POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Automation and provisioning of Kubernetes on bare-metal Telco edge infrastructures

Supervisors
Prof. Fulvio RISSO
Mr. Stefano GALANTINO

Candidate
Francesco BRUNO

July 2024

Summary

Cloud computing has revolutionized the deployment of innovative services, with cloud-native applications becoming a fundamental design pattern. Kubernetes has emerged as the leading platform for container and application management in this paradigm. For telecommunication companies (Telcos) with numerous edge sites, adopting cloud-native solutions offers the potential for rapid deployment, configuration, and maintenance of network applications, while ensuring a robust certification process.

However, this shift requires a certified, modular, and extensible framework to manage multiple edge sites along with their associated Network Functions and services. The Sylva Project aims to provide such a framework, offering a reference implementation based on open-source software tailored to European Telcos' needs. Sylva seeks to automate and standardize the traditionally manual and fragmented process of deploying and managing edge sites with bare-metal or virtualized nodes.

The automation of bare-metal Sylva deployments presents a unique challenge, requiring deep, domain-specific knowledge of the underlying infrastructure. This thesis proposes an architecture based on the Server Operator and its related Custom Resource Definitions (CRDs).

An important component of this solution is to seamlessly integrate within the existing stack, while following the patterns and standards of Kubernetes and the tools used within Sylva. This is achieved by extending Sylva's current operators to simplify the configuration of bare-metal nodes. By doing so, the proposed architecture aims to enhance collaboration between infrastructure administrators and application developers, ultimately increasing deployment agility in Telco environments.

Acknowledgements

Ringrazio i miei genitori e la mia splendida Famiglia, mi avete sempre sostenuto e senza di voi niente di tutto questo sarebbe stato possibile.

Una ringraziamento particolare va a Laura, mia compagna di vita. Giorno dopo giorno mi hai spronato per andare avanti, guardando sempre al futuro. Non ce l'avrei fatta senza di te.

Ringrazio i miei amici Fabrizio, Rossana, Gabriele, Federica, ormai ci siete da sempre e sebbene a volte dobbiate faticare per starmi dietro, non mollate mai.

Ringrazio Elia, Lorenzo, Gabriele, in questi anni non abbiamo perso l'occasione per incontrarci fuori e dentro il Poli.

Ringrazio i colleghi in Tim, Carlo, Federico, Alessandro, Roberto, Raffaele e Teodoro per avermi seguito e aiutato durante il lavoro di tesi.

Ringrazio il prof. Risso per la sua disponibilitá e prontezza nel rispondere a qualsiasi domanda o problema.

Un ringraziamento a tutti coloro che non ho menzionato direttamente, ogni piccolo passo insieme a voi mi ha fatto andare avanti.

Acronyms

Test
: Telecommunication company

CRD
: Custom Resource Definition

NF
: Network Functions

K8s
: Kubernetes

SWC
: Sylva Workload Cluster

VM
: Virtual Machine

CaaS
: Container as a Service

CRI
: Container Runtime Interface

CRUD
: Create, Read, Update, Delete

DNS
: Domain Name System

SDK
: Software Development Kit

CLI
: Command Line Interface

DC

Data Center

AR/VR

Augmenter Reality / Virtual reality

RAN

Radio Access Network

CAPEX

Capital Expenditures

CNF

Cloud-native Network Function

SUR

SylvaUnitsRelease

SURT

SylvaUnotsReleaseTemplate

OCI

Open Container Initiative

CAPI

ClusterAPI

IaaS

Infratructure as a Service

BMH

BareMetalHost

BMC

Bare Metal Controller

Contents

Chapter 1

Introduction

1.1 Introduction

The telecommunications industry is undergoing a significant transformation, driven by the widespread adoption of cloud-native technologies and the increasing demand for edge computing capabilities. This shift presents both opportunities and challenges for telecom operators (telcos) as they strive to modernize their infrastructure and service delivery models.

Cloud-native applications, characterized by their scalability, resilience, and flexibility, are becoming the norm in modern software development. For telcos, this trend manifests in the growing need to efficiently manage an expanding number of edge sites and network functions.

The edge computing paradigm, which brings computation and data storage closer to the end-users, is particularly crucial for telcos due to their distributed nature and territorial presence. It enables them to offer ultra-low latency services, process data locally, ensure privacy, and maintain limited autonomy in case of disconnection from central data centers.

However, the adoption of cloud-native technologies in the telco industry is not without its challenges. One of the most significant hurdles is the fragmentation of infrastructure and platforms. Telcos often find themselves managing multiple separate infrastructures, each tailored to specific applications or network functions. This fragmentation leads to increased operational complexity, inefficient resource utilization, and difficulties in certifying applications across different cloud platforms.

The Sylva project, an initiative under the Linux Foundation Europe, aims to address these challenges by establishing a common, open-source cloud-native infrastructure stack for the telecommunications industry. Sylva provides a crucial reference framework for European telcos and vendors, promoting interoperability and standardization. However, despite its advantages, Sylva still faces several challenges, particularly in managing the inherent complexities of bare metal environments.

One critical issue is the substantial domain-specific knowledge required for the effective deployment of new bare metal Workload Clusters. This complexity often

creates a barrier between infrastructure administrators and application developers, hindering efficient collaboration and slowing down the deployment process. The current approach typically involves manual configuration and management of bare metal resources, which is time-consuming, error-prone, and requires specialized expertise.

Given this context, the primary objective of this thesis is to design and implement a robust architecture that streamlines the deployment process of Kubernetes clusters on bare metal instances, specifically focusing on managing domain-specific configurations for the deployment of Sylva Workload Clusters (SWC). The proposed solution aims to bridge the gap between high-level cluster requirements and low-level bare metal configurations, enabling a more seamless and automated deployment process.

To achieve this goal, we leverage Kubernetes closed-loop reconciliation patterns, implementing a custom controller and a Custom Resource Definition (CRD). This approach allows us to extend the existing Sylva framework, providing a more user-friendly and efficient method for deploying and managing bare metal clusters. The solution is designed to integrate seamlessly with Sylva's declarative and GitOps-driven workflow, maintaining compatibility with existing operators while offering enhanced functionality for bare metal deployments.

By addressing these challenges, this thesis contributes to the broader evolution and enhancement of the Sylva project, facilitating the adoption of cloud-native technologies in telco environments and supporting the industry's digital transformation journey.

1.2 Goal

The primary objective of this thesis is to design and implement a robust architecture that streamlines the deployment process of Kubernetes clusters on bare metal instances. Specifically, the focus is on managing domain-specific configurations for the deployment of Sylva Workload Clusters (SWC). This challenging task is accomplished through the implementation of Kubernetes closed-loop reconciliation patterns, which involve a custom controller and a Custom Resource Definition (CRD). The proposed solution is carefully crafted to integrate seamlessly with the existing Sylva project ecosystem and its established operators. By leveraging these advanced Kubernetes concepts, the architecture aims to enhance the efficiency and reliability of cluster deployments, ultimately contributing to more manageable and scalable infrastructure solutions in complex bare metal environments.

1.3 Structure of the thesis

This thesis is composed of the following chapters:

- **Chapter 2** will provide an overview of Kubernetes the core technology that enables The Sylva project and the Cloud Native approach.

- **Chapter 3** focuses on the Sylva project itself, its goals and architecture.
- **Chapter 4** will give an overview of some of the technologies that allow Sylva to operate in a bare metal environment.
- **Chapter 5** introduces the work of the thesis, the faced challenges and proposed solution.
- **Chapter 6** explains the practical implementation of the proposed CRD and controller.
- **Chapter 7** provides the achieved results and performance obtained.
- **Chapter 8** presents challenges of the current approach and possible future work.
- **Chapter 9** discusses the conclusions of the work carried out.

Chapter 2

Kubernetes

This chapter provides an overview of the Kubernetes architecture and its fundamentals, as the Sylva project and the subject of this thesis are based upon its model. Kubernetes, often abbreviated to k8s, is a large open-source system for automating deployment, scaling, and management of containerized applications [1].

We will give an overlook of the general concepts and components and how they work together. Particular focus is going to be given to its extensibility and versatility, by exploring Custom Resource Definitions and the Operator framework. These powerful tools allow Kubernetes to evolve beyond the out-of-the-box functionality, catering to diverse and specialized application requirements, such as the particular needs of telcos.

We will also spend some time to understand Kubebuilder, a tool used to scaffold these custom resources, with which the implementation of this work was built

2.1 Cloud Native Evolution

The way applications are developed has evolved substantially over time, reflecting the shift in technological capabilities and business requirements.

Traditionally, applications were built to run on a physical server, in this so-called 'bare metal' approach each application was tightly coupled with its hardware, environment and had virtually no way of defining resource constraints, forcing system administrators to deploy one or few applications per server, causing underutilization of server resources, increasing cost and maintenance work. Scalability was also a concern, as applications would rarely be designed to scale horizontally, leaving vertical scaling as the primary option, an increasingly costly option. Even when horizontal scalability was possible, augmenting more servers was an expensive proposition and required considerable time and effort.

Virtualization Technology emerged as possible solution to these challenges. Virtualization allows for multiple virtual machines to run concurrently on a single server. This grants isolation of the applications running on a physical server, allows for better control over resource limits and provides better overall resource utilization compared to the bare metal approach. Creating and managing VMs is

an easy task, a VM can be scaled up or down as needed and can run on uniform hardware, reducing operational complexity, hardware cost and much more.

For all its advantages, Virtualization still introduces significant overhead, as each VM includes its operating system, virtualized devices and related resources.

Containerization is the next step in the evolution of workload deployment. Containers, in a way, work similarly to Virtual Machines, but they share the same host operating system, this allows for virtually zero runtime overhead[2], at the cost of relaxed isolation and security properties.

One key feature of containers is their decoupled nature and thus portability, they encapsulate an application along with its dependencies into a standalone unit that can operate anywhere, offering a highly portable, environment-agnostic solution for software deployment.

2.1.1 Container Orchestration

Despite the advantages of containers, managing a large number of them across different servers presents new challenges, particularly around scheduling, networking, and scalability. **Kubernetes** was introduced to address these issues, providing a robust platform for orchestrating containers.

K8s automates the deployment, scaling, and management of containerized applications, and provides other essential tools such as Service Discovery and Load Balancing, Storage Orchestration, Secrets and Configuration Management, Automated rollouts and Rollbacks[4].

One of the primary advantages of Cloud Native infrastructure technologies, which makes use of Kubernetes, is the modular, interoperable and scalable approach to application deployment. Kubernetes provides an open platform called CaaS (Container as a Service) on top of which applications can be deployed consistently across locations.

2.2 Architecture

Kubernetes is based around a modular and highly extensible architecture in which we can identify two main components, a Control Plane, which is the source of truth for the state of the cluster and is responsible for managing and maintaining the desired state of the cluster, and Worker Nodes, which run the actual containerized applications and workloads.

2.2.1 Nodes

Nodes are the actual physical or virtual machines where applications run. Each node is a self-contained runtime environment containing the necessary services to support running containers.

These services include [5]:

- **Kubelet** An agent that resides in each node in the cluster. It's responsible for maintaining the state of the node and ensuring that all the components and applications within the node are running as expected. It communicates with the control plane to receive commands and report back.
- **Kube-Proxy** Serves as network proxy and load balancer for a node, it implements part of the Kubernetes Service concept. Allows network communication from inside or outside of the cluster.
- **Container Runtime** A fundamental component that allows Kubernetes to run containers effectively. Kubernetes supports many container runtimes, such as containerd, CRI-O and any other implementation of Kubernetes CRI.

Typically, many nodes constitute a cluster, however, even a single node can fulfill both roles, of control-plane and worker, thus forming a Kubernetes cluster.

2.2.2 Control Plane

The term is used to identify the set of nodes running control plane components. These make global decisions about the cluster, as well as detecting and responding to cluster events. Some of the main components include:

- **API Server** It exposes the Kubernetes API and acts as the front end of the Kubernetes control plane. It serves as the communication hub with which both internal and external components perform various operations.
- **etcd**Is a consistent and highly available key-value store, used as Kubernetes' backing store for all cluster data. An external database may be used instead.
- **Scheduler** The scheduler is responsible for scheduling pods (the smallest deployable unit of computing in Kubernetes), on nodes. It continually keeps the state of the cluster consistent taking into account resource requirements, hardware/software/policy constraints , data locality and user specifications.
- **Controller Manager** It runs various controllers, control loops which are responsible for reconciling the state of the cluster with the desired one. Such as the Node Controller, responsible for noticing and responding when a node goes down.

2.2.3 Objects

One fundamental aspect of Kubernetes, and central to its whole operation, is that of Objects. They are persistent entities used to represent the state of the cluster. An object is a "record-of-intent" which describes the desired resource[6]. The control plane will continually work to ensure the object exists and matches the provided specification.

Almost all Kubernetes objects includes two nested object fields that govern the object's configuration: the object **spec** and the object **status**. The spec of an

object must be set at creation time, providing a description of the characteristics the object must have, its desired state.

The status describes the current state of the object, it is supplied and managed by the Kubernetes system and its components. The control plane continually and actively manages every object's state to match the one specified in the spec.

On top of these parameters, to describe a Kubernetes object some other basic information needs to be provided:

- *ApiVersion* Identifies the specific version of the schema representing this object.
- *Kind* A string that identifies the REST resource for the object.
- *Metadata* Other information such as its name, annotations, label and an optional namespace

Trough the API Server we can interact with these resources in a typical CRUD (create, read, update, delete) fashion. The API may be accessed by internal components, such as controllers, or by external tools, like Kubectl.

Within the metadata of an object, we need to make a distinction between two fields that initially may seem very similar:

- **Labels** Key-value pairs containing information and are used to identify and Group objects into subsets by defining and organizing them. Objects can then be filtered and indexed based on labels.
- **Annotations** Extra key-value pairs attached to an object, not usable to select an object, unstructured, that can be used by clients and third-party tools. Can contain lightweight configurations.

In both cases the key of the metadata can optionally contain a prefix that must be in the format of a DNS subdomain name as defined in RFC-1123.

Listing 2.1. Example object metadata

```
1 metadata:
2   name: label-demo
3   labels:
4     environment: production
5     app: nginx
6   annotations:
7       imageregistry: 'https://hub.docker.com/'
```

Listing 2.2. Example Kubernetes Object

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: nginx-deployment
5 spec:
```

```
6    selector:
7       matchLabels:
8          app: nginx
9    replicas: 2 # tells deployment to run 2 pods matching the template
10   template:
11      metadata:
12         labels:
13            app: nginx
14      spec:
15         containers:
16            - name: nginx
17              image: nginx:1.14.2
18              ports:
19              - containerPort: 80
```

2.3 Namespaces

In Kubernetes, namespaces provide a mechanism for isolating groups of resources within the cluster. They act as virtual partitions providing a scope for names, meaning resource names only need to be unique within their namespace, not across the entire cluster. Each Kubernetes resource can only belong to one namespace, however It's important to note that namespace-based scoping is only applicable for namespaced objects, like Deployments and Services. Cluster-scoped objects like Nodes and PersistentVolumes do not exist within namespaces.

2.4 Helm

Kubernetes offers powerful and extensible tools for managing applications, however, as the size and complexity of an application grows so does its required configuration. In order to manage this complexity in a reusable way we can use Helm.

Helm Charts are collections of files used to define, install and manage Kubernetes applications. Charts simplify the deployment of applications that range in complexity from a single pod to a complete application with multiple components, networking, volumes and authorization settings Charts are versioned and can be stored in repositories, similar to Linux package managers, to facilitate sharing and reuse.

A Helm chart is based around the idea of templating. Hem Chart templates are written using the Go template language and provide a mechanism for dynamically generating Kubernetes manifests based on configurable values. Charts can be dependent on and make use of Charts, allowing for complex yet modular, versatile and dynamic definitions.

The main components of a Helm Chart are:

Chart.yml A file containing information and metadata about the chart;

Values.yaml A file containing the default configuration values for the chart;

Templates A directory of templates that reference the values to generate valid k8s manifests

2.5 Operator Pattern

Kubernetes was designed with automation in mind, and its core offers a wide array of built-in automation capabilities. One fundamental concept within Kubernetes is the Controller Pattern enabling automation through control loops that monitor and adjust the cluster's state.

Controllers are clients of the Kubernetes API and operate by comparing the desired state, specified in object specifications, with the actual state, fetched from the etcd data store. This pattern allows for continuous reconciliation, ensuring that the cluster's state aligns with user definitions.

The Operator pattern is a software extension mechanism in Kubernetes that uses custom resources to manage applications and their components. It essentially is a specialization of the Controller Pattern building upon custom resources to provide domain-specific automation encapsulating expert knowledge about a particular application or service. The Operator pattern, building upon the Controller pattern, introduces powerful self-healing capabilities to Kubernetes clusters. Self-healing is a crucial aspect of maintaining the health and stability of complex distributed systems, particularly in the context of cloud-native applications.

Self-healing in the Operator pattern works by continuously monitoring the state of custom resources and their associated components. When discrepancies between the desired state (as defined in the custom resource specification) and the actual state of the system are detected, the Operator automatically takes corrective actions. These actions might include restarting failed pods, scaling resources up or down, updating configurations, or even initiating more complex recovery procedures specific to the application domain.

This autonomous operation reduces the need for manual intervention, enhancing system reliability and operational efficiency. By encapsulating domain-specific knowledge and best practices, Operators can implement sophisticated self-healing strategies that go beyond simple restarts, addressing complex failure scenarios and maintaining the overall health of the application ecosystem.

2.5.1 Custom Resource Definitions (CRDs)

Custom Resource Definitions in Kubernetes provide a mechanism to define new Object Kinds, essentially allowing users to define their own API Objects integrating arbitrary configurations and states into Kubernetes' declarative management approach, that behave like native K8s objects. Their entire lifecycle is managed by the Kubernetes API server.

2.5.2 Controllers

As previously mentioned, controllers operate in a control-loop to manage the state of objects. In the context of the Operator pattern there usually is one Controller per CRD, that owns and reconciles the resource.

A Controller receives a filtered stream of events by the API Server to then call a Reconciler. The controller is also responsible for handling back-off, queuing and requeuing of events. These usually signify changes happening within the cluster.

It is common for Controllers to watch for changes to the Resource type that they Reconcile and Resource types of objects they create.

2.5.3 Kubebuilder

Kubebuilder is a SDK for building K8s APIs using CRDs. It greatly reduces the time needed to build an operator, by scaffolding the necessary structure for CRDs and related controllers, taking advantage of a collection of libraries and tools from the Kubernetes ecosystem, Kubebuilder streamlines the development process for Kubernetes APIs.

Kubebuilder encourages good development practices by organizing code into separate packages based on domain logic. Each package encapsulates the API definition and its associated controller, improving code maintainability

Trough the Kubebuilder CLI it is possible to easily generate files and scaffolding for custom CRDs and controllers. It provides facilities to generate CRDs and API objects, controllers, webhooks, manage RBAC permissions and much more.

Chapter 3

Sylva Project

Project Sylva is a collaborative effort within the Linux Foundation Europe focused on establishing a common, open-source cloud-native infrastructure stack for the telecommunications industry. The project seeks to streamline the adoption of cloud technologies in the telecommunications sector by addressing the technical challenges and fragmentation that hinder progress.

3.1 Introduction

In a close parallel to what's been discussed previously, the evolution toward cloud native solutions is particularly important for telecommunication companies and is one of the major technological trends of recent years in the telco industry.

3.1.1 Edge Computing

During recent years the need to increase the proximity between cloud services and end users has clearly emerged. The edge-computing model encourages the deployment of 'mini-cloudlets' close to the end users, catering to different use cases [7]:

- **Ultra Low Latency**
  Measured in $\approx$ ms is critical for real time applications such as Robotics, AR/VR, gaming.
- **Local Processing**
  Local elaboration of produced data to extract information and take decisions, avoiding moving large quantities of data towards a central DC.
- **Privacy**
  Sensible or personal data will remain local without going into the public domain.
- **Limited Autonomy**
  Ability to continue operations when disconnected from a central DC.

Given telco's distributed nature and territorial presence, they are in the best position to implement the cloud and digital communication model of edge computing.

It is increasingly common for cloud services to no longer be accessed solely trough remote data centers, instead they are deployed in multiple locations, providing the same capabilities and latency advantages of small remote cloudlets to the final users. This architecture of cloud evolution can be named "edge/cloud continuum".

As telecommunications and cloud computing industries increasingly converge, their technologies are also aligning. The software technologies and concepts that have driven the commercial success of cloud services are now available for telecommunications companies to leverage and implement in their own operations.

3.1.2 Opportunities

As the need to deploy hundreds of edge cloud nodes will manifest in the coming years, Telcos will be faced with accommodating in the same location traditional Telco functionality, such as 5G RAN functions and modern cloud services. Cloud Native infrastructure provides an opportunity to greatly simplify deployment and increase efficiency., as one computing platform can be used to share computing and connectivity resources among applications.

Examples of applications from the Telco industry are:

- 5G Radio Access Network (RAN) functions
- 5G Mobile Core network functions
- IP Network Functions
- Border Network Gateways

3.1.3 Market Fragmentation

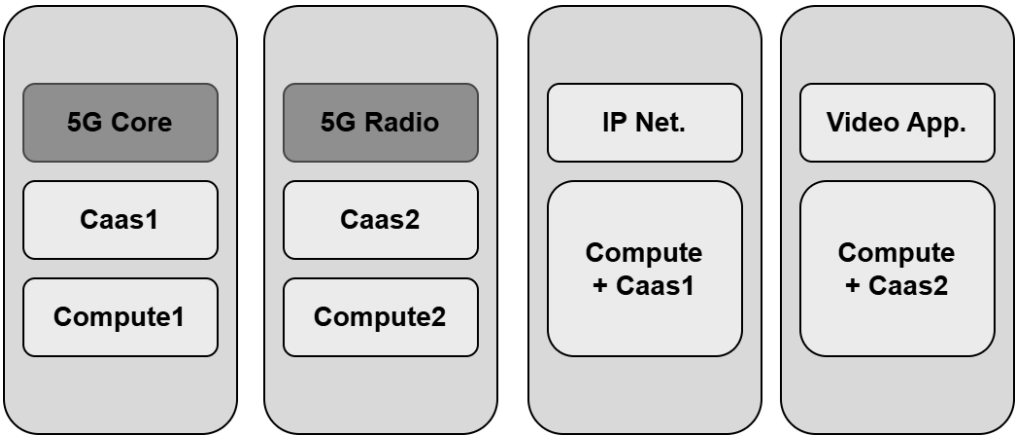

Figure 3.1. Fragmentation example

While the benefits of a cloud-native approach are clear, the Telco industry current deployment model leads to inevitable fragmentation as it mandates the use of proprietary CaaS, sometimes on top of specific physical machines

This inevitably leads to increased complexity for both vendors and operators. The former need to develop their applications, and NF and certify them for multiple cloud layers. While the Latter need to deal with multiple separate infrastructures, leading to increased operational complexity.

This presents several challenges that render the whole approach unsustainable:

- Prevents the possibility of sharing CaaS and physical resources among different applications, resulting in wasted compute power with CAPEX and energy impacts
- Creates unnecessary complexity for vendors as they look to certify multiple cloud platforms
- Creates operational burdens on Telcos by having to support several environments
- Fails to provide a solution that can evolve with the necessary speed of cloud native

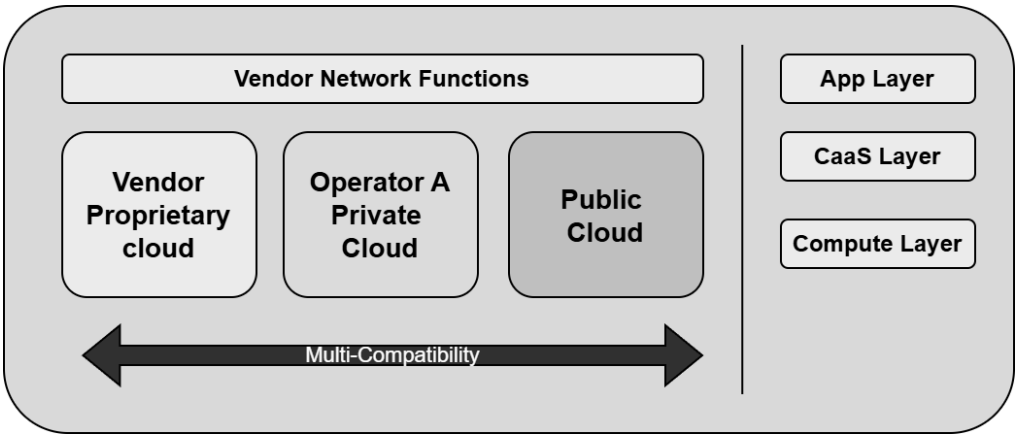

Figure 3.2. Fragmentation from the vendor point of view

3.2 Goals

The first goal of the Sylva Project is thus to converge the cloud layer, creating a common infrastructure among European operators by incorporating capabilities required for a CaaS to handle specific use cases such as 5G, O-RAN, and Edge deployments. benefitting both operators, vendors and ultimately leading to better services.

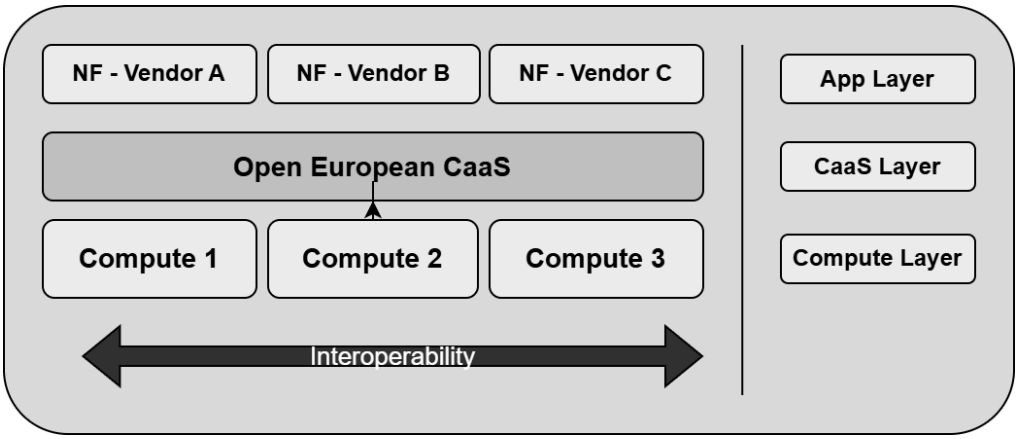

Figure 3.3. Unification goal of the Sylva project

this cloud layer should align with well-established cloud native technologies and operating processes. In particular, the target architecture should be based on multi-cluster Kubernetes automated deployments on Bare Metal, managed with GitOps approach (including Infrastructure as code).

The second goal of the Sylva project is to develop a Reference Implementation of this cloud software framework and create an Integration and Validation program to validate commercial network functions and implementations based on the released framework and its components [8].

3.3 Implementation Details

When analyzing the architecture of the Sylva implementation we can identify three main components, the Management Cluster, Workload Clusters, Units.

Sylva is designed to leverage open source projects and be lightweight enough to house all of the needed components even in a single node.

3.3.1 Management Cluster

As the name suggests, it is a Kubernetes cluster with all of the components needed to deploy, manage and monitor itself and other Sylva Workload Clusters using a declarative approach. Drawing an analogy to the Kubernetes model, it can be seen as the k8s control plane equivalent of a Sylva system. As such, its responsibilities and characteristics are:

- **Centralized Control** The Management Cluster acts as a central point for deploying and managing workload clusters.
- **Declarative Approach**
- **Gitops** The management cluster leverage GitOps tools and workflows to reconcile the desired state with the deployed state. All configuration about

itself or Workload clusters are stored in central repositories, enabling version control, automated deployments and simplified rollbacks.

- **Units** The MC is responsible for deploying and managing Sylva Units within itself and its workload clusters.

3.3.2 Workload Cluster

A Workload Cluster is a full Kubernetes cluster, with its own Control Plane and Worker node. It is managed by the Management Cluster, the specific units (applications or components) deployed to a workload cluster vary depending on the target infrastructure.

Due to Kubernetes's lack of strong multi-tenancy support, critical CNFs requiring high security isolation often necessitate dedicated workload clusters, potentially on separate hardware. This approach ensures better security for sensitive applications and services.

3.3.3 Units

Sylva units are applications or components, which can be either mandatory or optional, that come from open-source projects and make up Sylva clusters. "Units" is used instead of "components" to avoid confusion with the Kustomize terminology also used in Sylva.

To manage the large number of clusters in the Sylva stack, a declarative approach is used for both clusters and units. This means that unit kustomizations and cluster definitions are kept in central repositories that are managed by GitOps tools, which are responsible for reconciling what is declared with what is actually deployed.

Within Sylva, Units are deployed via the `sylva-units` Helm Chart. This chart is not a typical one, while most Helm Charts create base Kubernetes objects to deploy a given service, this chart instanciates Flux resources that tell Flux how to deploy the units. Throughout the chart extensive user of Helm templating is used, allowing greater flexibility. Clear dependencies are specified for all components to avid wasting time and processing resources when the environment is not ready yet.

3.4 Sylva Operators

Thanks to the highly modular design of Sylva, the creation and management of Sylva Clusters is very similar between any of the types of Sylva Clusters, Bootstrap, Management and Workload.

Before the introduction of these operators, the creation and configuration management of SWC and clusters in general was possible trough the usage of Bash scripts and baseline configuration files present within the official Sylva Core repository.

While it is possible to automate their execution, the workflow remained quite manual. The final result of is the creation or modification of a Helm Release containing

all the necessary configuration for creating the cluster. To better align the project with Kubernetes and Cloud Native standards, two operators have been introduced

3.4.1 Sylva Units Operator

The Sylva Units Operator Introduces two CRDs, the SylvaUnitsRelease (SUR) and SylvaUnitsReleaseTemplate (SURT). They are essentially the same CRD, sharing the exact same parameters except for the status, which is not needed and thus omitted from the Template.

Its controller is in charge of keeping a Helm Release of the Sylva Units chart in sync, along with its required resources, such as a Git or OCI repository for the chart source. It will sync the required values and value references from the SUR to the HR. Given Sylva modular design, this operator is capable of generating a HR for all types of Sylva clusters, bootstrap, management and workload. This requires some extra configuration parameters to be given to the HR, which the operator will provide based on the type of requested cluster.

3.4.2 Sylva Workload Cluster Operator

The Sylva Workload Cluster Operator could be considered an extension of Sylva Units Operator, it introduces a new CRD, the SylvaWorkloadCluster, which holds curated configuration parameters to deploy a Sylva workload cluster, such as the k8s version, the number of requested machines for the control plane and worker nodes, the infrastructure provider and some cluster customization parameters.

The SWC resource holds few parameters compared to those required to spin up a Workload Cluster, therefore, the SWC Operator makes use of the SURT to correctly generate a SUR. The SURT role is to hold default configuration values for new Workload Clusters which contain information specific to their parent Management Cluster.

In essence, once a SWC resource is created on the MC, the SWC Operator will generate a SUR, copying over the spec from the SURT and including any configuration fields specified in the SWC Resource and other parameters required by a workload cluster. The SUR Operator, in turn, will generate a HelmRelease with all the components necessary for a successful deployment.

Thus, the process of creating and managing Workload Clusters is greatly simplified. Global values can be shared across all deployments and the creation of new Workload Clusters is as simple as creating one single resource in its own namespace.

Chapter 4

Bare Metal Deployment

To enable the creation and management of Workload Clusters and specifically ones deployed on bare metal infrastructure, Sylva makes use of a couple key technologies.

4.1 ClusterAPI

ClusterAPI (CAPI) is an open-source Kubernetes project that provides declarative APIs and tools for creating, configuring, and managing Kubernetes clusters. Its primary goal is to simplify cluster lifecycle management and enable infrastructure providers to offer Kubernetes as a service[9].

Within the context of the Sylva project, CAPI is a core technology, that enables the creation and management of all types of Sylva clusters.

4.1.1 Key Conecpts

ClusterAPI is composed of several key components:

- **ClusterAPI Provider** This component extends ClusterAPI to manage clusters on a specific infrastructure platform. It allows ClusterAPI to interact with an infrastructure provider using platform-specific implementations.
- **Cluster Controller** It manages the life cycle of cluster resources, handling operations such as creating, scaling and deleting. Ensuring that the desired state of the cluster is maintained.
- **Bootstrap Provider** This provider handles the initialization and bootstrapping of the nodes within the cluster. It provisions necessary resources and configures the nodes to join the desired Kubernetes cluster. Some platforms supported by the bootstrap provider are: Amazon EKS, K3s, Kubeadm, Microk8s, ecc.
- **Infrastructure Provider** It interacts with the underlying infrastructure to provision and manage resources required for the cluster. This includes virtual machines, bare metal servers, containers or any other implementation,

depending on the infrastructure being used. Examples of supported platforms go from big cloud vendors: AWS, Azure, GCP. To enterprise IaaS providers like OpenStack, vSphere, Proxmox. To local environments like Metal3, Docker.

Given Sylva framework's deep reliance on ClusterAPI we can find similarities in the terminology and workflows. A ClusterAPI Management Cluster is a Kubernetes cluster that manages the life cycle of Workload Clusters. One or more providers can run within a Management cluster and it is the location where Machine resources are stored.

The typical workflow for deploying a cluster using CAPI involves the following steps:

1. **Infrastructure Provisioning**
   The initial phase of cluster deployment is infrastructure provisioning. This involves setting up virtual machines or bare metal servers that will act as the cluster's worker nodes. ClusterAPI offers infrastructure providers to handle this process. These providers interact with the underlying infrastructure, automating the creation and management of necessary resources. They provision the required virtual machines or bare metal servers according to the specified cluster configuration.

2. **Cluster Configuration**
   After provisioning the infrastructure, the next step is cluster configuration. This involves setting parameters like the number of control plane nodes, worker node specifications, networking details, and any necessary customization. ClusterAPI typically uses YAML manifests or custom resources to define this cluster configuration.

3. **Bootstrap**
   Once the cluster configuration is set, bootstrapping begins. The bootstrap provider initializes the control plane nodes. It provisions resources and configures these nodes to join the Kubernetes cluster. This process establishes a functional control plane ready to manage the cluster.

4. **Worker Node Joining**
   After control plane bootstrapping, worker nodes join the cluster. ClusterAPI configures these nodes, whether virtual machines or bare metal servers, with required Kubernetes components. It manages the joining process, ensuring proper configuration and communication with the control plane.

5. **Validation and Health Checks**
   ClusterAPI then performs validation and health checks on the cluster. It verifies node connections, control plane functionality, and worker node readiness. If issues arise, ClusterAPI offers diagnostics and solutions. This workflow enables efficient cluster deployment and management. The declarative approach simplifies life cycle management, ensures consistency, and facilitates scaling and upgrades.

4.2 Metal3

Managing bare metal servers has always been a particularly challenging topic. With the current shift towards Cloud Native and Kubernetes Native software, it is important to have a solution that abstracts away complexities and allows administrators to seamlessly integrate physical servers within Kubernetes deployments.

Closely linked to ClusterAPI, Metal^3 (pronounced "metal cubed") is an open-source project that provides a set of tools for managing bare-metal infrastructure using Kubernetes [10]. Empowering organizations with a flexible, open-source solution for bare metal provisioning that combines the benefits of bare metal performance with the ease of use and automation provided by Kubernetes.

Metal^3 aims to build on top of other technologies, such as Ironic, for bare metal host provisioning. It leverages the Kubernetes control plane and various components to manage bare metal infrastructure. Its close integration with ClusterAPI allows Metal^3 to be used as an infrastructure provider backend.

4.2.1 Major Components

Metal^3 is made up of a collection of operators that work together to manage and operate bare metal infrastructure.

Bare Metal Operator

The Bare Metal Operator, a fundamental component of Metal3, manages the entire lifecycle of bare metal hosts within a Kubernetes environment. It serves as the interface between the Kubernetes cluster and the underlying physical infrastructure, orchestrating critical processes such as host discovery, provisioning, and integration.

This operator automates the registration of bare metal hosts, oversees their provisioning with required operating systems and software stacks, and seamlessly incorporates them as worker nodes into the Kubernetes cluster. It maintains continuous surveillance of host health and status, initiating automated repair procedures when necessary and facilitating software updates to ensure optimal performance and security.

The Bare Metal Operator's responsibilities extend to the graceful decommissioning of hosts, managing this process to maintain cluster stability. By automating these complex tasks, it significantly reduces the manual overhead typically associated with bare metal infrastructure management, enabling more efficient and scalable operations in Kubernetes environments utilizing physical hardware.

BareMetalHost Custom Resource Definition

The BareMetalHost (BMH) Custom Resource Definition (CRD), allows administrators to define and manage bare metal hosts as Kubernetes objects. The BareMetalHost CRD specifies the desired configuration for each host, including hardware

characteristics, network settings, storage options, and other custom at tributes. Administrators can create, update, and delete BareMetalHost objects using standard Kubernetes API operations.

Cluster API Provider Metal3 (CAPM3)

CAPM3 is one of the providers for Cluster API and enables users to deploy a ClusterAPI based cluster on top of bare metal infrastructure using Metal3. By leveraging ClusterAPI's capabilities for cluster lifecycle management and Metal^3's functionalities for bare metal provisioning, administrators can seamlessly deploy and manage Kubernetes clusters on bare metal servers.

In the context of the work of this thesis, understanding the inner working and available configuration parameters exposed by Metal^3's APIs is crucial.

Chapter 5

Automation of bare metal infrastructure provisioning

In a typical telco environment, the deployment of a new application involves several teams with distinct roles and responsibilities. We can broadly categorize these teams into a user team and an infrastructure admin team. The user team is responsible for defining the application requirements and specifying the desired state of the application deployment, while the infrastructure admin team ensures the availability and configuration of the underlying infrastructure resources.

Given the declarative nature of Kubernetes and Sylva's GitOps workflow, there are one or more Git repositories that act as the source of truth for the desired state of the system. These repositories contain the definitions and configurations for the applications, workload clusters, and other infrastructure components. To maintain a separation of concerns and ensure the security and stability of the system, it is crucial to have a clear delineation of responsibilities and access controls between the internal teams.

In certain scenarios, such as demanding performance or security requirements, it may be necessary to deploy applications directly on bare metal servers rather than virtualized environments. Since applications deployed using Kubernetes make use of containers they share the kernel on the same host, depending on the type of security requirements this kind of deployment may not be suitable.

However, configuring and managing bare metal clusters can be a complex and knowledge-intensive task, requiring expertise in areas like hardware provisioning, network configuration, and low-level system administration. Thus, user teams, whose primary focus is on application development and deployment, may not possess the specialized knowledge or privileges required to supplement the configuration needed to create a bare metal cluster.

5.1 Challenges

We need a system that leverages the Kubernetes patterns and is able seamlessly integrate within the existing Sylva framework. It should enable SWC deployment

to users and infrastructure admins with ease of use yet granular end extensive configuration. This includes respecting the patterns of underlying technologies in use within Sylva.

There should be deep integration with the existing operators granting complete compatibility, yet enough decoupling to allow for smooth day to day operation and independent usage. The sylva project is an open-source, distributed and quickly evolving environment, ensuring compatibility and adherence to the latest updates necessitates careful consideration.

5.2 Server Operator

By extending the Kubernetes API server with an operator, we are able to design a solution to manage bare metal configuration complexity and offer an ergonomic way of supplementing the required configuration.

The Server operator, running within the management cluster, is the core element of the proposed architecture. It is responsible for assigning Workload clusters with their respective bare metal configuration parameters and keeping the system up to date.

It is supposed to be deployed alongside the official Sylva operators, making use of their schema and flexibility to offer a better bare metal deployment environment.

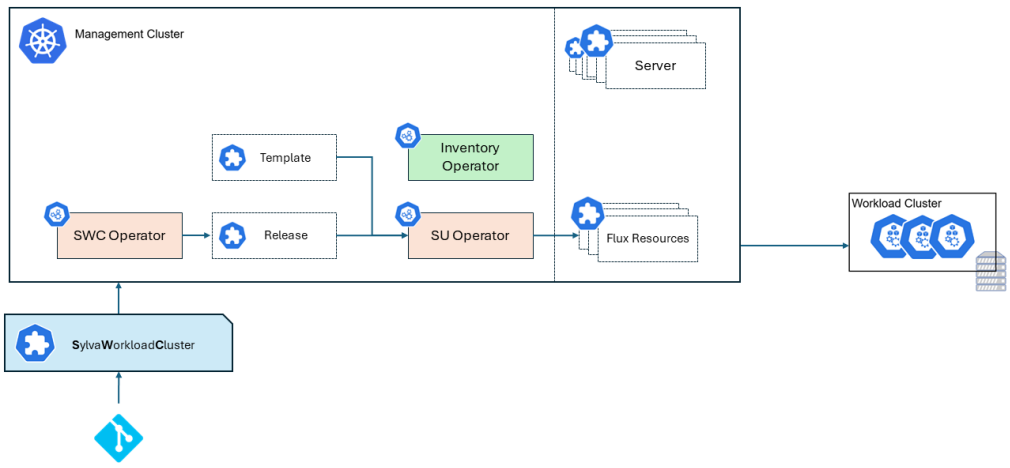

Figure 5.1. Proposed architecture

5.2.1 Server CRD

The central resource that enables us to easily represent bare metal servers and thus manage their configurations is the ‘Server‘ CRD.

This is a fairly simple structure that represents a single physical machine within one DC or edge location, externally provisioned in the management cluster. Its parameters are linked to a physical machine, it does not need to be updated, created or deleted by the operator itself, instead, it is the duty of the infrastructure admin team to manage them. When in use it becomes linked to the relative Workload Cluster, as such there must be a way to represent this state.

Within the resource are present the parameters needed for node provisioning as required by Metal^3, such as Address and credentials for the BMC, Respective MAC address, physical location of the machine, Network interfaces to use.

5.2.2 Server Controller

The controller is tasked with keeping the system up to date and bring the configuration to a coherent state. It makes use of both the 'Server' CRD and SWC/SURT to perform its duties.

In contrast to conventional controllers which directly manage their own resources, this controller will instead complementarily manage a SWC resource in concert with the existing Sylva operators, using the Server CRDs as source of additional information.

The controller must provide the configuration necessary to the SWC and keep the state of Server resources updated. This can be achieved by exploiting the Sylva Units Operator design to our advantage, as the Sylva Workload Operator reconciles SWC resources by creating, updating or deleting SUR resources. These, in turn, rely on a respective SURT. This can either be missing or already preconfigured in the corresponding namespace. The job of the Server controller is to select appropriate servers to be allocated and create or update an existing SURT with the all the relevant configuration parameters needed to successfully deploy the cluster.

The SUR and SURT CRDs share the same spec, and together share one parameter with Flux's HelmRelease API called ValuesFrom, that implements a list of the the ValuesReference spec. This in essence is a list of value files that offers a very powerful pattern, allowing the composability of multiple value files.

This layering system also implements a precedence system, so that any configuration present in files "higher" in the list takes precedence over values defined lower in the list. By placing the bare metal configuration within the base layers we can provide a baseline to the rest of the configuration, without precluding any other possible manual or automated intervention.

Since the Server objects are allocated to the SWC and every reconciliation must be stateless and idempotent, we need to keep track of which servers are allocated to which SWCs, this can be achieved by updating the Servers status fields.

5.2.3 Workflow

There are multiple considerations to be made when designing a workflow that integrates many complex technologies and the interests of various teams, from the

source of truth of the cluster state to the way the different teams should interact.

To achieve this, we can make use of the annotation mechanism of Kubernetes to complement a new or existing SWC object. Only two parameters are needed: a flag to check if the operator should act on the resource and a string identifying the physical location in which to create the cluster. Additional annotations can be used for example to specify a particular class of machines to be used, or some other implementation detail.

In essence, to deploy a Workload Cluster using the proposed operator, a SWC resource needs to be created or annotated with the "enabled" and relative "site" keys. Provided there are enough Server resources available in the requested site, the Server Operator will provision the necessary configuration within the SURT, thus producing a valid configuration that will be reconciled first by the SWC Operator and then by the SUR Operator, producing a valid HelmRelase and associated resources that will initiate the workload cluster deployment.

A SWC can be easily deployed on different providers, such as vSphere or OpenStack, however when the target infrastructure are bare metal servers, more configuration is required. Unlike virtual machines, bare metal nodes are intrinsically more coupled to the underlying hardware. Special care needs to be taken when provisioning them.

Chapter 6

Implementation

This chapter contains a proof-of-concept implementation that has been tested and proven to work under virtualized and bare metal scenarios. We will look into more detail how the implementation is achieved, the languages used and how it integrates within the Sylva framework.

This work leverages extensively the comprehensive scaffolding and development ecosystem provided by Kubebuilder. The project structure, API definitions, and controller implementations are all built upon the foundational elements generated by Kubebuilder's CLI tool. Beyond mere scaffolding, Kubebuilder's utilities for testing, building, and deploying have been integral to the development process. The tool's opinionated approach to Kubernetes extension development has guided the architecture of our custom controllers and resources, ensuring adherence to best practices in cloud-native application development.

By utilizing Kubebuilder's framework, this work benefits from a standardized, maintainable codebase that aligns with the broader Kubernetes ecosystem, thereby enhancing compatibility and easing future extensions or modifications to the operator.

6.1 Server CRD

We start with the implementation of a base version of the CRD as previously described:

Listing 6.1. Server CRD Implementation

```
1  package v1alpha1
2
3  import (
4          corev1 "k8s.io/api/core/v1"
5          metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
6  )
7
8
9  type ServerSpec struct {
10         // +kubebuilder:validation:Required
```

```
11          Bmc string json:"bmc"
12
13          // +kubebuilder:validation:Required
14          Credentials BmcCredentials json:"credentials"
15
16          // +kubebuilder:validation:Required
17          BootMac string json:"bootMac"
18
19          // +kubebuilder:validation:Required
20          Site string json:"site"
21  }
22
23  type BmcCredentials struct {
24          Username string json:"username"
25          Password string json:"password"
26
27          // +optional
28          SecretRef corev1.SecretKeySelector json:"secretRef"
29  }
30
31  type ServerStatus struct {
32
33          // +kubebuilder:default=false
34          Allocated bool json:"allocated"
35          Role string json:"role,omitempty"
36          Errors []string json:"errors,omitempty"
37
38          WorkloadCluster WorkloadCluster
                json:"workloadCluster,omitempty"
39  }
40
41  type WorkloadCluster struct {
42          Name string json:"name"
43          Namespace string json:"namespace"
44  }
45
46  //+kubebuilder:object:root=true
47  //+kubebuilder:subresource:status
48  //+kubebuilder:printcolumn:name="Site",type="string",JSONPath=".spec.site"
49  //+kubebuilder:printcolumn:name="Allocated",type="boolean",JSONPath=".status.alloca
50  //+kubebuilder:resource:scope="Cluster"
51
52  type Server struct {
53          metav1.TypeMeta json:",inline"
54          metav1.ObjectMeta json:"metadata,omitempty"
55
56          Spec ServerSpec json:"spec,omitempty"
57          Status ServerStatus json:"status,omitempty"
58  }
59
60  //+kubebuilder:object:root=true
```

```
61
62 // ServerList contains a list of Server
63 type ServerList struct {
64        metav1.TypeMeta json:",inline"
65        metav1.ListMeta json:"metadata,omitempty"
66        Items []Server json:"items"
67 }
68
69 func init() {
70        SchemeBuilder.Register(&Server{}, &ServerList{})
71 }
```

We can note the simplicity of the design as only the base parameters are required within the object itself.

There is one main object defined in this code, `Server`, which is implemented as a Go struct. As a standard Kubernetes object, it is composed of the standard Type and Object metadata, followed by its **spec** and **status**.

The status contains the parameters necessary to maintain a link between a SWC and its allocated Servers. We also specify some validation parameters using Kubebuilder's Markers, they are single-line comments that start with a plus, followed by a marker name, optionally followed by some marker specific configuration. We make use of this mechanism to also add some extra parameters when retrieving objects with Kubectl, namely we print the site and allocation status, to allow for debugging.

6.2 Server Controller

The management of Servers and their interaction with SWCs falls to the Server Controller. Thanks to Kubebuilder scaffolding, the implementation follows Kubernetes standards and is easy to navigate and maintain.

A Kubernetes controller is based upon the Reconcile loop, that usually involves:

- **Observing** the state of the cluster and listening for changes and events the controller is responsible for.
- **Retrieving** the current state of objects affected by said events and any other related resources.
- **Comparing** the retrieved result with the expected one and determine if action needs to be taken.
- **Take Action** Based on the result of the comparison. This may involve creating, deleting or modifying Kubernetes objects or calling external services.
- **Update the Status** of affected resources to reflect their current state and any action taken, providing visibility within the reconciliation process.

- **Requeue** the reconciliation in the event that another reconciliation is needed in the future, for example due to errors in the previous one. This may happen immediately or after a period of time. The framework will take care of eventual exponential backoff.

When implementing the proposed solution, care must be taken to ensure idempotency, to avoid any unwanted side effects, which are likely to happen in an asynchronous and event-based architecture. The controller framework also takes care internally of optimizations like caching to reduce load and improve performance.

6.2.1 Sylva Operators reconciliation loop

To better fit in withing the Sylva context we imitate their reconciliation loop structure, which could be broadly simplified as follows:

Listing 6.2. Sylva controller structure

```
1  package controller
2
3  import (
4          ...
5  )
6
7  type ResourceReconciler struct {
8          client.Client
9          Scheme *runtime.Scheme
10         record.EventRecorder
11 }
12
13 const finalizerName = "..."
14
15 func (r *ResourceReconciler) Reconcile(
16                 ctx context.Context,
17                 req ctrl.Request,
18         ) (result ctrlResult, retErr error) {
19
20         // Retrieve the object that triggered the reconciliation
21         resource := &apis.Resource{}
22         if err := r.Get(ctx, req.NamespacedName, resource); err != nil
               {
23                 // log
24                 return ctrl.Result{}, client.IgnoreNotFound(err)
25         }
26
27         // Initialize the patch helper.
28         patchHelper := patch.NewSerialPatcher(resource, r.Client)
29
30         // Always attempt to Patch the Cluster object
31         // and status after each reconciliation.
32         defer func() {
33                 patchOpts := []patch.Option{}
```

```
34                if retErr != nil {
35
36                } else if ... {
37                        // Patch ObservedGeneration as the reconciliation
38                        // completed successfully
39                        r.Eventf(
40                                resource, "Normal",
41                                "ReconciliationSucceeded",
42                                "Successfully reconciled Resource %s",
43                                resource.Name
44                        )
45                        patchOpts = append(
46                                patchOpts,
47                                patch.WithStatusObservedGeneration{}
48                        )
49                }
50                if err := patchHelper.Patch(ctx, resource,
                      patchOpts...); err != nil {
51                        retErr = kerrors.NewAggregate([]error{retErr,
                          err})
52                }
53        }()
54
55        // Handle deletion request
56        if !resource.ObjectMeta.DeletionTimestamp.IsZero() {
57                return r.reconcileDelete(ctx, resource)
58        }
59
60        // Apply finalizer if not present
61        if !controllerutil.ContainsFinalizer(resource, finalizerName) {
62                controllerutil.AddFinalizer(resource, finalizerName)
63                return ctrl.Result{Requeue: true}, nil
64        }
65
66        // Handle normal reconciliation loop.
67        return r.reconcile(ctx, resource)
68
69 }
70
71 func (r *ResourceReconciler) reconcile(
72        ctx context.Context,
73        req ctrl.Request
74 ) (ctrl.Result, error) {
75        // Check current status against requested status,
76        // take action as needed
77 }
78
79 func (r *ResourceReconciler) reconcileDelete(
80        ctx context.Context,
81        req ctrl.Request)
82 (ctrl.Result, error) {
```

```
83          // Gracefully delete associated resources,
84          // once completed remove the finalizer
85  }
```

This structure is modular, adaptable and is peculiar due to its use of the patch-Helper. The controller can act directly on the main object specified in line 19, greatly simplifying object handling and, within the deferred function at the end of the reconciliation loop, we try to apply any changes to the Kubernetes server.

6.2.2 Server Operator implementation

We can adapt this general structure to fit the needs of our controller, considering that the Server controller does not act directly on the Server resource, but watches for SWCs and acts on SURTs.

The process begins when a user team creates or annotates a SylvaWorkloadCluster (SWC) resource. Two key annotations are required:

1. An **enabled** flag to indicate that the Server Operator should act on this resource.
2. A **site** string to specify the physical location for cluster deployment.

These simple annotations trigger the Server Operator to begin its reconciliation process, bridging the gap between the user's high-level cluster requirements and the low-level bare metal configuration.

We specify this requirement when registering the reconcile loop with the Controller Manager, where we also specify additional parameters such as resource indexing for more efficient query operations:

```
1  func (r *ServerReconciler) SetupWithManager(mgr ctrl.Manager) error {
2     // Index Server resources based on allocation status
3     if err := mgr.GetFieldIndexer().IndexField(
4        context.Background(),
5        &inventoryv1alpha1.Server{},
6        allocatedStatusField,
7        serverIndexer,
8     ); err != nil {
9        return err
10    }
11
12    // Index SylvaWorkloadCluster by the enabled annotation
13    if err := mgr.GetFieldIndexer().IndexField(
14       context.Background(),
15       &wc1a1.SylvaWorkloadCluster{},
16       annotationEnabledField,
17       sylvaWorkloadClusterIndexer,
18    ); err != nil {
19       return err
```

```
20    }
21
22    return ctrl.NewControllerManagedBy(mgr).
23    // watches for changes of SylvaWorkloadCluster resources
24    Named("ServerReconciler").
25    Watches(
26       &wc1a1.SylvaWorkloadCluster{},
27       &handler.EnqueueRequestForObject{},
28       builder.WithPredicates(cdPredicate{})
29    ).
30    Complete(r)
31 }
```

This controller reacts to updates on a SWC Resource which is managed by another operator, furthermore, to not waste time and computing resources reconciling foreign objects, we need to consider events only when the resource in question has out enabled annotation. This is achieved by implementing custom `Predicates` which will filter events before enqueuing.

Listing 6.3. Filtering events before handing them to the reconciler

```
1  type cdPredicate struct {
2     client.Client
3  }
4
5  func (cdPredicate) Create(e event.CreateEvent) bool {
6     if !wcIsEnabled(e.Object) {
7        return false
8     }
9     return true
10 }
11
12 func (cdPredicate) Delete(e event.DeleteEvent) bool {
13    if !wcIsEnabled(e.Object) {
14       return false
15    }
16    return true
17 }
18
19 func (cdPredicate) Update(e event.UpdateEvent) bool {
20    if !wcIsEnabled(e.ObjectNew) || !wcIsEnabled(e.ObjectOld) {
21       return false
22    }
23    // Only update on changes to relevant parameters
24    old, ok := e.ObjectOld.(*wc1a1.SylvaWorkloadCluster)
25    if !ok {
26       return false
27    }
28    new, ok := e.ObjectNew.(*wc1a1.SylvaWorkloadCluster)
29    if !ok {
30       return false
31    }
```

```
32
33     ...
34 }
35
36 func (cdPredicate) Generic(e event.GenericEvent) bool {
37     if !wcIsEnabled(e.Object) {
38         return false
39     }
40     return false
41 }
42
43 func wcIsEnabled(o client.Object) bool {
44     wc, ok := o.(*wc1a1.SylvaWorkloadCluster)
45     if !ok {
46         return false
47     }
48     return wc.Annotations[annotationEnabledKey] == "true"
49 }
```

Once the controller receives an event with the proper requisites we can begin. The reconciliation process for a SWC involves several key steps:

1. **Initiation and Suspension** The process begins by suspending the target SWC. This prevents potential conflicts with other controllers operating concurrently, creating a stable environment for our operations.

2. **Resource Assessment** The controller retrieves the specifications for both control plane and worker nodes from the SWC. It also identifies any servers already allocated to this resource, comparing the current state to the desired state.

3. **Resource Allocation** If discrepancies exist between the current and desired states, the controller allocates additional servers that meet the specified location requirements. As servers are assigned, their status is updated to reflect the new allocation.

4. **Configuration Generation and Deployment** The controller generates the necessary configuration, which is split into two Kubernetes resources:

   ConfigMaps Contain non-sensitive information about the bare metal infrastructure.

   Secrets Store sensitive data, such as the BMC credentials

   Both resources are deployed in the same namespace as the SWC, using the server-side apply strategy. This approach allows for configuration merging while avoiding potential conflicts.

5. **SURT Configuration Integration** The new configuration is integrated into the SylvaUnitsReleaseTemplate (SURT). This is achieved by adding the bare metal configuration to the "lower" levels of the SURT spec, ensuring it coexists with other configuration elements.

6. **SWC Reactivation** Finally, the SWC is resumed, allowing other controllers to resume their activities. The cluster, now updated with its bare metal components, is ready to be deployed and operated upon by other controllers.

```
func (r *ServerReconciler) reconcileSylvaWorkloadCluster(
ctx context.Context,
wc *wc1a1.SylvaWorkloadCluster
) (ctrl.Result, error) {

   // Suspend the SylvaWorkloadCluster
   if err := r.suspendSylvaWorkloadCluster(ctx, wc); err != nil {
      return ctrl.Result{}, err
   }

   // Gather requirements
   controlPlaneNodes := getControlPlaneNodes(wc)
   workerNodes := getWorkerNodes(wc)

   // Get already allocated servers
   allocatedServers, err := r.getAllocatedServers(ctx, wc)
   if err != nil {
      return ctrl.Result{}, err
   }

   // Check if more servers need to be allocated
   if needsMoreServers(
   controlPlaneNodes,
   workerNodes,
   allocatedServers
   ) {
      availableServers, err := r.getAvailableServers(ctx, wc)
      if err != nil {
         return ctrl.Result{}, err
      }

      if len(availableServers) <
          int(controlPlaneNodes)+int(workerNodes)-allocatedServers.Total()
          {
         return ctrl.Result{Requeue: true}, nil
      }

      allocatedServers, err = r.allocateServers(
         ctx,
         wc,
         allocatedServers,
         controlPlaneNodes,
         workerNodes,
         availableServers
      )
      if err != nil {
         return ctrl.Result{}, err
```

```
46          }
47      }
48
49      // Update server statuses
50      if err := r.updateServerStatuses(ctx, allocatedServers); err != nil
            {
51          return ctrl.Result{}, err
52      }
53
54      // Generate resources
55      resources, err := r.generateResources(ctx, allocatedServers.All(),
            wc)
56      if err != nil {
57          return ctrl.Result{}, err
58      }
59
60      // Apply generated resources
61      if err := r.applyResources(ctx, resources); err != nil {
62          return ctrl.Result{}, err
63      }
64
65      if err := r.resumeSylvaWorkloadCluster(ctx, wc); err != nil {
66          return ctrl.Result{}, err
67      }
68
69      return ctrl.Result{}, nil
70  }
```

This reconciliation process demonstrates how bare metal resources can be effectively managed within the Kubernetes orchestration framework, bridging traditional infrastructure with cloud-native paradigms.

Configuration Generation

In the current implementation we use a simple approach. When deploying a Workload Cluster on bare metal machines, we need to supply the Sylva Units Chart Values with information about its `baremetal-hosts`. Specifically the BMH metadata, spec and a reference to the location of relative credentials for each bare-metal machine we wish to use.

This is accomplished by making use of Go templates for both the ConfigMap and Secret generation. These templates may be hard-coded as strings within the operator itself, or can be saved on a Kubernetes resource like a ConfigMap and referenced, to facilitate easier configuration and upgrades.

Server Resource Cleanup

The Server Operator can gracefully delete resources when it receives the event of a SWC deletion, but to ensure consistency of resources within the cluster, the Server

Operator implements a robust cleanup mechanism to ensure efficient resource management and maintain system integrity. This process is crucial for handling scenarios such as SWC deletion, scaling down operations, or recovering from failures. The cleanup procedure gets called at every reconciliation in a deferred function, following the Sylva Operator pattern, and it involves they key steps:

1. **Orphaned Server Detection** The operator scans for orphaned servers - those that are marked as allocated but are no longer associated with an existing SWC.

2. **Server Relase and Resource Deletion** When an orphaned server is detected the operator initiates the server release process.

Considerations

While the SWC is in a transition phase, as in, it has been created but not yet reconciled by the Server operator, we need to discuss some considerations:

The other Sylva operators may be acting on the same resources. Given the asynchronous, and event-based architecture of operators, changes could have been made concurrently to our operator. So the SWC Operator could have already created a relative SUR, which in turn could have already been handled by the Sylva Units Operator, generating an HelmRelase. This is not a problem for our operator, as any conflicting or concurrent change would be rejected by the Kubernetes API server. Such rejections would trigger a requeue of our operator's reconciliation loop, ultimately leading to eventual consistency across the system.

The HelmRelease is not complete before the Server operator has completed its tasks. In fact, it would be missing any infrastructure detail needed for a successful deployment. This is not a problem since, just as in the case of our operator, the Helm controller would requeue the reconciliation to eventually achieve a succesful deployment.

The solution maintains a clear separation of concerns between the user team and the infrastructure admin team. The user team only needs to specify high-level cluster requirements in the SWC, while the infrastructure admin team manages the underlying Server resources. This separation allows each team to focus on their area of expertise while the Server Operator bridges the gap between them.

6.3 Usage

Before deploying a Workload Cluster on bare-metal instances using the Server Operator, an infrastructure admin must create Server resources within the Management Cluster.

The origin of these resources is not fixed, they can be create manually, can be included in the main repository detailing the cluster infrastructure and definition, or can reside in a repository controlled separately from the main one, for separation of concerns. Following GitOps standards and procedures it is advisable to have their definition in a separate repository, linked to the Management cluster.

Listing 6.4. Sample Server Object

```yaml
apiVersion: inventory.sylva.telecomitalia.it/v1alpha1
kind: Server
metadata:
   labels:
      app.kubernetes.io/name: server
      app.kubernetes.io/instance: server-sample
      app.kubernetes.io/part-of: inventory-operator
      app.kubernetes.io/managed-by: kustomize
      app.kubernetes.io/created-by: inventory-operator
name: server-sample-1
   spec:
      bmc: "test:ip-address/uri"
      credentials:
         username: "username"
         password: "password"
      bootMac: "mac"
      site: "to-1"
```

Once the Server resources are present in the cluster a SWC can be deployed succesfully by simply requesting the intervention of the Server Operator and specifying in which location the Workload Cluster should be created. This is done via annotations.

Listing 6.5. Example of annotated SylvaWorkloadCluster

```yaml
apiVersion: workloadclusteroperator.sylva/v1alpha1
kind: SylvaWorkloadCluster
metadata:
   labels:
      app.kubernetes.io/name: sylvaworkloadcluster
      app.kubernetes.io/instance: sylvaworkloadcluster-sample
      app.kubernetes.io/part-of: workload-cluster-operator
      app.kubernetes.io/managed-by: kustomize
      app.kubernetes.io/created-by: workload-cluster-operator
   name: sylvaworkloadcluster-baremetal
   annotations:
,     inventory.sylva.telecomitalia.it/enabled: "true"
,     inventory.sylva.telecomitalia.it/site: "to-1"
spec:
   k8sVersion: "1.27"
   infrastructure: metal3
   controlPlane:
      provider: rke2
      replicas: 1
      machineDeployments:
```

```
21          md0:
22            replicas: 2
```

At this point the reconciliation of all the involved controllers takes place, creating a Workload Cluster and connecting it to the Management Cluster.

Chapter 7

Results

To test and verify proper operation of the Server Operator, two different environments have been utilized, an emulated setup and a bare metal setup.

Operating with real bare-metal machines is an expensive and time consuming process, so during the development stages of the thesis, we exploited an emulated bare-metal environment. This allowed for quick iterations and testing, with increased flexibility and without occupying entire servers.

7.1 General Configuration

Regardless of the environment used, the goal of the thesis is that of simplifying and automating the provisioning of bare metal infrastructure, facilitating the work of infrastructure admin and product teams. Thus we can compare the amount of additional configuration needed, measured in lines of code, for the deployment of a Workload Cluster made up of 3 control plane nodes.

	Manual approach	Server Operator
Sylva Units values	54	2
Additional Resources	18	33
Total	72	35

Table 7.1. Additional LOC required for bare metal deployment

In the manual approach case, on top of the standard Sylva Units configuration we need to take into account the Secrets containing credentials of the BMH relative to the physical machines in use. While for the Server Operator case we have to consider the configuration required to create the Server objects in the cluster.

7.2 Emulated Environment

The emulated environment makes use of another project born within Sylva, **Libvirt Metal**. It builds an image that can emulate a bare metal server within a container, without the need to use other possibly expensive operations.

It leverages libvirt, a collection of software that provides a convenient way to manage virtual machines and other virtualization functionality, and sushy-emulator, a virtual Redfish BMC, to mimic a baremetal server that can be provisioned with metal3 [11].

The tests have been performed on a virtual machine with 32 CPU Cores and 128 GB of RAM which hosted both the Management Cluster and any created Workload Cluster within virtual machines created using Libvirt Metal.

7.3 Bare Metal Environment

For the bare metal environment, two servers of the same model have been used as target for the Workload Cluster. These are Dell PowerEdge R640 servers, each with the following specifications:

- CPU: Intel(R) Xeon(R) Gold 6252N, operating at 2.30 GHz, with 24 cores
- Storage: 2x 1.2 TB SAS 10k
- RAM: 384 GB
- Network Interfaces: 4x 10GbE and 2x 25GbE

The nodes belonging to the Management Cluster instead were run inside vSphere VMs, 3 for the control plane and 2 for worker nodes, each with 4 CPU cores and 8GB of RAM.

7.4 Results

Running the Server Operator on both environments produces extremely similar results as it does not directly communicate with bare metal machines. While the time to complete the creation of a Workload Cluster varied greatly between the environments, it is mostly outside the scope of this work.

We can compare the time and resources spent reconciling different events regarding a SWC of the Server Operator against the official Sylva Operators. We will take in account the time and resources elapsed between the creation of an annotated SWC resource requesting 2 bare metal servers in a management cluster with already provisioned and available Server resources.

We note how the majority of the time spent in the deletion scenario by the Sylva operators reconciliation process is caused by deletion requests sent to the Kubernetes API server, not their own processing time.

Similarly we can compare the amount of physical resources used by all of the operators. We use the resident memory size as measure of RAM usage. Since operators usually are idle and wait for events emitted by the control plane, there is not a continuous usage of resources thus we measure the cumulative CPU spent time to reconcile one SWC resource.

	Server Operator	SUR Operator	SWC Operator
SWC Creation	260 ms	675 ms	300 ms
SWC Edit	450 ms	185 ms	1850 ms
SWC Deletion	305 ms	12905 ms	15215 ms

Table 7.2. Time to reconcile SWC events

	CPU	RAM
Server Operator	50ms	42.8 KB
SUR Operator	100ms	35.4 KB
SWC Operator	50ms	33.9 KB

Table 7.3. Resources used by involved operators

7.5 Conclusions

Thanks to the lightweight and event-based model of the Kubernetes operator pattern, trough this benchmarking we can see that the Server Operator is lightweight on resources and given the time taken to perform its operations when responding to SWC events, such as selecting, allocating Servers and generating the relative Kubernetes resources, it is relatively efficient compared to the rest of the Sylva ecosystem.

Nevertheless, it's essential to recognize that the results obtained from these benchmark tests are closely linked to the particular hardware employed during the testing phase. The computational capabilities and features of servers, virtual machines, and network devices directly impact performance outcomes. In cloud environments, system performance is often intricately connected to the capabilities of the nodes in operation. Consequently, observed performance characteristics could vary significantly if the system were to undergo changes.

Chapter 8

Future Work

8.1 Possible Integrations

While the work of this thesis focuses on a solution to easily provide the correct parameters for bare metal servers, deploying bare metal Workload Clusters requires even more configuration, which for the purposes of this work has been hard-coded within the resource generation code.

One example is the IP Address configuration required by Metal^3, while it could exist as a completely separate component, a deeper integration with the Server Operator could greatly simplify the workflow.

As it stands, the proposed solution requires that Server objects be provisioned within the Management Cluster. An integration with some external inventory management tool, which Telcos are bound to use already in some capacity, could automate the provisioning and management of Server resources, reducing the manual burden placed onto infrastructure administrators.

Chapter 9

Conclusion

The shift towards Cloud Native applications has driven the ever growing necessity of efficiently managing an increasing number of Edge sites and Network Functions, renewing challenges for Telcos.

The work of this thesis fits in the broader evolution and enhancement process of the Sylva project. While providing an important, common reference framework for european Telcos and vendors, it still faces several challenges, particularly in managing the inherent complexities of bare metal environment.

One critical challenge addressed in this work is the substantial domain-specific knowledge required for effective deployment of new bare metal Workload Clusters. The solution proposed and implemented in this thesis offers a practical and efficient approach to this challenge. By leveraging Kubernetes patterns and extending the existing Sylva framework, we have developed a Server Operator that seamlessly integrates with Sylva's declarative and GitOps-driven workflow.

This operator significantly simplifies the process of bare metal cluster deployment, allowing infrastructure administrators and application developers to focus on their respective areas of expertise. It achieves this by automating the provisioning of necessary configurations, thereby reducing operational complexity and minimizing the potential for human error. The implemented solution not only streamlines the deployment process but also enhances collaboration between different teams involved in cluster management.

As the telecommunications industry continues its digital transformation, solutions like the one presented in this thesis will play a crucial role in realizing the full potential of cloud-native technologies in telco environments.

Bibliography

[1] The Kubernetes Project https://kubernetes.io (visited on 12/07/2024)

[2] W. Felter, A. Ferreira, R. Rajamony and J. , "An updated performance comparison of virtual machines and Linux containers", 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2015, pp. 171-172, DOI 10.1109/ISPASS.2015.7095802

[3] NGMN Cloud Native Manifesto https://www.ngmn.org/wp-content/uploads/NGMN_Cloud_Native_Manifesto.pdf (visited on 12/07/2024)

[4] Kubernetes Overview https://kubernetes.io/docs/concepts/overview/ (visited on 12/07/2024)

[5] Kubernetes Components https://kubernetes.io/docs/concepts/overview/components/ (visited on 12/07/2024)

[6] Kubernetes Objects https://kubernetes.io/docs/concepts/overview/working-with-objects/ (visited on 12/07/2024)

[7] Gitlab - Operators Telco Cloud - White Paper https://gitlab.com/sylva-projects/sylva/-/blob/main/White_Paper_Operators_Sylva.pdf?ref_type=heads (visited on 12/07/2024)

[8] Sylva Project https://sylvaproject.org/ (visited on 12/07/2024)

[9] GitHub ClusterAPI https://github.com/kubernetes-sigs/cluster-api (visited on 12/07/2024)

[10] Metal^3 https://metal3.io/ (visited on 12/07/2024)

[11] Virtual Redfish BMC https://docs.openstack.org/sushy-tools/latest/user/dynamic-emulator.html (visited on 12/07/2024)