusters. 20, 39

RD Route Distinguisher. *Glossary:* RD

ReadMe A ReadMe File is a file with a brief describtion and information of the project. Sometimes with initial commands to use the software. 87, 93

RT Route Target. *Glossary:* RT

SA "Semesterarbeit"(German) stands for Term project. 8 ECTS project mandatory for every Computer-Sciences student at OST. 8

SAFI Subsequent Address Family Indicator. *Glossary:* SAFI

SID The Segment ID (SID) identifies a segment in SRv6. 16

SRH The Segment Routing header (SRH) is an extension header added to IPv6 packets to implement Segment Routing IPv6 (SRv6) based on the IPv6 forwarding plane. 34

SRv6 Segment Routing IP-Version 6 (SRv6) is the next-gen routing approach based on routing via segments. 8, 36, 57, 58

SSH Secure Shell Protocol (SSH). SSH is a cryptographic network protocol for connecting and operating on network services securely over an unsecured network. 19

TCPdump Command line tool the show details and connections on the interfaces. 41

TLV Type Length Value. 81, *Glossary:* TLV

TLV The Type Length Value (TLV) makes IS-IS extendable. 81

VM Virtual Machine. 16, 17, 22, 32, 38–40, 45, 67, 69, 73, 74, 78, 79, 83–85, *Glossary:* VM

VM Virtual Machine (VM) is the virtualization or emulation of a computer system. 16

VPC Virtual Private Cloud. 67, 76, *Glossary:* VPC

VPC A VPC is a virtual network that allows users to create and control their own isolated network environment. 67

VPN Virtual Private Network. 16, 36, 37, 45, 46, 58, 67, 76, *Glossary:* VPN

VPN Virtual Private Network (VPN) stands for different approaches to expand given networks. 16

VPNv4 Virtual Private Network Version 4 (VPNv4). BGP IPv4 prefixes of customers that are exchanged between PE routers to provide routing information for reachability in the provider network. 58

VPNv6 Virtual Private Network Version 6 (VPNv6). BGP IPv6 prefixes of customers that are exchanged between PE routers to provide routing information for reachability in the provider network. 58

VXLAN Virtual Extensible LAN. 33, 57, *Glossary:* VXLAN

VXLAN VXLAN is a network encapsulation technique that enables the creation of virtual L2 networks over an existing L3 network, allowing for efficient communication and scalability in distributed systems. 33

WYSIWYG What You See Is What You Get. 24, *Glossary:* WYSIWYG

WYSIWYG What You See Is What You Get (WYSIWYG). WYSIWYG means that the final product will look exactly like how it looks now. Microsoft Word is e example for a WYSIWYG editor. What you see in the editor is identical to the final product and there are no changes from the program on the way. 24

CHAPTER F

Abbreviations

AS Autonomous System. 39, 45, *Glossary:* AS

BA Bachelor Arbeit (German). *Glossary:* BA

BGP Border Gateway Protocol. 16, 36, 37, 45, 50, 56–59, 67, 74, 75, 77, 80, 85, *Glossary:* BGP

CE Customer Edge Router. 11, 59, 76, 82, *Glossary:* CE

CIDR Classless Inter-Domain Routing. 33, 37, 56–59, 75, 76, 87, *Glossary:* CIDR

CNI plugin Container Network Interface plugin. 15, 16, 20, 22, 33, 37, 38, 43, 44, 46, 56, 74, 77, 81, 87, 97, *Glossary:* CNI plugin

CRD Custom Resource Definition. *Glossary:* CRD

eBGP external Border Gateway Protocol. *Glossary:* eBGP

FIB Forwarding Information Base. *Glossary:* FIB

iBGP internal Border Gateway Protocol. *Glossary:* iBGP

IGP Interior Gateway Protocol. 81, *Glossary:* IGP

INS Institute for Network and Security. 8, 18, 38, 39, 45, 69, 80, *Glossary:* INS

IP Internet Protocol. 48, 81, *Glossary:* IP

IPSec Internet Protocol Security. 37, *Glossary:* IPSec

IPv4 Internet Protocol version 4. 33, 36, 39, 40, 44, 46, 48, 56–58, 67, 95, *Glossary:* IPv4

IPv6 Internet Protocol version 6. 33, 34, 36, 37, 39, 44, 46, 48, 56–58, 67, 76, 95, *Glossary:* IPv6

IS-IS Intermediate System to Intermediate System. 81, *Glossary:* IS-IS

ISP Internet Service Provider. 13, 14, 77, *Glossary:* ISP

L3VPN Layer 3 Virtual Private Network. 16, 36–38, 56–59, 67, 74, 75, 82, 85, *Glossary:* L3VPN

LDP Label Distribution Protocol. *Glossary:* LDP

LTB Lab Topology Builder. 15–17, 38–40, 45, 67, 69, 74, 76, 77, 80, 82–84, 90, 101, *Glossary:* LTB

MPLS Multiprotocol Label Switching. *Glossary:* MPLS

NAT Network Address Translation. 33, 57, *Glossary:* NAT

NET Network Entity Title. *Glossary:* NET

NSAP Network Service Access Point Address. *Glossary:* NSAP

NSEL Network-Selector. *Glossary:* NSEL

OSI Open Systems Interconnection. *Glossary:* OSI

OSPF Open Shortest Path First. 81, *Glossary:* OSPF

P Provider Router. 82, *Glossary:* P

PE Prover Edge Router. 36, 37, 58, 59, 76, 82, 85, *Glossary:* PE

PoC Proof of Concept. 13, 15, 17, 27, 31, 38, 54, 69, 73, 77, 78, 82, 86, 99, *Glossary:* PoC

RIB Routing Information Base. *Glossary:* RIB

RR Route Reflector. *Glossary:* RR

RUP Rational Unified Process. *Glossary:* RUP

SA Semesterarbeit (German). 8, *Glossary:* SA

SCRUM SCRUM. *Glossary:* SCRUM

SDN Software Defined Networking. *Glossary:* SDN

SID Segment ID. 16, 34, 36, 37, 57–59, 75, 76, *Glossary:* SID

SLAAC Stateless Address Autoconfiguration. *Glossary:* SLAAC

SR Segment Routing. *Glossary:* SR

SRH Segment Routing Header. 34–36, *Glossary:* SRH

SRv6 Segment Routing version 6. 8, 16, 34–37, 39, 44–46, 50, 56–59, 61, 65, 67, 75–77, 80–82, 85, *Glossary:* SRv6

SSH Secure Shell Protocol. 19, 38–41, 45, 69, 101, *Glossary:* SSH

VPNv4 Virtual Private Network version 4. 58, 67, 75, *Glossary:* VPNv4

VPNv6 Virtual Private Network version 6. 58, 67, *Glossary:* VPNv6

VRF Virtual Routing and Forwarding. 16, 37, 58, 59, 65, 75, 76, 88, 90, 91, *Glossary:* VRF