Politecnico di Torino

Politecnico di Torino

Master of Science in Computer Engineering

A.Y. 2024/2025

Graduation Session October 2025

Framework for Network Services Orchestration with a Cloud Native Approach

Supervisors:

Prof. Fulvio Risso

Dott. Davide Miola

Company Tutors:

Dott. Paolo Fasano

Dott.ssa Anna Andreotti

Dott. Antonio Giannone

Candidate:

Giacomo Olivero

Abstract

The management of modern telecommunications networks is complex, driving the adoption of *Software Defined Networking* (SDN). A typical SDN approach may rely on a monolithic architecture, with very strict resource requirements and a centralized database, that could lead to scalability issues in large-scale environments. This thesis faces these challenges by designing, implementing, and validating a new framework for network services orchestration based on cloud-native principles.

The proposed architecture uses *Git* as a single source of truth and leverages GitOps workflow principles to translate high-level network service definitions into low-level device configurations, which are then applied to the targets using the *gNMI* protocol.

Implementation required the development of a completely new module and the contribution to an open-source project, *Kubenet SDCIO*, that is in charge of the configurations provisioning to the network devices. The framework functionality was validated both on virtual and physical network devices. The successful deployment and positive performance results confirm the scalability of the proposed solution, also for large-scale networks. This work concludes that the adoption of cloud-native principles provides a robust and scalable solution for modern network automation.

Acknowledgements

Writing a Master's thesis is a challenging task that requires a lot of effort and dedication. It is also a journey of personal growth and learning, that allows you to put into practice the knowledge and skills you have acquired during your studies, and I am thankful to all those who have supported me along the way. This work would not have been possible without the contribution of all these individuals, and I am deeply grateful to each of them.

Thanks to *TIM* for the opportunity to work on this project, with a special mention to my company tutors, Anna and Antonio, for their constant support and guidance throughout the development of this thesis, and to Dott. Paolo Fasano, for his availability and coordination.

Thanks to my supervisor, Prof. Fulvio Risso, who acted as a guide and a point of reference rather than a simple evaluator, and to my academic tutor, Davide, for his precious support and experienced suggestions.

Thanks to my family, for their unconditional support and encouragement, that allowed me to face this challenge with confidence and determination, always providing me with the tools I needed.

Thanks to my friends, inside and outside the university, for their companionship and for helping me maintain a balanced perspective during the challenging moments of this journey.

Thanks to everyone who has proposed me new challenges and opportunities over the years, allowing me to develop a broader vision and a more complete set of skills, building a portfolio far richer than what a Master's degree alone could provide.

Thanks to all who will read this thesis, for their interest and curiosity.

"I am always doing what I cannot do yet, in order to learn how to do it."
— Vincent van Gogh

Table of Contents

List of Figures

Chapter 1

Introduction

Everyday, billions of people are connected to the Internet, someone with a wired connection in their home or office, others with a mobile connection on their smartphone. All of that is a fundamental part of our life, and is getting more and more important. Behind the scenes, *Telco* (TELecommunications COmpany) infrastructures are the backbone of this connectivity. They are complex systems, with thousands of devices, components, and technologies involved. Monitoring all of that and assuring each person in the world can access the Internet every time they want is a challenging task, that requires a lot of effort and resources.

Such a big system is supervised by a specific framework that helps the management team both to operate on the network maintaining the control, and to monitor the status continuously. Several tools have been deployed thus far, some open-source, others proprietary. However, when applied to a real-world scenario, each of them shows its limitations: each *Telco* operator has specific needs, and general-purpose tools are not always able to satisfy them.

1.1 Context

This thesis has been carried out at *TIM S.p.A.*, the largest Italian telecommunications company. The company vision of *Software Driven Network* (SDN) is structured as an organism with three main pillars that work together:

- **Read**: collect data from the network, using different protocols and technologies;
- **Write**: act on the network, changing its state and configuration;
- **Brain**: the core of the system, that processes data and decides what to do.

Thanks to this approach, when something new should be applied to the network, the *Brain* can decide what to do, and then the *Write* pillar can act on the network, and finally the *Read* module can check if everything is working as expected. On the other hand, using the *Read* pillar, the *Brain* can monitor the network status and take intelligent decisions based on the data collected, that are then applied to the network using the *Write* module.

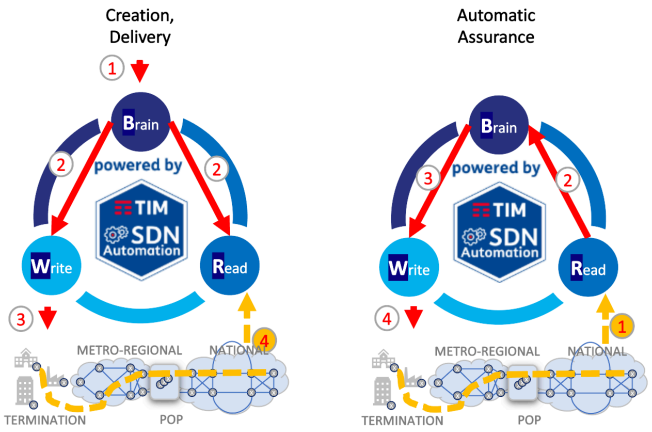

Figure 1.1: Software Driven Network at TIM [1]

Each pillar relies on several smaller components, some of them developed in-house, others are open-source or proprietary tools. This thesis focuses on the *Write* pillar, and in particular on the module responsible for applying changes to network devices. The current solution is based on **Cisco NSO** (Network Services Orchestrator), a proprietary tool developed by *Cisco Systems*, that have been in use from years in *TIM*.

1.2 Problem

Cisco NSO has been chosen among the others because it is flexible and able to manage devices from different vendors and with different technologies, using many communication protocols. However, using it at scale some issues are now arising in the module:

- *Cisco NSO* is structured as a centralized point of control, that is very convenient from a logical point of view, but it becomes a bottleneck when the number of devices to manage is very high, principally caused by blocking operations on the internal database.

- the technicians must be able to operate directly on the network devices, and this can sometimes conflict with the NSO operations: the current system is reading back from the device the configuration before applying a change, but this cause a great overhead and it is not always reliable;

1.3 Goal

The problem of having a centralized module that is not able to adapt to the traffic it might have is not new in the IT world. In the last years, the *Cloud Native* approach has been adopted by many companies to solve similar problems in other fields than networking. The idea is to design and implement systems that can scale horizontally, respecting three principal criteria:

- **Statelessness**: each component should not store any state information, that should be stored in an external database;
- **Microservices**: each component should be as small as possible, and should do only one thing, interact with other components using well-defined APIs, and should be replaceable;
- **Scalable**: the system should be able to scale horizontally, working with more instances of the same component in an organized way.

The main goal is then to study the feasibility and produce a prototype of a *Cloud Native* module that can replace the current *Cisco NSO* based one, exploring the technologies and tools already present in the world and eventually developing new ones if needed.

The long term goal, of which this thesis is the starting point, is to have a fully functional *Cloud Native* module that can replace the actual one, solving the issues described before and being more flexible and reliable.

1.4 Structure of the Thesis

The thesis is structured as follows:

- **Chapter 1 - Introduction** (this chapter) introduces the context of the work, the problem to solve, and the goals to achieve;
- **Chapter 2 - Background** provides the necessary background about the technologies and tools used in the work;

- **Chapter 3 - Services in Telcos** describes what services are and how they are provided to customers in a *Telco* context;
- **Chapter 4 - Framework Architecture** illustrates the proposed Framework architecture, the components used, and how they interact together;
- **Chapter 5 - Implementation** describes the implementation details of the proposed Framework, the challenges faced, and how they have been solved;
- **Chapter 6 - Validation** reports the tests performed to validate the proposed Framework and the results obtained;
- **Chapter 8 - Future Work** discusses the possible future developments and improvements that can be made to the proposed Framework;
- **Chapter 7 - Conclusion** summarizes the work done and the results achieved.

Chapter 2

Background

The automation and orchestration of modern network infrastructures require a set of technologies and paradigms that allow efficient management of both distributed workloads and network device configurations. This chapter presents the fundamental technologies on which our framework is built, illustrating how each of them contributes to the proposed solution.

The following sections explore Kubernetes, ArgoCD, Network Operating Systems, and the Kubenet framework, highlighting their key features and their relevance to our work.

2.1 Kubernetes

Kubernetes, often abbreviated to k8s, is a large open-source system for automating deployment, scaling, and management of containerized applications [2]. Particularly interesting for our purpose is the general concepts of declarative idempotency and the use of *custom resources* (CRs) to extend the Kubernetes API, providing extensibility and flexibility in managing resources.

Kubernetes represents the de facto standard for container orchestration, being the state of the art of Cloud Native evolution. Traditionally, applications were deployed on bare metal servers, that over time have evolved into virtual machines, which are now being replaced by containers, lightweight isolated systems that are fully independent of one another, but rely on the host OS, reducing the overhead. The first tool to popularize containers was Docker, while Kubernetes permits to manage them at scale, orchestrating containerized applications across clusters of machines. Particularly interesting in that sense is the vision Kubernetes has of servers, which pass from being treated as 'pets' to be cared for, to 'cattle' to be managed, where the focus is on the applications and their deployment rather than on the underlying infrastructure.

2.1.1 Declarative Model

Every resource in Kubernetes, such as pods[1], services[2], or nodes[3] themselves, is defined in a declarative manner, meaning that the desired state of the resource is specified and Kubernetes takes care of ensuring that the actual state matches the desired state. This approach, known as declarative idempotency, allows for easier management of resources, as the system automatically reconciles the current state with the desired state, ensuring that the system converges towards the specified configuration. Moreover, the system administrator can focus on defining what the system should look like, rather than how to achieve that state, which is handled by Kubernetes itself.

Each of that resources is managed through a YAML or JSON manifest file, that has some fields that are common to all resources, such as `apiVersion`, `kind`, and `metadata`, that are used to identify and centrally manage the resource, and some fields that are specific to the resource type, that are enclosed in the `spec` and `status` sections. The `spec` section defines the desired state of the resource, while the `status` section contains the current state of the resource, which is automatically updated by Kubernetes.

Resources are stored in a distributed key-value store called 'etcd', that stands for 'distributed etc'[4], which is in charge of persisting the state of the cluster and ensuring that all nodes in the cluster have a consistent view of the resources. The interaction with external tools and users is permitted by a RESTful API, that allows to create, read, update, and delete resources in the cluster. The *API server* is a special pod that acts as the central point of communication for all resources in the cluster, exposing this API.

Figure 2.1: Interaction Model of API Server

[1] A pod is the smallest deployable unit in Kubernetes, representing a single instance of a running process in a cluster, and can contain one or more containers.

[2] A service is an abstraction that defines a logical endpoint and a policy for accessing resources, often used to expose applications running on a set of pods.

[3] A node is a worker machine in Kubernetes, which may be a physical or virtual machine, and is where the pods are run.

[4] `etc` is a Unix directory that contains configuration files.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

Example of a Kubernetes resource manifest

Custom Resource Definitions

There is a set of built-in resources that Kubernetes provides, that are the bricks of the system, such as pods, services, deployments, and nodes. However, Kubernetes also allows users to define their own custom resources, which can be used to extend the Kubernetes API and provide additional functionality.

These are called **Custom Resources** (CRs) and are defined using *Custom Resource Definitions* (CRDs), which are themselves Kubernetes elements that define its schema and behavior. Each CR must register a `kind` and an `apiVersion`, which are used to identify the resource and its version, and it can customize the `spec` and `status` sections to define the desired and current state of the resource.

Usually, CRs are then supervised by a controller, which is a special pod that watches for changes to the CR and takes action to ensure that the actual state matches the desired state, eventually adding, modifying, or deleting other resources in the cluster or performing some other actions on external systems.

However, CRs can also be used to extend the Kubernetes API without the need for a controller, by simply defining the resource and its schema, using it as a simple data structure to store information with the advantage of an already implemented API and a consistent way to manage the resources.

API Service Resources

Kubernetes also provides a way to extend the API server itself, allowing users to define their own API services that can be used to expose CRs or to provide additional functionality, without relying on the default Kubernetes 'etcd'.

This is done through API Service resources. With this configuration, the API server acts as a proxy able to forward requests to the custom API service, which can be implemented in any language or framework, and deployed inside the cluster or externally.

If the underlying service exposes a RESTful API with the same structure of the Kubernetes API, the API server can automatically handle the requests and responses, allowing users to interact with the custom API service as if it was a native Kubernetes resource.

This allows for a seamless integration of custom services into the Kubernetes ecosystem, without relying on the default 'etcd' store, but still preserving the advantages of the Kubernetes API. Persisting the data in a persistent store, such as a database, if needed, is the responsibility of the business logic.

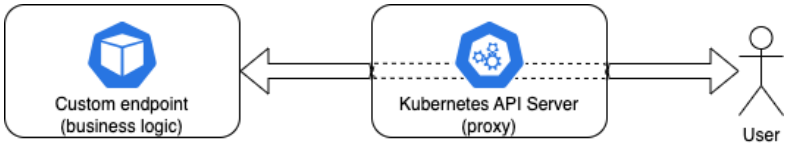

Figure 2.2: Interaction Model of API Service

2.1.2 Running Applications

Applications in Kubernetes are called Workloads, and they are the actual computing elements that run on the cluster. Basically, workloads are defined as pods, eventually running more than one container. If multiple containers are defined in the same pod, they are usually tightly coupled and share the same network namespace, allowing them to communicate with each other using 'localhost', without exposing any port to the outside world.

Usually, we do not want to run a single pod, but rather a set of pods, able to scale up and down based on the load, and to provide high availability and fault tolerance. To achieve this, Kubernetes provides a set of higher-level abstractions, each one with its own specific use case.

- **Deployments** are the most common way to run applications in Kubernetes, allowing users to define a desired state for a set of pods, and Kubernetes will ensure that the actual state matches the desired state, automatically scaling the number of pods, and providing rolling updates and rollbacks to ensure that the application is always available and up-to-date.

- **StatefulSets** are used to run stateful applications, such as databases, that require a stable network identity and persistent storage. The main difference with deployments is that Pods in a StatefulSet are assigned a unique, stable identity that can be used to directly interact with a specific replica, rather than as one casual instance of a set of pods.

- **DaemonSets** are used to run a single instance of a pod on each node in the cluster, ensuring that the pod is always running on all nodes, and providing a way to run system-level services, such as logging or monitoring agents, that need to run on all nodes in the cluster.

- **Jobs** and **CronJobs** are used to run batch jobs, that are short-lived tasks that need to be run respectively once or periodically.

Each set can schedule one or more pods, which are then managed by the Kubernetes scheduler, that is responsible for placing the pods on the most suitable nodes in the cluster, based on the available resources and the defined constraints. So, in each workload type except *DaemonSets*, the pods are not tied to a specific node, and can be scheduled on any node in the cluster, meaning that they can be moved all on a single node, or spread across multiple nodes. It is also possible to force a pod to run on a specific node, using node selectors or affinity rules, that allow users to define the placement of the pods based on labels or other criteria. This can be useful to ensure that the pods are running on nodes with specific hardware or software requirements, or to guarantee that pods of the same application are running on different nodes to provide high availability and fault tolerance.

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

Example of a Kubernetes Deployment

2.1.3 Virtual Clusters with KinD

As described in the starting of this chapter, Kubernetes is a system that is designed to run on several nodes, that can be physical or virtual machines, and run workloads on them orchestrated by the Kubernetes scheduler. However, if the main interest is to run a simple cluster for developing and testing purposes, it is possible to run a single-node cluster, that is able to run all the components of a Kubernetes cluster, including the API server, the scheduler, and the controller manager, without the need of a full-fledged cluster of machines.

This can be achieved using *KinD* (Kubernetes in Docker) [3], which is a tool that allows users to run Kubernetes clusters in Docker containers. KinD then exposes the API server so that it can be accessed from the host machine with a dedicated kubeconfig file[5], with no difference from a real cluster.

2.2 ArgoCD

Given that Kubernetes is a declarative and idempotent system, it is possible to deploy resources in a way that ensures that they are always on the same state provided by the manifest files. Kubernetes provides a way to deploy resources using the *kubectl* command-line tool, which allows users to apply, delete, or update resources in the cluster, and that provide the option '-f' that is able to read the manifest files from the local filesystem or from a remote URL and apply them to the cluster.

However, this approach is not suitable for managing complex applications because it does not provide a way to manage the state of the resources over time, and it does not allow for easy rollback or versioning of the resources. There are no built-in mechanisms to ensure that behavior is maintained. To address this issue, we can version the manifest files, that are the source of truth for the resources, and that can be stored in a standard version control system, such as GIT.

At that point, we would need a tool that is in charge of ensuring that the resources in the cluster are always in sync with what is defined in the GIT repository. This is the role of *ArgoCD* [4], that is a declarative, GitOps continuous delivery tool for Kubernetes. ArgoCD allows users to define applications as Kubernetes resources, and to manage them using GIT as the source of truth. It provides a web-based user interface and a command-line interface to manage the applications, and each application is defined as a set of Kubernetes resources that are stored in a specific folder in the GIT repository. ArgoCD then monitors the GIT repository for changes in polling mode, and

[5] A *kubeconfig* is a YAML file that contains the information needed to connect to a Kubernetes cluster, such as the API server endpoint and the authentication credentials.

automatically applies the changes to the cluster just when the resources are updated.

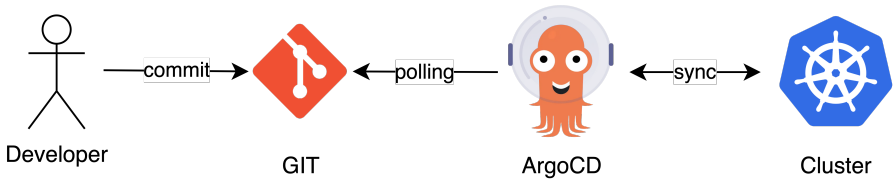

Figure 2.3: ArgoCD Workflow

2.2.1 ArgoCD Applications

The resources that should be deployed on the cluster are defined as *ArgoCD Applications*, which are themselves Kubernetes resources that tell ArgoCD how to gain information about the application and how to deploy it. There are several ways to define an application, and below we will see the most common ones.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: guestbook
spec:
  source:
    repoURL: https://github.com/argoproj/argocd-example-apps.git
    targetRevision: HEAD  # Git branch or tag to use
    path: guestbook  # Path to the resources in the repository
    directory:
      recurse: true
      include: "*.yaml"
      exclude: "config.yaml"
  # Destination cluster and namespace to deploy the application
  destination:
    name: in-cluster
    namespace: guestbook
```

Example of an ArgoCD Application

- **Helm Charts** are a popular way to package and deploy applications on Kubernetes, that rely on an Helm repository, normally managed by the Application developers, and a file of values that is used to customize the deployment.
- **Kustomize** is another popular way to manage Kubernetes resources, that allows users to define a base set of resources and then customize them for different environments using overlays.

- **Directory** is the most straightforward way to define an application, where the resources are stored in a specific folder in the GIT repository, and basically all the resources in that folder are considered part of the application[6].

2.2.2 ArgoCD with multiple clusters

ArgoCD is deployed as a set of pods in Kubernetes, and as default it is configured to manage the resources in the so-called 'in-cluster', that refers to the local installation. However, it is also possible to configure ArgoCD to manage resources in multiple clusters, allowing users to deploy applications across different Kubernetes clusters, even with a single ArgoCD installation.

To do that, it is necessary to configure the ArgoCD server to connect to the other clusters, basically providing the foreign API server endpoint and the authentication credentials to access the cluster. This can be done using the 'argocd cluster add' command from ArgoCD CLI, which will add a secret in the ArgoCD namespace that contains the information needed to access the cluster.

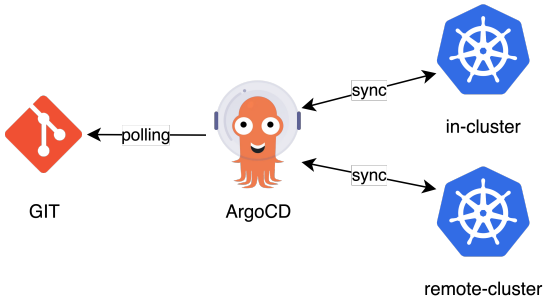

Figure 2.4: ArgoCD with Multiple Clusters

2.3 Network Operating System

Each device in the network has its own operating system, composed of the data plane, the control plane, and the management plane. Maintainers are able to interact with the device either through a command line interface (CLI) or, in some cases, through a graphical user interface (GUI). However, in a big infrastructure that relies on SDN, configuring each device manually is not affordable, so a more automated approach is needed.

[6] It is possible to exclude some files or create some include rules to better select wanted resources.

This is the role of the **Network Operating System** (NOS), and in particular of the **SDN controller**. There are several ways to create a SDN controller: in some cases, both the control and the management plane are centralized, but this can lead to errors in case of failures. A more conservative approach is to keep the control plane distributed on each device, and to centralize only the management plane. This is the approach taken by most of the SDN controllers, including the one used in *TIM* and object of this thesis. It has a centralized view of all devices in the network and is able to configure them in a consistent way, applying the appropriate configuration on each device to obtain the desired status. It is also able to monitor the devices and collect information about their status, such as the traffic statistics, the CPU and memory usage, and so on.

This centralized system must be able to interact with the devices in a standardized way, so that it can be used with different vendors and different devices. Historically, each vendor had its own way to interact with the devices, and its own operating system, that is different based on the functionality and the hardware capabilities of the device itself. Even the same vendor could have different operating systems for different devices: this is the case of Cisco[7].

The majority of the devices exposes an SSH server, that allows to connect to the device and execute commands on it. This approach is not suitable for automation, because it requires to know the exact commands to execute, differentiated by each device type. For this reason, a more standardized approach is needed, that is based on the use of protocols and data models.

2.3.1 Language of Devices: YANG

YANG is a data modeling language used to model configuration data, state data, Remote Procedure Calls, and notifications for network management protocols. Although it is defined in RFC 7950 [5] and adopted by the major vendors, there are still some differences in the implementation of the language itself, so it is not immediate that a model written by a vendor can be used with any tool using YANG.

As an example, *GoYang* [6] is a Go package that provides a YANG parser and validator, allowing to work with YANG models in Go. It is able to parse YANG models and generate Go code that can be used to interact with the devices, but does not fully support the way some of the *Cisco* models are written, with particular regard to the so-called *unified models* and the way each of them is customized by specific devices. The limitation is that, if using this library, all the configurations and queries must

[7] Cisco is a leading provider of networking equipment and software, offering a wide range of products and solutions for service providers.

be written using a subset of the original YANG models.

2.3.2 Interacting with Devices: NETCONF and gNMI

While YANG is the *language spoken* by the devices, there are two main protocols that allow to interact with them: **NETCONF** and **gNMI**.

NETCONF (Network Configuration Protocol), defined in RFC 6241 [7], provides mechanisms to configure network devices using a client-server model. It uses XML encoded messages over SSH or TLS, with transactions that support validation and rollback. The protocol defines base operations like `<get>`, `<get-config>`, and `<edit-config>`.

gNMI (gRPC Network Management Interface), specified by the *OpenConfig*[8] project [8], is a newer alternative using gRPC as transport and Protocol Buffers for data serialization. It supports both streaming telemetry and request-response interactions, with core operations including `Get`, `Set`, and `Subscribe`.

The key differences between these protocols are:

- **Efficiency**: gNMI is generally more efficient due to its compact Protocol Buffers encoding.
- **Telemetry**: gNMI has native support for streaming telemetry, while NETCONF requires additional mechanisms.
- **Transactions**: NETCONF offers a more sophisticated transaction model with explicit commit and rollback.
- **Adoption**: NETCONF has broader vendor support, while gNMI is more commonly used in modern cloud-native environments.

2.3.3 Examples of Existing SDN Controllers

In the Telco industry, several SDN controllers have emerged to address the complexity of network management, with both proprietary and open-source options available. **Cisco Network Services Orchestrator (NSO)** [9] stands as a leading proprietary solution, originating from Tail-f Systems before Cisco's acquisition. NSO utilizes a model-driven approach with NETCONF/YANG, enabling multi-vendor management through network element drivers (NEDs) that translate vendor-specific configurations to standardized models. Its key strength lies in the transaction-based configuration system

[8] *OpenConfig* is an informal working group of network operators that develops vendor-neutral data models for network configuration and management. It was formed in 2015 by major service providers including Google, Microsoft, and Facebook to address the lack of standardization in network device configuration.

that ensures atomic changes across the network, with robust validation and rollback capabilities.

Among open-source alternatives, **OpenDaylight (ODL)** [10] and **ONOS (Open Network Operating System)** [11] are widely adopted. OpenDaylight provides a modular platform supporting multiple southbound protocols including OpenFlow[9], NETCONF, and REST, while offering northbound APIs for applications. ONOS, developed with a focus on service provider networks, emphasizes high availability and scalability, making it suitable for mission-critical networks. Both platforms offer rich plugin ecosystems that enable users to customize and extend their core capabilities.

Each controller offers distinct advantages depending on specific use cases: Cisco NSO excels in service orchestration across multi-vendor environments with its commercial support and mature transaction system; OpenDaylight provides flexibility through its modular architecture and protocol support; while ONOS delivers performance optimized for carrier-grade networks.

2.4 Kubenet Framework

The *Kubenet* framework is a collection of tools designed to manage network devices using *Kubernetes* as an automation and orchestration engine [12]. It is completely open-source on GitHub and is developed by the *Kubenet* community, mostly without relying on already developed tools. The framework is designed to be modular, allowing users to select only the components they need.

The project is pretty new, with the first commits dating back to 2023, and the majority of the modules are still in development, or just in the planning phase. For the same reason, the documentation is not yet complete, and in some cases, not aligned to the actual implementation.

Since most of the developers involved in the project are *Nokia* engineers, it is fully operational with devices from this vendor, and all the examples and tests done by the team are based on their devices. However, the community shows the framework as a vendor-neutral solution, and it is designed to be compatible with any network device implementing the *gNMI* or the *NETCONF* protocol.

In the following sections, we will analyze the main components of the framework, focusing on the ones that are already available and stable.

[9]OpenFlow is a communications protocol that gives access to the forwarding plane of a network switch or router over the network, maintaining the control plane separate from the data plane.

2.4.1 SDCIO

The most stable component of the *Kubenet* framework is the *SDCIO* module [13], which is in charge of managing the connection to the network devices. It is designed to be a generic solution, allowing users to connect to any device that supports the *YANG* language and one protocol among *NETCONF* and *gNMI*. The actual implementation allows only *gNMI* connections, while *NETCONF* support is still in planning.

The module gains the running configuration of the devices and saves them in a local database, exposing them as Kubernetes resources using the *API Service* paradigm (section 2.1.1). Using the *Configurations* provided by the users, that will be discussed in section 2.4.1, it then calculates the wanted final state of the devices and applies the changes using the *gNMI* protocol. The module also provides a way to discover the devices in the network, allowing users to connect to them with minimal configuration.

Architecture

SDCIO is designed to be deployed as a Kubernetes *Deployment*, and is composed by a caching mechanism and three main components written in *Go*:

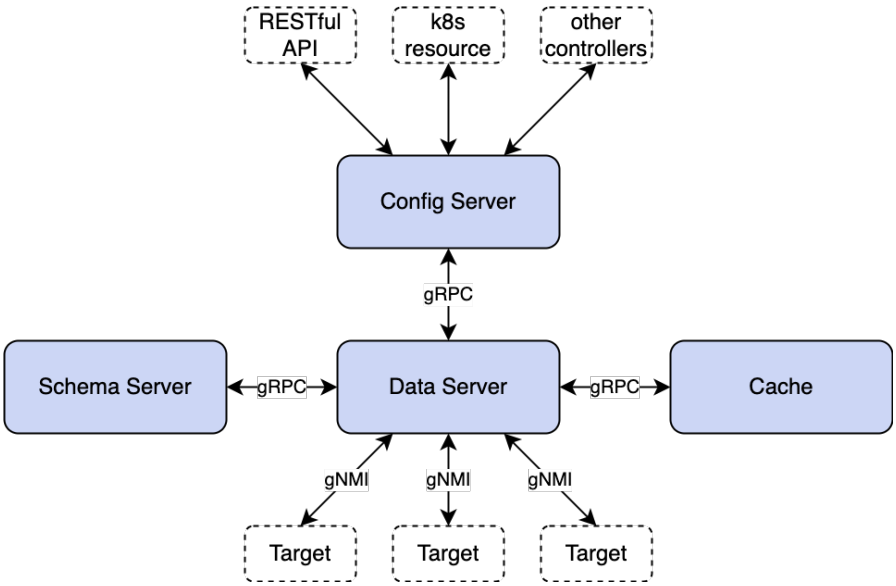

Figure 2.5: SDCIO Architecture

- **Schema server**: it has the responsibility of reading the YANG schemas and exposing them as internal resources, that are then consumed by other components of the module.

- **Config server**: it is the northbound interface of the module, exposing the device configurations as Kubernetes resources. It also allows interactions through a RESTful API. It is designed to be a *Kubernetes* controller, managing all the custom resources defined by *SDC* module.
- **Data server**: it is the southbound interface of the module, managing the connection to the devices and applying the changes to them. It is responsible for the actual communication with the devices, using the *gNMI* protocol.

The modules communicate with each other using the *gRPC* protocol, allowing them to be decoupled and independently scalable.

SDCIO also has a cache mechanism to internally store the device configurations, and intents, allowing the module to not rely on the Kubernetes API server and providing a faster access to the data. This cache is actually implemented using a file storage inside the *data-server* component, but it is planned to be replaced with a more robust solution based on *BadgerDB*[10].

Device Discovery and Connection

There are several resources that need to be created in the Kubernetes cluster to allow the *SDCIO* module to connect to the devices. The overview schema is shown in Figure 2.6, where some resources are Kubernetes native, while others are Custom Resources defined by the module.

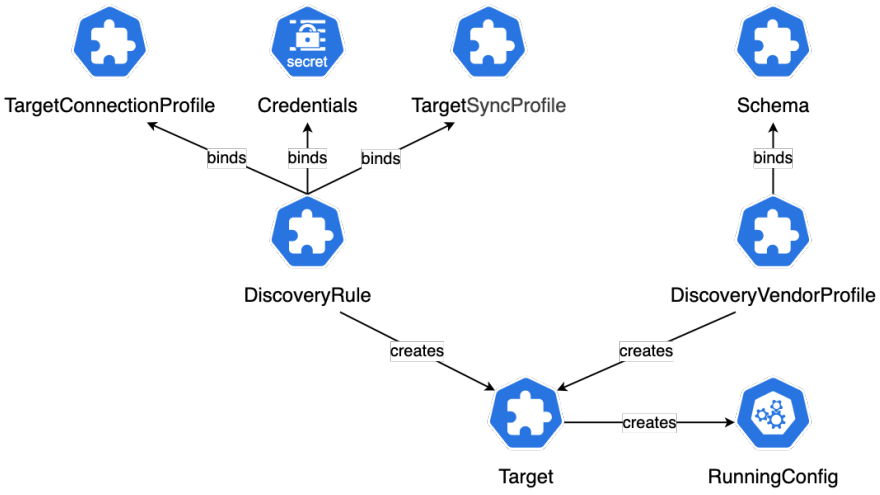

Figure 2.6: SDCIO Target Discovery and Connection

[10] *BadgerDB* is a fast key-value store written in Go and designed for high performance and efficiency.

- **Target Connection Profile**: is a CR that defines the connection parameters to the devices, such as the protocol to use, the target port, and the encoding.
- **Credentials**: is a Kubernetes secret that contains the username and password to use to connect to the devices.
- **Target Sync Profile**: is a CR that defines the synchronization parameters, such as the interval to use to poll the devices for changes. It includes parameters similar to *Connection Profile*, with the addition of all the YANG paths that shall be monitored and synced.
- **Schema**: is a CR that defines where to find the YANG schemas to use to parse the device configurations. It uses a GIT repository as a source, and can have two different paths for *schemas* and *includes*. It is possible to include an entire folder or single files. This feature can be particularly useful to include only a subset of the schemas.
- **Discovery Rule**: is a CR that defines where to discover the devices in the network. It includes a reference to *Target Connection Profile*, *Credentials*, and *Target Sync Profile*, and the targets can be discovered with their direct IP address, or scanning an entire subnet, defined as a CIDR.
- **Discovery Vendor Profile**: is a CR that defines some characteristics of the device based on its vendor and model. It is used to map each device that matches the discovery rule to a specific Schema, allowing the module to use the correct YANG schemas to parse the device configurations. It contains the YANG paths for version, platform, hostname, and other information that can be used to identify the device and will be reported in the Target resource.
- **Target**: is a CR that represents the device in the Kubernetes cluster. It is created by the *SDCIO* module after discovering the device and contains all the information about it, such as the vendor, model, and connection parameters. It also contains a reference to the *Discovery Vendor Profile* used to parse the device configuration.
- **Running Config**: is an API Service resource that contains the running configuration of the device, parsed using the YANG schemas defined in the *Schema* resource, on paths defined in the *Target Sync Profile*. It is updated periodically based on the *Target Sync Profile* defined for the device.

Configurations

Once the Target resource is created, the *SDCIO* module starts managing the device configurations. All the resources related to configurations are defined as API Service

resources, using the paradigm described in section 2.1.1.

One or many *Configuration* objects can be created, each one for a specific purpose, and containing the desired state of the device. The Configuration is linked to a specific Target through a couple of special labels, and several values referenced to different YANG paths can be defined in the same resource. Creating multiple configurations that act on the same YANG paths and the same device can lead to conflicts, that are managed by the module using a priority system. Each *Configuration* is associated with a specific *Priority*, and the final configuration applied to the device is calculated taking the configuration with higher priority for each specific field.

The following example shows the resource needed to configure an OSPF process on a Cisco device, with the process name, router ID, and interfaces.

```
apiVersion: config.sdcio.dev/v1alpha1
kind: Config
metadata:
  name: conf-edge01-svc1
  labels:
    config.sdcio.dev/targetName: edge01
    config.sdcio.dev/targetNamespace: default
spec:
  priority: 10
  config:
  - path: /router/ospf/processes
    value:
      process:
        - process-name: "1"
          router-id: 10.10.10.10
          areas:
            area:
            - area-id: 0
              interfaces:
                interface:
                - interface-name: Loopback0
                - interface-name: GigabitEthernet0/0/0/1
```

Example of Configuration in SDCIO

Configurations Lifecycle

One of the major problems when porting configurations to a Kubernetes-like system is the idempotency of the operations. *SDC Configurations* are designed to be idempotent, meaning that applying the same configuration multiple times will not change the device state, and when a configuration is applied, the state of the device reflects

what is defined in the configuration, as long as the configuration is valid, and then removed when the original resource is deleted. The *data server* is in charge of detecting any sort of changes or deletions and convert them into *gNMI* operations to apply to the device.

This is the default behavior of the module, but in some cases there is the need of applying a configuration that should remain on the device for all its lifetime, without the risk of being removed unintentionally. For this reason, the **lifecycle** property can be added to the configuration, that accepts two values:

- **Delete**: the default value, meaning that the configuration is applied to the device and removed when the resource is deleted.
- **Orphan**: the configuration is applied to the device and remains there until explicitly removed[11], even if the resource is deleted.

2.4.2 KUID

KUID is a cloud-native application that extends the *Kubernetes* API, dedicated to manage network and infrastructure resources within *Kubernetes* environments. It focuses on inventory, IP addresses, VLANs, AS numbers, and similar resources required by network services [14]. By leveraging the *Kubernetes*-native architecture and customizable fields, *KUID* facilitates streamlined resource organization and tracking.

The module offers robust IP Address Management (IPAM) capabilities for efficient allocation and oversight of IP resources across the infrastructure. Additionally, it provides sophisticated infrastructure management functionalities, empowering users to organize and manage various infrastructure components within a structured hierarchy, aligning with the cloud-native approach of the *Kubenet* framework.

Similarly to the *SDCIO* module, it is designed to be a Kubernetes controller, managing all the custom resources defined by *KUID* module, and deployed as a Kubernetes *Deployment*.

2.4.3 Choreo

Choreo is an advanced open-source orchestration framework designed to simplify and enhance network automation and orchestration [15]. While built on the principles of *Kubernetes*, it does not rely directly on *Kubernetes* resources, but rather leverages

[11] The only way to delete a configuration with *Orphan* lifecycle is to apply another configuration that explicitly deletes or resets the wanted YANG fields.

the *Kubernetes* Resource Model (KRM) to define its own custom resources. This approach provides a familiar declarative syntax for network engineers, without requiring an actual *Kubernetes* cluster for operation.

The framework utilizes an event-driven and declarative approach to automation, allowing users to define the desired state of their network infrastructure and letting the system handle the details of how to achieve that state. This aligns with the overall philosophy of the *Kubenet* framework, emphasizing configuration as code and declarative models.

The module is still in development, many features are not yet stable, and it is difficult to have a complete overview of the project, since some features are not yet implemented.

Project Structure

The *Choreo* project works with files organized in a specific structure, that enables the module to understand the relationships between different resources.

- **crd**: contains the Custom Resource Definitions (CRDs) that define all the resources the module interacts with.
- **in**: contains the input files that define the business logic, such as input data and reconcilers.
- **out**: is the folder used to store the output files generated by the module, such as the final configurations to apply to the devices.
- **refs**: can contain entire submodules, that are defined in the same way as the main module, but can be used to extend the functionalities.

Architecture

By default, *Choreo* is not shipped containerized, but as an executable file that can run in *Dev* or *Prod* mode: the difference is that in *Dev* mode the module runs when the user requests it, while in *Prod* mode it runs as a long-running process, listening for changes in the input files and reconciling them with the output files.

A first instance of the module is launched to act as a *Server* who reads the input files and waits for instructions from the user. A second instance is launched to act as a *Client* that tells the server to reconcile the input files and generate the output files.

Reconcilers

Reconcilers are defined in a similar way to the *Kubernetes* controllers, allowing each resource to watch, own, and act on specific resources defined in the configuration

file. The logic is to provide an interface that is simple to use, also for users that are not familiar with controllers programming.

Basically, a reconciler is a function that takes an input file (single resource) and generates one or more output files (one or more resources) based on the business logic defined in the module. *Choreo* provides different ways to define reconcilers:

- **Starlark**: a Python-like language that is easy to learn and use. The file defines the output files to generate, using a specific syntax to reference input files. *Choreo* provides a set of built-in functions useful for network automation.
- **Jinja2**: a templating engine that allows users to define templates with placeholders that are replaced with actual values from the input files. It is widely used in the Python ecosystem and easy to learn. The limitation of this approach is that it cannot define complex business logic, and each reconciler can generate only one output file.
- **GoTemplate**: similar to Jinja2 templates, but more powerful and allows users to define complex business logic.

Among these, *Jinja2* templates are the most used in other environments due to their ease of use and popularity in the Python ecosystem. The *Choreo* application of Jinja2 relies only on the builtin functions and filters, though Jinja2 permits extending the functionality with custom filters and functions, allowing implementation of more complex business logic.

2.4.4 Pkgserver

According to what its developers claim, *Pkgserver* would be a component of the *Kubenet* framework designed for streamlined Kubernetes Resource Model (KRM) package management in conjunction with GitOps systems [16]. However, like several components of this framework, the module is still in an embryonic state of implementation and with very limited documentation.

Based on the creators' vision, the module should facilitate continuous delivery practices for network infrastructure, ensuring reliable and consistent rollouts of configuration changes to network devices. Whether managing infrastructure in development or production settings, *Pkgserver* should provide the necessary tools to automate the deployment process, following the declarative approach that characterizes the entire *Kubenet* framework. Despite these promising features, the current implementation is largely incomplete and difficult to use in real-world contexts. The lack of concrete examples and technical documentation makes it complex to understand the actual functionalities already developed and those still in the planning phase, significantly limiting the adoption of this module.

Chapter 3

Services in Telcos

As a customer, we usually see the ISP as a simple 'plug' that, in some way, connects us to the Internet. We also know that the ISP is able to give us several services, such as a private connection between two sites of the same company, but we consider it as a black box.

Each Telco has its own organization and uses a different set of vendors for their infrastructure, but we can abstract some common elements. Normally, there is a clear separation between the *network infrastructure* and the *services delivery system*. The first one is responsible for the physical and logical connectivity between the nodes of the network, providing a robust and reliable communication channel. The second one, that can rely on the first one, is responsible for the delivery of services to the customers, such as Internet access, private connections, and so on.

While the interest of this Thesis is focused on the services part, we must not forget the importance of the underlying infrastructure, specially because we need a similar underlying infrastructure to run our experiments.

3.1 Network Topology

A Telco network is composed of thousands of nodes, connected in a complex topology that is designed to provide high availability and redundancy, while preserving the costs of the infrastructure. All these nodes are usually divided from a logical point of view into several zones, that might have a relationship with the geographic location of the nodes, but not necessarily.

There are then two main layers in the network: the **backbone** and the **edge**. Customer endpoints are connected only to the latter one, while the backbone is responsible for the interconnection of the edge nodes and are connected only to other nodes of the operator. Services are usually delivered over the edge nodes, so the backbone

is not directly involved in the delivery of services to the customers, even if the real traffic will traverse this layer.

Above all these nodes, a dynamic routing protocol is used to ensure that the traffic is routed correctly, even in case of failures. Each node is recognized in the network by a unique identifier, that is an IPv4 address assigned to a specific loopback interface[1]. MPLS is then used to allow the creation of virtual circuits between the nodes, that can be used to deliver services to the customers, optimizing the traffic over the whole network.

In our study case, we will focus on a single vendor, assuming that all the nodes of the network are Cisco devices.

An example of a Telco network topology is shown in Figure 3.1, where the backbone links are represented with solid lines, while the edge links are represented with dotted lines.

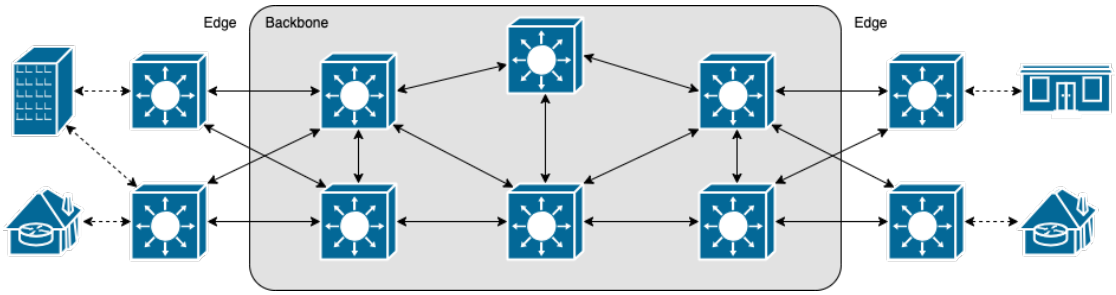

Figure 3.1: Example of Telco Network Topology

3.1.1 Topology Simplification for Experiments

In order to simplify the network topology for our experiments, we will create a reduced version of the original topology. This simplified topology will retain the essential characteristics of the original one, while removing unnecessary complexity.

From the *services* point of view, the number of nodes in the backbone is not important as long as there is a connection between the edge nodes. For this reason, we could consider a single backbone node, and the edge nodes which the customers are connected to, obtaining a topology like the one shown in Figure 3.2.

If we go deeper into the abstraction, we can assume that all the services are directed from one edge node to another one, that can eventually be the same one, but

[1]A loopback interface is a virtual interface that is always up and reachable, even if the physical interfaces are down. For this reason, it is used to provide an address that is always reachable, even in case of failures.

definitely not the backbone node. So, the backbone is not involved into the service itself, but it simply guarantees the connectivity between the edge nodes. In our case, we can then consider the two edge nodes as the only ones and directly connected, as shown in Figure 3.3.

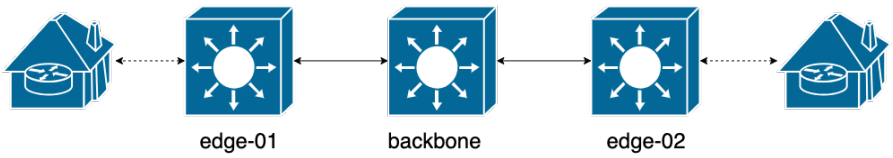

Figure 3.2: Simplified Telco Network Topology

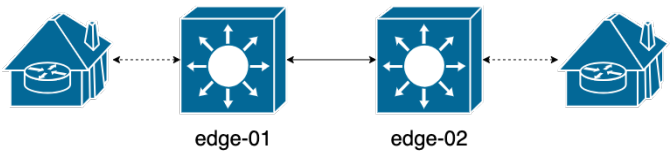

Figure 3.3: Minimal Telco Network Topology

3.1.2 Topology Emulation: Containerlab

Containerlab [17] is a tool that allows to emulate network topologies using Docker containers, providing a lightweight and flexible environment for testing and experimentation.

All the wanted topology is described in a YAML file, as an array of nodes and an array of links between them. When deployed, each node becomes a Docker container, and virtual links are created between them.

Keeping focused on *Cisco* devices, we can use the *Cisco IOS XRd* image [18], that is a lightweight version of the Cisco IOS XR operating system, designed for emulation purposes. It is shipped in two versions: one with the full set of features (*Cisco XRd vRouter*), and one that has only the control plane, and a very simplified data plane (*Cisco XRd Control Plane*). It is possible to use the latter one to reduce the resource consumption, considering that we are not interested in the real traffic forwarding, but only in the configuration and management of the devices.

Finally, we can use simple Linux containers to represent the customer endpoints, that will be connected to the edge nodes of the network. In this way, we can easily simulate the customer traffic and test the network services.

```
name: topo2nodesxrd
mgmt:
  mtu: 1500
  network: kindnet
  ipv4-subnet: 172.21.0.0/16
topology:
  kinds:
    linux:
      image: ghcr.io/hellt/network-multitool
    cisco_xrd:
      image: ios-xr/xrd-control-plane:7.11.2
  nodes:
    edge01:
      kind: cisco_xrd
      type: ixrd2
    edge02:
      kind: cisco_xrd
      type: ixrd2
    client1:
      kind: linux
    client2:
      kind: linux
  links:
    - endpoints:
      - "edge01:Gi0-0-0-1"
      - "edge02:Gi0-0-0-1"
    - endpoints:
      - "client1:eth1"
      - "edge01:Gi0-0-0-2"
    - endpoints:
      - "client2:eth1"
      - "edge02:Gi0-0-0-2"
```

Containerlab Topology for Minimal Telco Network

3.2 Customers Services Delivery System

When talking about services, the interest is not focused on the *network infrastructure*, but on the willing to have an easy to use and reliable system that allows operators to deliver services to the customers.

In this optic, the unification of the management of all the devices in a single system, as seen in the previous section, is crucial, but not enough. The system should be able to receive intents from the operators and translate them into configurations for the

devices before applying them across the network. This is the role of yet another component (*Service Abstraction Layer* in Figure 3.4), that is in between the *SDN controller* and the creation of services, that could be done directly by a human operator, or by an automated system that is able to create services based on the customer requests. The interaction model of this system is shown in Figure 3.4.

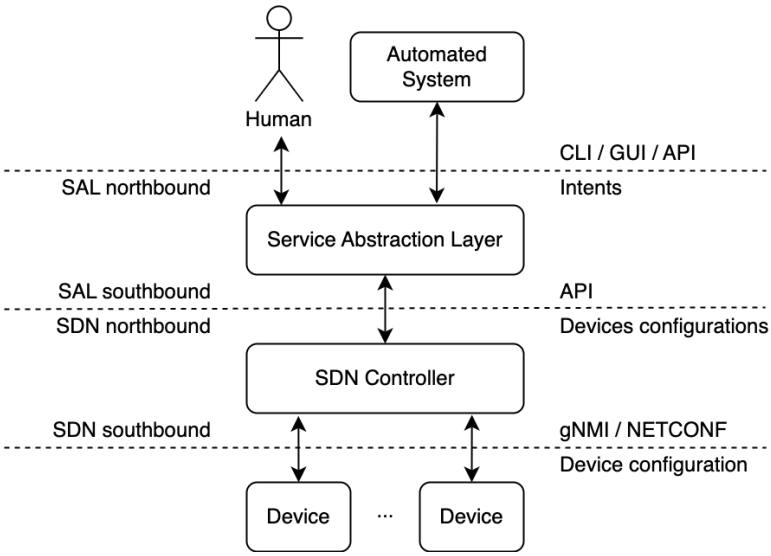

Figure 3.4: Service Delivery System Interaction Model

3.3 Examples of Services

It is now possible to define some examples of services that can be delivered to the customers, using the previously defined systems. For the next examples, we will consider the simplified topology shown in Figure 3.3, where the two edge nodes are directly connected to each other, and the customers are connected to them. Clients node in the network are basic Linux containers, while all the others are Cisco devices.

3.3.1 L2 MPLS Tunnel

A **Layer 2 MPLS Tunnel** is a service that allows to connect two customer endpoints over a virtual circuit, using the same VLAN tag on both sides. This service is useful to create a private connection between two sites of the same company, or to connect two customers that need to communicate with each other.

For the network, this is deployed using a *Cisco Xconnect* service, that is a virtual circuit that connects two endpoints over the network using MPLS. It can be deployed

on *Cisco IOS XRd* devices using the following configuration written in *Cisco IOS XRd* configuration language, deployed on both edge nodes, where the keyword prefixed with $ are variables that must be customized for each service instance.

```
config
  interface $port.$serviceInstanceId l2transport
    description $serviceInstanceDescription
    encapsulation dot1q $VLANId
    rewrite ingress tag pop 1 symmetric
    exit
  l2vpn
    xconnect group svcGroup
      p2p $pseudowireId
         interface $port.$serviceInstanceId
           neighbor $destinationLoopbackAddress pw-id $pseudowireId pw-class CW
           exit
         exit
      exit
    exit
  exit
```

Cisco IOS XRd Configuration for L2 MPLS Tunnel

The simplest configuration is shown in Figure 3.5, but the versatility of the service allows to create more complex scenarios, such as connecting one customer to multiple endpoints, as shown in Figure 3.6, or connecting two endpoints that are physically associated at the same edge node, as shown in Figure 3.7. In all the Figures, VLANs and pseudowire IDs are examples.

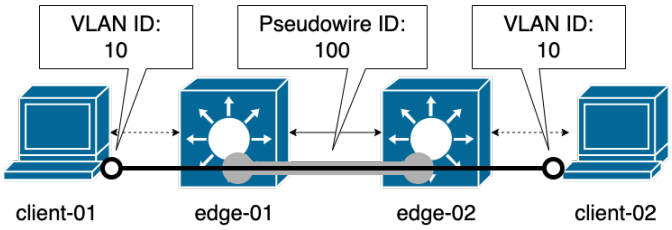

Figure 3.5: L2 MPLS Tunnel Service

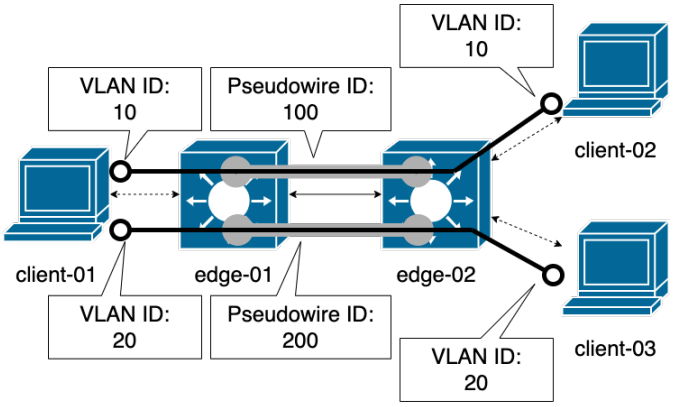

Figure 3.6: L2 MPLS Tunnel Service with Multiple Endpoints

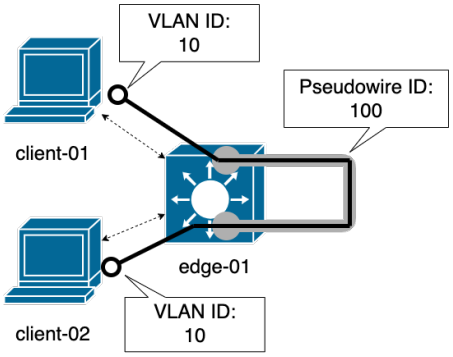

Figure 3.7: L2 MPLS Tunnel Service with Same Edge Node

3.3.2 L2 Tunnel with Bandwidth Control and Monitoring

The previous service can be customized in several ways. This second example shows a Layer 2 Tunnel service similar to the previous one, but terminated on both sides without a VLAN tag. This means that a single service instance can be attached to the same endpoint.

This service is also equipped with a bandwidth control and monitoring system, that is crucial for the Telco operators to ensure that the traffic is correctly managed and that the service is not overloaded by the customers.

The goal is obtained using *Cisco policies* and *SLA profiles*, that are used to define the bandwidth limits and the monitoring parameters. Unfortunately, these configurations are not supported by the *Cisco XRd Control Plane* image, so it is necessary to use physical devices to test this service.

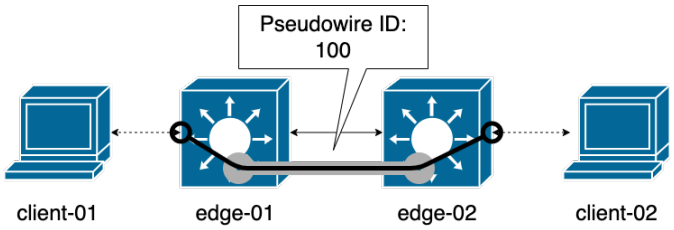

Figure 3.8: L2 Tunnel with Bandwidth Control and Monitoring Service

Chapter 4

Framework Architecture

Knowing the best features of all the tools explored in the previous chapters and their limitations, we can make informed decisions about which technologies to adopt. In this chapter, we will outline the proposed framework architecture, focusing on the key components and their interactions.

The general approach is to use established technologies where possible, growing-up tools when necessary, and creating new ones when needed. The final chain of tools will be a mix of existing and new components, allowing the user or the upper layer to provide simple services configuration snippets, and make the framework automatically push into each device the right configuration to obtain the desired state.

4.1 Overview

The proposed interaction model, as illustrated in Figure 4.1, consists of several modules, each one with a specific function. Following a top-down approach, we will have:

- **Humans** or **External tools** can trigger the entire process by simply adding a specific file in a dedicated GIT repository, and pushing it to the remote server.
- A first GIT repository (named **Services** in the figure) will act as a central database, able to version[1] all the services required, providing also accountability[2].
- When new files are added, they trigger the **Config Translator**, which is in charge

[1]Versioning refers to the ability to track changes and maintain different versions of the services configurations over time.

[2]Accountability refers to the ability to trace back changes to specific users or processes, ensuring that all modifications are logged and can be audited.

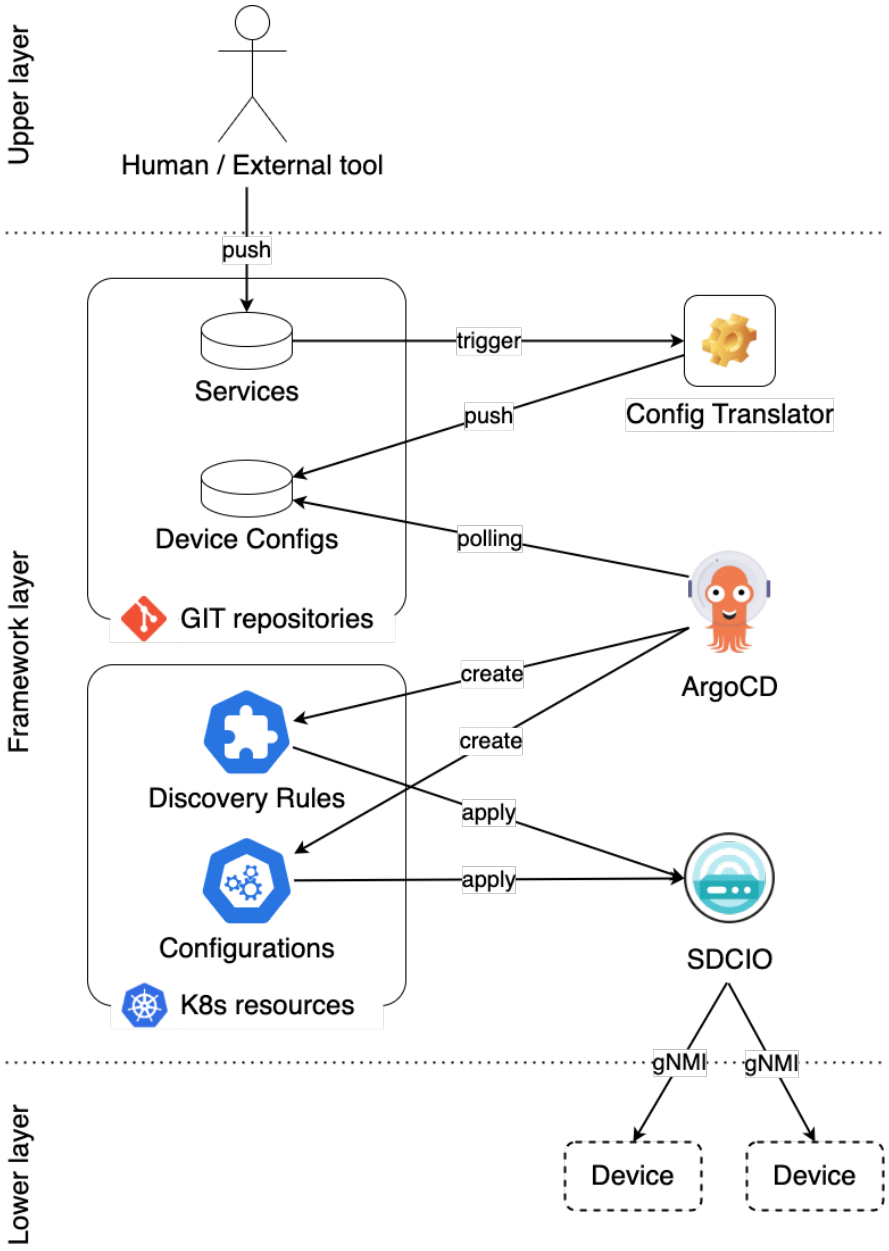

Figure 4.1: Framework Architecture Overview

of parsing the files and translating them into a set of SDCIO resources (as described in section 2.4.1). After storing them as YAML files, they are pushed to a second GIT repository.

- The second GIT repository (named **Device Configs** in the figure) will act as another database, storing all the discovery rules needed to connect to the involved devices and all the service-specific configuration files.
- These configurations are then read by **ArgoCD** that creates the associated resources in the right k8s cluster, as described in section 2.2.
- The applied resources finally trigger the **SDCIO** reconciler, that calculates the wanted configurations in each device, translates them in YANG modules, and pushes them to the devices using the *gNMI* protocol, as described in subsection 2.4.1.

From the external point of view, the framework exposes two standard interfaces, providing an easy integration in other processes. The upper layer is a simple GIT repository, and the interaction is made with a commit-and-push model. In the bottom layer, the interaction is based on *YANG* schemas and *gNMI* protocol, which is a growing-up standard for network management, adopted by several vendors and open-source projects.

Internally, each component interacts with others using standardized interfaces, so that it will be simple to replace one element in the chain if needed. *ArgoCD* is a widely-used tool, while *SDCIO* is the most stable component of the *Kubenet* framework, even if still in development. The last component, the *Config Translator*, is completely new and implemented in this proposed solution.

4.2 Involved GIT Repositories

The most of the storage is handled through GIT repositories. It is distinguished in two different and independent repositories since they have different scopes and utilizations. The *Services* one is managed by the external tool (or the human), so the framework has only to read from it. Instead, the *Device Configs* repository is managed internally by the framework, which is responsible for reading and writing to it.

Another possible solution is to use a single repository for both services and device configurations, but this would require careful management of the folder structure and access controls. Moreover, it would create a big data structure, exposing also internal details to the upper layer, which is not interested in it. The two repositories approach also facilitates the management of conflicts, since there are less entities involved in the editing of a single repository.

4.2.1 Services Repository

The *Services* repository contains the data of all the devices involved in the process, all the details of the services to be applied, and the business logic responsible of the translation from the user-defined resources to the SDCIO resources. Each service is translated by a dedicated *Translator*, whose details will be exposed in subsection 4.3.1.

There is a great degree of flexibility in the organization of the repository, allowing users to better customize it based on their needs. However, a base structure is provided to better maintain consistency and organization, composed of three main folders:

- **data**: contains all the definitions of what needs to be translated in actual device configurations, including all the related data. This folder can be further divided into subfolders to better organize the content: in the example below, there are the **services** and **nodes** subfolders, but it is possible to organize them as needed.
- **nodes**: contains all the information about the devices involved in the process. As the devices need to be mapped to a *Discovery Rule*, they themselves are part of the first category (*data*); this folder is consequently a subfolder of **data**.
- **translators**: contains all the translation logic for each service, implemented as separate modules.

The following is an example of structure of the *Services* repository.

```
repository
├── data
│   ├── nodes
│   │   ├── edge01.yaml
│   │   └── edge02.yaml
│   └── services
│       ├── service1-customerA.yaml
│       ├── service1-customerB.yaml
│       ├── service2-customerA.yaml
│       ├── service2-customerC.yaml
│       └── custom-subfolder
│           └── service3-customerD.yaml
└── translators
    ├── device
    │   └── ...
    ├── service1
    │   └── ...
    └── service2
        └── ...
```

The relative paths of each of that folders can be customized as we will see in subsection 5.2.6.

While the specific structure of the *Service* files may vary depending on the needs, the *devices* configuration is more rigid, because some of the data present in these files might be used by other translators, as we will see in subsection 4.3.3. An example of device configuration file is the following:

```yaml
apiVersion: infra.telecomitalia.it/v1alpha1
kind: Node
metadata:
  name: edge01
spec:
  connectIP: 172.21.0.6
  loopbackIP: 10.10.10.10
  zone: zone0
  type: cisco-xrd
```

Example of Device configuration

```yaml
apiVersion: service.telecomitalia.it/v1alpha1
kind: L2Link
metadata:
  name: edge1-2
spec:
  description: "L2 Link between two edge nodes"
  firstEndpoint:
    deviceName: edge01
    interface: eth5
  secondEndpoint:
    deviceName: edge02
    interface: eth3
```

Example of Service configuration

4.2.2 Device Configs Repository

The *Device Configs* repository has a more rigid structure, as it is designed to be used only internally by the framework, and so there is no need to deeply customize it. The *Config Translator* acts as the only writer for this repository, while *ArgoCD* is the consumer. However, there is always the ability to access directly for monitoring purposes.

The first aim of this component is to provide configuration files using a structure that can be easily managed by *ArgoCD*. It means that files can be stored in different directories, that will be then mapped to different *ArgoCD Application*. For example, this feature can be used to separate devices *Discovery Rules* and services configuration, so that they will remain in a distinct structure also in the *ArgoCD* dashboard, assuring better visibility and management.

The second aim is acting as database for *Config Translator* component, preserving some information about the configurations already applied. In particular, it keeps track of two main aspects:

- **Last Applied Commit**: a reference to the last commit of *Services* repository that has been translated. As we will see in subsection 5.2.1, this feature is particularly important to guarantee the minimum use of computational resources, avoiding to re-translate configurations that have not changed.
- **Service log file**: for each executed translation, a file is maintained with the references to all the commits which changed the configuration and the timestamp of the calculation, and the list of created files. This is particularly useful in configuration deletion, allowing to easily identify all the resources that need to be cleaned up.

In addition, as this repository is the one read by *ArgoCD*, there is a special folder that contains all the *ArgoCD Application* definition. Inserting all these configuration files in a single directory means that the only application that would be created in the *ArgoCD* dashboard is the one configured to read this directory, and all the defined *ArgoCD Applications* will become sub-applications and then loaded automatically. Examples of repository structure and principal files are shown below.

```
repository
├── applications
│   ├── infra.yaml
│   └── services.yaml
├── configurations
│   ├── infra
│   │   ├── edge01-discoveryrule.yaml
│   │   └── ...
│   └── services
│       ├── service1-customerA-edge01.yaml
│       ├── service1-customerA-edge02.yaml
│       └── ...
└── translator-db
    ├── applied-configs
    │   ├── nodes
    │   │   ├── edge01.yaml
    │   │   └── ...
    │   └── services
    │       ├── service1-customerA.yaml
    │       ├── service1-customerB.yaml
    │       └── ...
    └── last-applied-commit.txt
```

```yaml
metadata:
  name: /services/service1-customerA.yaml
status:
  managedFiles:
  - service/service1-customerA-edge01.yaml
  - service/service1-customerA-edge02.yaml
  updates:
  - date: 2025-07-30 16:04:20.285463942 +0200 CEST
    commit: b5eed70d57fd7173f0b0b582876039d0b985bdef
```

Example of Service Log file

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: infra
  namespace: argocd
spec:
  destination:
    name: in-cluster
    namespace: configurations
  source:
    repoURL: https://my-git-endpoint.com/device-configs.git
    targetRevision: HEAD
    path: ./configurations/infra
  project: default
  syncPolicy:
    automated:
      prune: true
    syncOptions:
    - CreateNamespace=true
```

Example of ArgoCD Infrastructure Application file

4.3 Config Translator

The role of *Config Translator* module is similar to the one offered by *Choreo* in the context of *Kubenet* framework, as discussed in subsection 2.4.3. Therefore, it will have similar paradigms, including the use of templates and a GIT-based approach, while several other features, that were not particularly useful in our context, have been removed.

We decided to implement a new module, rather than using the existing one, mainly because it was in an embryonal phase, and so not usable in production, and also because we wanted to have a tool tailored to our specific needs.

The module is triggered whenever a change is detected in the upstream GIT repository, and runs one or more[3] *Translator* in order to translate the configurations to the downstream repository. There are two principal ways to trigger a program when changes are detected in a repository:

- **Active mode**: the program is always active and checks for changes in the repository at regular intervals. This is the approach used by *ArgoCD* and has the advantage of being simple to implement, but it can lead to delays in processing changes and a waste of resources as it continuously checks for updates even when there are no changes.
- **Passive mode**: the program is triggered by webhooks or other event-driven mechanisms when changes are detected. This can be done exploiting the CI pipelines[4], and is more efficient, as it allows the program to react immediately to changes, but generally requires permissions to be set up.

4.3.1 Translators

The *Translator* is the minimal module that has the logic to translate configurations from the upstream repository to the downstream one. Its role is essentially to map the configuration elements from a single service file description into one or more *SDCIO* configurations.

Jinja2 is used as template engine, thanks to its simplicity and flexibility. It allows to parse information from an external source, using exports and includes to provide modularity and reusability, and allows to execute some simple transformation functions to manipulate the data thanks to its powerful filtering capabilities, that brings extendability to the system [19].

Each translator is defined in a dedicated folder into the *Services* repository, and is composed of a YAML configuration file, one or more *Jinja2* 'main' template files, and eventually other *Jinja2* supporting files. Each of the *Jinja2* 'main' files creates a different resource, allowing to create multiple resources from a single service file.

The configuration file is YAML based, but it is not a k8s resource: there are no specific k8s fields or metadata, even if it uses a similar approach.

[3] Each *Translator* is responsible for a specific translation task. If multiple translation tasks are needed, then multiple *Translator* instances will be created, eventually acting on the same configuration files.

[4] Continuous Integration pipelines, a way to automatically run elements based on GIT events

```yaml
spec:
  for:
    group: service.telecomitalia.it
    version: v1alpha1
    kind: L2Link
  files:
  - first-endpoint.main.jinja2
  - second-endpoint.main.jinja2
```

Example of translator configuration

The generated files have no specific rules enforced by the module: they only have to follow the rules of k8s resources and *SDCIO*. The only constraint when writing a *Translator* is to add a couple of labels in the metadata:

- `config-translator.telecomitalia.it/targetDevice`: the module uses the *targetDevice* field to identify the device to configure and define properly in which *ArgoCD* application it should be placed;
- `config-translator.telecomitalia.it/kind`: the kind of resource being created (e.g. *infra*, *service*) is also used to determine the right *ArgoCD* application in which the resource should be placed.

4.3.2 Example of Translator: Devices

Given a device description file, already seen in Listing 4.2.1, we want to generate the associated *Discovery Rule* to allow the system to discover the device and then add configurations to it. The generated *Translator* is composed of YAML configuration and a single *Jinja2* template file, as shown below.

```yaml
spec:
  for:
    group: infra.telecomitalia.it
    version: v1alpha1
    kind: Node
  files:
  - discoveryrule.main.jinja2
```

Device translator configuration

```
apiVersion: inv.sdcio.dev/v1alpha1
kind: DiscoveryRule
metadata:
  name: dr-{{ metadata.name }}
  labels:
    config-translator.telecomitalia.it/targetDevice: {{ metadata.name }}
    config-translator.telecomitalia.it/kind: infra
spec:
  period: 1m
  concurrentScans: 2
  addresses:
  - address: {{ spec.connectIP }}
    hostName: {{ metadata.name }}
  discoveryProfile:
    credentials: cisco-cred
    connectionProfiles:
    - conn-gnmi-insecure
  targetConnectionProfiles:
  - credentials: cisco-cred
    connectionProfile: conn-gnmi-insecure
    syncProfile: gnmi-cisco
  targetTemplate:
    labels:
      sdcio.dev/region: {{ spec.zone }}
```

Jinja2 DiscoveryRule template

This is the most basic implementation of this translator. It is possible to extend it by adding some more options, for example changing the *Discovery Profile* based on the device Vendor and Model.

The output of this translator, assuming the input provided in Listing 4.2.1, is shown in the following page.

As described in section 2.4.1, the *Discovery Rule* is not enough to properly setup a device. We assumed that the depending resources are already deployed on the target cluster, but the presented solution can be extended to also create all the needed resources.

```
apiVersion: inv.sdcio.dev/v1alpha1
kind: DiscoveryRule
metadata:
  name: dr-edge01
  namespace: default
  labels:
    config-translator.telecomitalia.it/targetDevice: edge01
    config-translator.telecomitalia.it/kind: infra
spec:
  period: 1m
  concurrentScans: 2
  addresses:
  - address: 172.21.0.6
    hostName: edge01
  discoveryProfile:
    credentials: cisco-cred
    connectionProfiles:
    - conn-gnmi-insecure
  targetConnectionProfiles:
  - credentials: cisco-cred
    connectionProfile: conn-gnmi-insecure
    syncProfile: gnmi-cisco
  targetTemplate:
    labels:
      sdcio.dev/region: zone0
```

DiscoveryRule output

4.3.3 Obtain Devices Specific Information

The Service Configurations often needs specific information about one of the target nodes to be properly defined, but this information is normally not available nor needed at the upper layer. As an example, to configure a L2 tunnel, the Config must include the neighbor loopback IP, but the upper layer wants to refer to it using its name rather than the IP address.

The association between the device name and its IP address is already known by the *Config Translator* module because it is stored in the node description, as seen in the Listing 4.2.1. It is possible to define specific *Jinja2* filters that outputs the device loopback IP address based on its name.

Where we want to insert the IP address, we can simply write:

```
{{ spec.deviceName | nodeLoopback }}
```

where `spec.deviceName` is the name of the device we want to obtain the loopback IP for, and it is automatically translated to its loopback IP address thanks to the `nodeLoopback` filter.

4.4 Architecture Scalability

A framework that aims to manage thousands of nodes, with much more services, must be able to scale properly. To ensure that, each element in the chain must be designed with scalability in mind. For the most common and worldwide used tools, such as GIT and ArgoCD, scalability has been thoroughly tested and proven in large-scale environments, and there are several scientific studies that analyze their performance and scalability limits.

- **GIT** is known to handle large repositories with ease. In the 'Git is for Data' paper, published in 2023 [20], its performance has been benchmarked for repositories containing over 550M objects, observing that it could also be scaled to billions of objects following some optimizations, as developed in GitHub [21].

- **ArgoCD** is designed to handle large-scale GitOps workflows and is described as the best GitOps tool for Kubernetes referring to its performance and scalability in an analysis completed in 2025 [22].

- Kubernetes **etcd** is known to not scale well with large amounts of data, as it is designed to handle small to medium-sized key-value pairs. This limitation can impact the performance of Kubernetes resources that rely on etcd for storage and retrieval. However, the *SDCIO Config* resources does not rely on that distributed system, as discussed in section 2.4.1, since it is implemented through a k8s API Service. The *SDCIO Discovery Rules*, instead, rely on etcd for storing and retrieving the discovered device information. However, since the number of discovery rules is relatively small compared to the overall number of Kubernetes resources, the impact on scalability is limited. This limitation can also be mitigated thanks to the *SDCIO* scalability solution proposed in subsection 4.4.2.

The same performances must be addressed by the *Config Translator* and *SDCIO* components. The first one is completely new, so it is not possible to refer to existing studies, but it is crucial to ensure that it can handle large-scale configurations efficiently. The second one is not yet in production, so its performance characteristics are still under development, and we need to apply some strategies to prove its scalability.

4.4.1 Config Translator Scalability

The *Config Translator* is already modular by design: each *Translator* is responsible for a specific type of configuration, and does not depend on what other translators do, making it easier to scale and maintain. As the number of supported services grows, new translators can be added without impacting the existing ones, ensuring that the system can evolve and adapt to changing requirements.

As it was presented in section 4.3, all the translators are run by a single controller, that reads information from the upstream repository and writes its output to the downstream repository. It reads the content of the `translator` folder to understand what it has to do, and this pattern can be used to customize the behavior to better suite a scalable architecture. Splitting the translators in several subfolders, it is possible to run a different controller for each subfolder, allowing to run multiple instances in parallel. It can run without conflict, because every translator generates distinct files, given that each translator is configured to store last commit information in a different location.

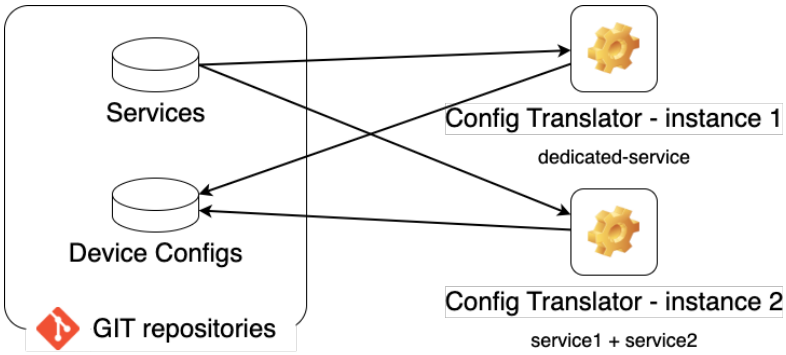

Figure 4.2: Config Translator Scalability

The following repository structure can be used to run two instances of the controller in parallel, one with the responsibility of two services, and the other dedicated to a specific service (for example, in case of some services with high traffic and others with lower traffic).

```
repository
├── data
│   └── …
└── translators
    ├── dedicated-service
    │   └── …
    └── grouped-services
        ├── service1
        │   └── …
        └── service2
            └── …
```

4.4.2 SDCIO Scalability

Scalability issues can arise from three main aspects of the *SDCIO* module:

- the interaction with physical devices can become a bottleneck, especially in large-scale deployments, due to the high traffic it can generate;
- the calculation of the actual configuration that needs to be pushed to the devices can cause a great CPU load;
- the storage of all the *running configs* and the *intents* can become very large, causing performance issues on the caching mechanism.

As discussed in section 2.4.1, the *SDCIO* architecture is designed to have an internal caching mechanism using *BadgerDB* that can improve the handling of the third issue, but is not yet implemented. The second problem might be addressed with code optimization, or with an horizontal scaling of the module. The interaction with physical devices, even if limited, is in any case a potential bottleneck that needs to be monitored and optimized.

Given that the *Telco operator* is internally split in several zones from a logical point of view, it is possible to deploy multiple Kubernetes clusters, one for each zone, each one with its own *SDCIO* instance and local database. In this way, the traffic is reduced and the overall load on each instance is balanced, which can help to mitigate potential bottlenecks, since there are less devices to manage. This approach can also help to reduce the load on the etcd database, as each instance will only need to interact with its local database, rather than a centralized one.

ArgoCD supports the management of multiple Kubernetes clusters, as already discussed in subsection 2.2.2, but with the limitation that each application must be deployed on a single target. For this reason, a little change in the *Device Configs* repository is needed.

In the *Service* repository, each node is already associated with a zone, so the *Config Translator* can be slightly modified to take into account the zone information when generating the configuration files, and outputting the configurations in different subfolders, based on the zone of the target device. The obtained repository structure, considering `edge01` node in `zone0` and `edge02` in `zone1`, is depicted in the following page.

At this point, *ArgoCD* is configured with a different application for each target zone, with the specification to deploy them in different clusters. Note that the provider might also choose to use *virtual clusters* rather than physical ones, if the underlying infrastructure needs to have a more centralized approach. Using different clusters, however, reduces also the risks associated to a single point of failure.

```
repository
├── applications
│   ├── zone0-infra.yaml
│   ├── zone0-services.yaml
│   ├── zone1-infra.yaml
│   └── zone1-services.yaml
├── configurations
│   ├── zone0
│   │   ├── infra
│   │   │   ├── edge01-discoveryrule.yaml
│   │   │   └── ...
│   │   └── services
│   │       ├── service1-customerA-edge01.yaml
│   │       └── ...
│   └── zone1
│       ├── infra
│       │   ├── edge02-discoveryrule.yaml
│       │   └── ...
│       └── services
│           ├── service1-customerA-edge02.yaml
│           └── ...
└── translator-db
    └── ...
```

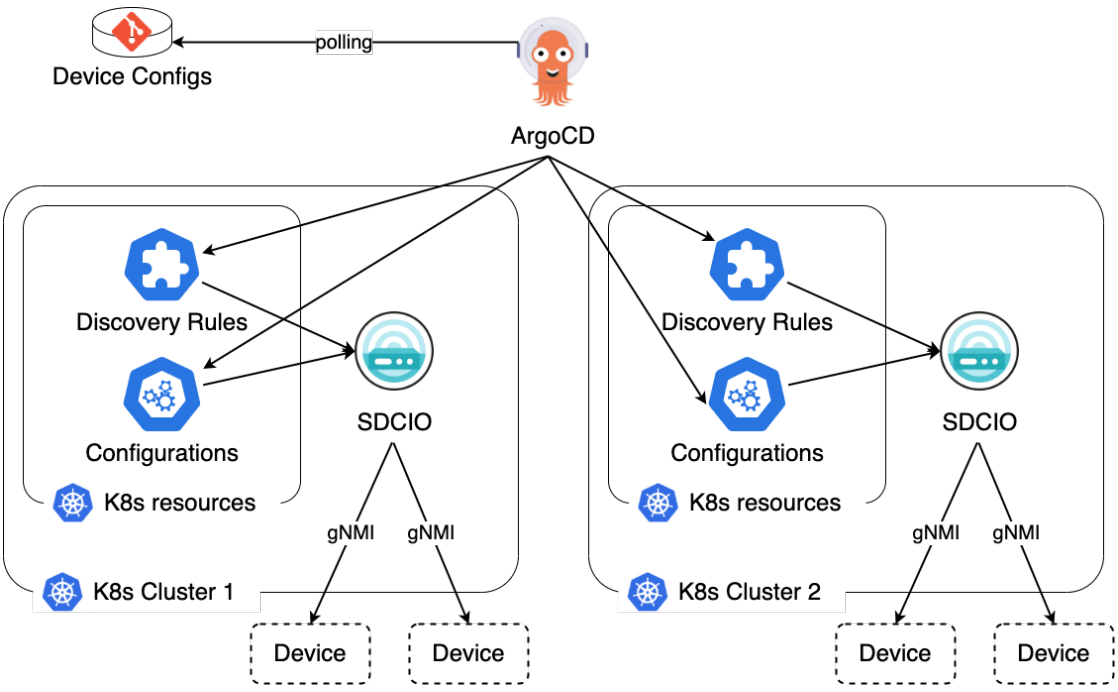

Figure 4.3: SDCIO Scalability

Chapter 5

Implementation

Given the overall architecture of the system, the implementation of the various components is crucial to ensure that the system operates as intended. This chapter will detail the implementation of the main components of the new elements on the system, *Config Translator*, and all the modifications needed in other open source tools, referencing some internal modules of *SDCIO*.

5.1 Changes in SDCIO

As discussed in subsection 2.4.1, *SDCIO* is open source on GitHub and open to contributions from the community. It is pretty new and still evolving, with ongoing efforts to improve its features and capabilities. In particular, there was a great development and testing effort on Nokia devices, and the compatibility with other vendors was assumed with the consideration that all of them would implement the language and communication protocols (YANG and gNMI) at the same way.

The main focus area of this work is on *Cisco* devices, which were not initially discovered using the standard configuration available at that time. Entering in debug mode, and looking what was happening, it was clear that the problem is in the different approach in which *YANG* models are defined from the two vendors. *Cisco* has an extensive use of the namespaces, defining the same key (or path) in different namespaces, making this element a crucial aspect to consider when working with their devices. As an example, `router` path is defined both in the namespace responsible of *static* routing and in the namespace responsible of *dynamic* routing with *OSPF*. Nokia, on the other hand, defines the same key (or path) in a single namespace, which simplifies the model and reduces potential conflicts. The main point of investigation is then the handling of these namespaces in all the elements of the chain. A first manipulation of

the schemas themselves is needed to ensure that they are correctly read by the underlying GoYang library [6], used by the modules to parse YANG models, as discussed in subsection 2.3.1.

The *SDCIO* architecture, presented in section 2.4.1, is composed of three main components: the *Schema Server*, the *Config Server*, and the *Data Server*, each one maintained in a different GIT repository on GitHub, containerized in a separate Docker container, and then unified only at deploy time in a single k8s pod. The *Schema Server* has not been changed as it was already compatible with the management of multiple namespaces, while both the *Config Server* and the *Data Server* required modifications to handle the different namespace structures used by *Cisco* and *Nokia*. The specific changes made to each component are detailed in the following subsections.

5.1.1 Config Server

The *Config Server* uses YANG models and gNMI in the *discovery phase* to identify the basic information about the device configuration and capabilities. This information is then used to manage the device by the *Data Server*.

The paths are already correctly read from the *Discovery Vendor Profile*, and a compatible gNMI request is sent to the device, comprehensive of all the necessary parameters (namespace and path). The node replies with the requested configuration data, including the relevant namespace information directly in the path in the form `namespace:path`. This is not compatible with *SDCIO*, which expects to have only the path without the namespace, and then is not parsed correctly, causing an exception and the termination of the discovery process.

A simple but effective solution to this problem is to modify the *Config Server* to strip the namespace information from the paths before processing them. This way, the paths will be in the expected format, and the discovery process can continue without issues. This is not a problem even in the case of the same path defined in multiple namespaces, since the discovery process already knows which namespace it refers to, and so this is presented as the solution to the community with a *Pull Request*[1] [23].

This feature was appreciated by the community, but a more comprehensive solution was proposed by the community, which involved a deeper integration of namespace handling [24]. However, the management of these namespaces in the *Config Server* is finally in the public codebase.

[1] A *Pull Request* is a proposed change to the codebase that is submitted for review and discussion on the GitHub platform.

5.1.2 Data Server

The *Data Server* is the principal module that interacts with the devices, handling the data retrieval and manipulation tasks. It communicates with the devices using gNMI and is responsible for applying the configuration changes required. For doing this, it uses two principal kind of requests:

- **gNMI Get Requests** are used to obtain the running configuration on the devices, and is already managing the namespaces correctly;
- **gNMI Set Requests** are used to apply the configuration changes on the devices, and is not managing the namespaces correctly. Each *Set Request* has internally multiple **Updates** and **Deletes** that are sent in a single request to the device.

The current implementation of the *gNMI Set Requests* did not account for resources defined in different namespaces. This can cause configuration failures when attempting to update elements, since the paths were ambiguous.

To address this issue, a solution was developed introducing two critical modifications to the system:

- **Namespace-aware request separation**: Updates and deletes are now segregated into different entries based on their respective namespaces, as *Cisco* devices cannot process modifications to resources in multiple namespaces within a single entry; entries are then packaged into a single Set Request, assuring that only one interaction with the device occurs.
- **Path prefixing with namespaces**: Each path in the request is now properly prefixed with its corresponding namespace (in the form `namespace:path`).

The implementation uses an in-memory map to associate paths with their correct namespaces. While this solution effectively resolved the immediate issue, it was acknowledged to have certain limitations, primarily that the in-memory map is not persisted across component restarts, potentially causing inconsistencies in case of crashes.

In addition to the namespace handling changes, two other significant issues were identified and addressed:

- **Boolean value parsing**: *Cisco* devices return boolean values as strings (`"true"` or `"false"`) instead of native boolean types. This required implementing additional parsing logic to correctly interpret these string representations as boolean values during configuration operations.
- **Optional fields handling**: *Cisco*'s implementation of the YANG model handles optional fields differently from what was initially expected. Some keys defined as

optional in *Cisco* YANG models were being interpreted as mandatory by *SDCIO*, causing requests to be blocked before they could be forwarded to the device. A validation mechanism was added to properly identify and handle these optional fields.

Despite some limitations, the implemented solution successfully enabled *SDCIO* to properly interact with *Cisco* devices.

The solution has been proposed to the community through a *Pull Request* [25]. At the time of writing, this contribution is still under discussion and review by the community and has not yet been integrated into the main codebase. Nevertheless, it represents an important step forward in enhancing *SDCIO*'s multi-vendor compatibility, with the understanding that further refinements will be made in future iterations once accepted.

5.1.3 Debugging Structure and Test Deployment

For debugging purposes, the Go codebase of *SDCIO* can be executed locally with minimal configuration. This approach requires setting a few environment variables and utilizes the local kubeconfig file to interact with the Kubernetes cluster where the associated resources, described in section 2.4.1, are deployed.

The main challenges arise when considering a more stable deployment within the *Kind* environment. Several issues need to be addressed. First, regarding network connectivity, the system must be able to interact with the *Cisco XRd* devices running in *containerlab* as detailed in subsection 3.1.2. By default, Docker (within which *Kind* operates) does not have access to external networks. This limitation necessitates the implementation of specific firewall rules to enable proper communication between the containers.

The Docker network architecture creates isolated network namespaces for containers, and by default, implements security measures that prevent traffic flow between different networks. In our setup, the *Kind* cluster operates on one Docker network, while the *Cisco XRd* containers run through *containerlab* on a separate network. To allow communication between these isolated environments, the following firewall rule was implemented:

```
1 iptables -I DOCKER-USER -o \
2   br-$(docker network inspect -f '{{ printf "%.12s" .ID }}' kind)\
3   -j ACCEPT
```

Firewall rule to allow communications between Kind and Containerlab

This rule modifies the Linux *iptables* configuration by inserting a new rule into the

DOCKER-USER chain, which is specifically designed for user-defined rules in Docker environments. The rule targets the bridge interface associated with the *Kind* network (dynamically identified by querying Docker for the network ID) and sets the policy to ACCEPT, effectively allowing outbound traffic from the *Kind* network to reach other networks, including the one where *containerlab* containers reside.

Additionally, due to the modifications made to the *Config Server* and *Data Server* components, the standard public images could no longer be used. This required establishing a local registry server instance, following the methodology outlined in the *Kind* documentation [26].

5.2 Config Translator

The *Config Translator* is designed to address the specific needs provided by the proposed solution. As intended to be a cloud-native module, and for coherence with the other tools used in the presented framework, the GoLang language is used for its implementation. This is the most used language for cloud-native applications, and has libraries that interact effectively within the Kubernetes ecosystem. It can read and write YAML files natively, and is able to use the Jinja2 templating engine for configuration file generation thanks to the *pongo2* package [27].

The principal aim of this module is to translate the configuration files from the *Services repository* format to *SDCIO* resources. For doing that, it has to manage different roles:

- manage upstream and downstream repository using GIT commands;
- actively or passively react on changes in the upstream repository;
- translate configuration files from the *Services repository* format to *SDCIO* resources;
- inject devices information into the configuration files;
- understand the correct output location for each configuration file;
- delete configurations no more needed from the downstream repository.

Moreover, this module is designed to work with big repositories, that contain all the services offered by a *Telco* operator, and then it needs to keep performance in mind. Knowing that it uses GIT repositories, it can exploit GIT features to optimize its operations. For example, it can calculate creations, updates, and deletions only for changed files, avoiding unnecessary processing on unchanged files.

5.2.1 Calculate Only Changed Files

The translation logic is very simple, and detailed in the Flow Chart in Figure 5.1.

As described in subsection 4.2.2, the last applied commit SHA is stored, so that it is simple to determine the diff[2] between the last applied configuration and the current one. If the last commit SHA is not available, the module assumes it is the first time it runs, and then assumes all files are new.

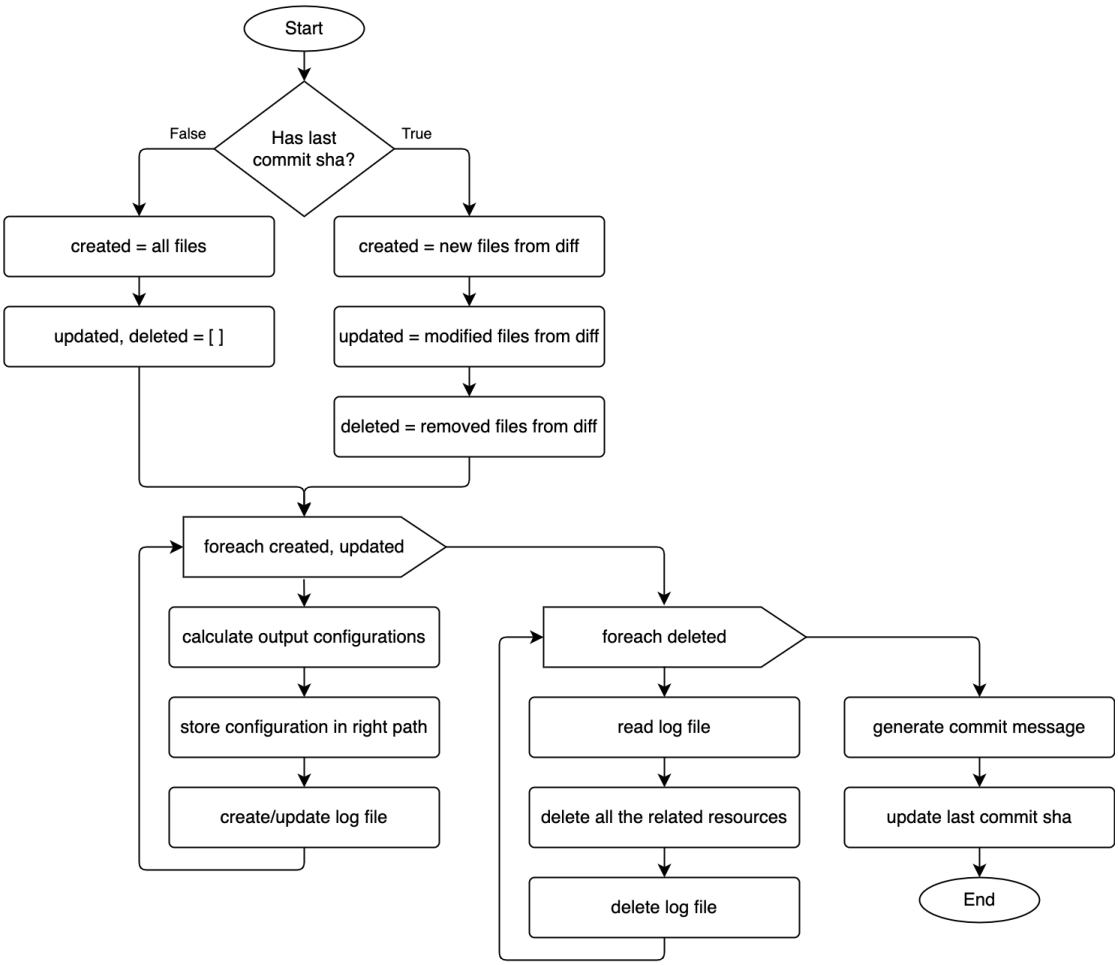

Figure 5.1: Translation Logic Flowchart

Then, operations are different based on the type of change detected. For each new or modified file, the module:

1. reads the input configuration file;

[2]The difference between two versions of a file, showing what has changed.

2. identifies the correct translator to use based on the input configuration file path;
3. runs all the parsers associated with the translator, generating all the output configuration files;
4. stores the output configuration files in the correct location on the downstream repository;
5. creates or updates the log file for that input configuration, inserting the list of output files generated, the commit SHA of the commit being processed, and the current timestamp.

For each deleted file, the module:

1. reads the log file for the specific input configuration[3];
2. deletes all the output files generated for that input configuration;
3. deletes the log file itself.

Finally, a commit message is generated based on the changes made during the translation process, and the last applied commit SHA is updated with the current commit SHA[4].

5.2.2 GIT Repositories Management

When the entire process of the *Config Translator* is triggered, the chain of operations described in the Flow Chart in Figure 5.2 must be executed.

First of all, a *pull* operation is performed on the *Services* repository (*upstream* in the scheme), to ensure that the latest changes are available. Then, the *Config Translator* checks if there are any changes in the repository, and, if so, it pulls also the *Device Configs* repository (*downstream* in the scheme), to ensure the alignment with other active translators. At this point, the entire process of translators described in subsection 5.2.1 can be executed.

Finally, the changes are applied with a commit whose message was already calculated during the translator running, and a push is performed to the *downstream* repository to save the changes and trigger the next component in the framework.

[3] If the log file is not available, it means that the file was never processed, and then there is no further action to take.

[4] Remember that the commit SHA refers to the *upstream* repository, while the current changes are made on the *downstream* one: for this reason is possible to insert it, even if the commit has not yet been performed on the downstream repository

Since there could be other entities working on the same repository, a push operation might fail: in this case, the module automatically tries a *rebase* operation and a new push. Other errors might arise different from the non-aligned one: in that case, the operation must report the error, and a further action could be taken by a human operator or another automated system.

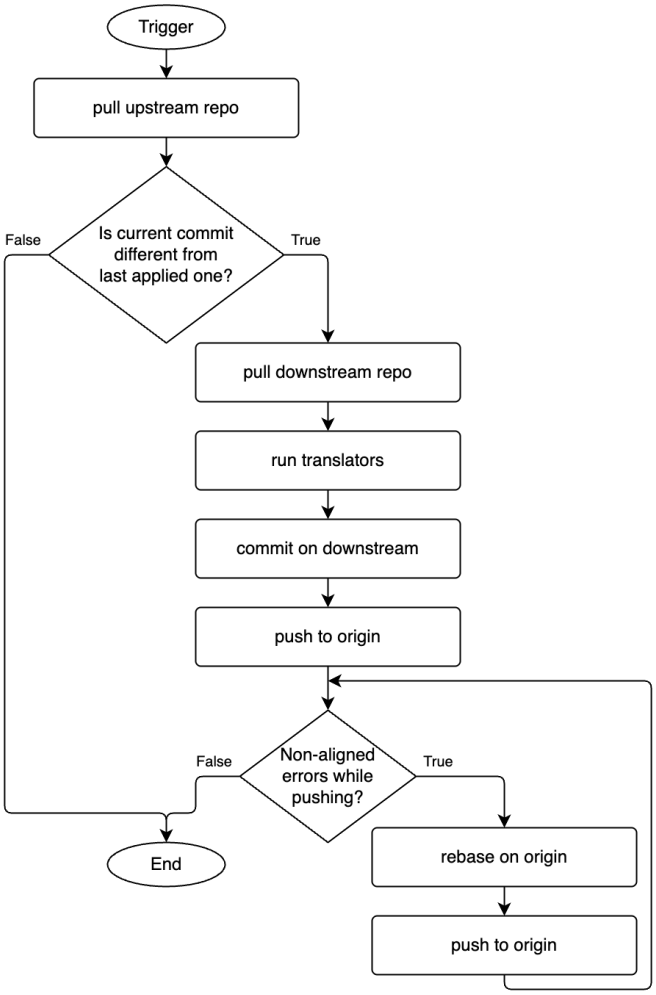

Figure 5.2: GIT Repositories Management Flowchart

The **go-git** package in GoLang [28] is responsible of managing GIT repositories. It supports the execution of all standard operations, excluding merges and rebases. The first implementation used this package to perform all the supported operation, but, given that the rebase ability was a strong requirement, another approach was used for this specific task. The *Config Translator* uses the GIT command line interface to perform the rebase operation, and then uses the *go-git* package for all the other operations. A drawback is that the CLI should be installed separately on the system,

but it should not be a problem since the entire module is intended to be containerized, and so it can be included as a dependency in the container image.

Given that this module shall be able to manage big repositories, a test was conducted in that sense. The module is asked to run translations on a single file changed, but within a big repository ($10,000$ configurations, corresponding to $\sim 30,000$ files), using the *go-git* module for all the supported operations in the first attempt and using the CLI GIT interface also for the *push* operation in the second attempt.

Operation	Attempt	Completion Time (ms)	Std Dev (ms)
Create	1 (*go-git*)	17,881	330
Create	2 (*CLI*)	6,207	105
Update	1 (*go-git*)	17,930	489
Update	2 (*CLI*)	6,080	130
Delete	1 (*go-git*)	17,041	436
Delete	2 (*CLI*)	6,264	362

The completion time of the entire operation is shown in the table above: it clearly shows that the use of the CLI GIT for the *push* operation significantly improved the performance, and for this reason it is the approach used in the final implementation. Same checks were made for all the other GIT operations, but the results showed no such difference between the two approaches, and so the *go-git* package is used for all the other operations, since it is more readable and maintainable.

5.2.3 How To Trigger Translators

We already discussed in section 4.3 that the *Config Translator* can be triggered in *active* or *passive* mode. By the implementation point of view, this has been achieved in three different modalities: **singular invocation**, **HTTP server**, and **polling**. Each of these modes can be chosen at runtime, allowing for greater flexibility in how the reconciler is triggered, since all of them are implemented.

Singular Invocation

This is the simplest way to launch the reconciler, since it works as a command-line tool. The user can simply invoke the tool with the desired flags, and the tool will take care of the rest.

In a CI pipeline, this mode can be easily integrated as a step in the pipeline configuration: the execution is triggered whenever a change is detected in the *Services* repository, causing the creation of the configuration resources and a push to the *Device Configs* repository.

This requires the runners to be enabled on the Git server, and the user to have the

necessary permissions to create and push changes to the repository and to create CI jobs.

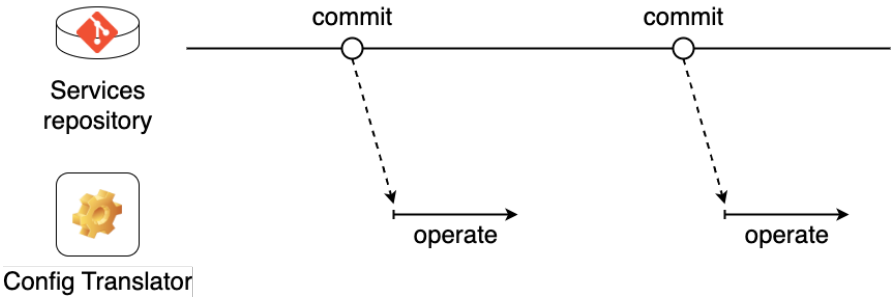

Figure 5.3: Singular Invocation Triggering

HTTP Server

In this mode, the reconciler runs as an HTTP server, listening for incoming requests. This allows for more flexibility, as other components in the system can trigger the reconciler by sending HTTP requests to the server.

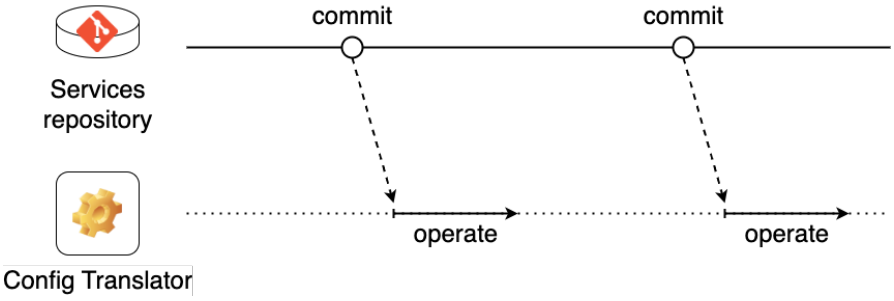

Figure 5.4: HTTP Server Triggering

In a CI pipeline, this mode can be integrated as a webhook that is triggered whenever a commit is pushed to the repository. The drawback is that the server should always be running, and the user must ensure that the server is reachable from the CI pipeline.

This could ensure that any change is immediately processed by the reconciler, but requires the user the permissions to add webhooks to the repository.

Polling

In polling mode, the reconciler periodically checks for changes in the configuration files and triggers the reconciliation process automatically.

This does not require any integration with the CI pipeline, but may introduce some latency in processing changes, depending on the polling interval. The only authorization required is for the reconciler to read the *Configurations* repository and read/write on the *Device Configs* repository.

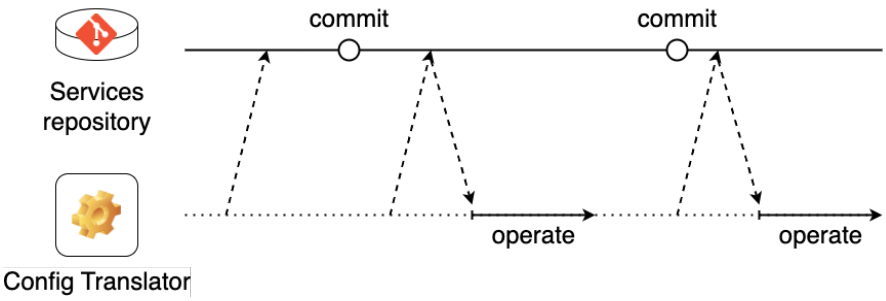

Figure 5.5: Polling Triggering

Comparison

The following table summarizes the key differences between the three triggering modes:

Mode	CI Integration	Latency	Config Permissions
Singular Invocation	Easy	Low	CI settings access
HTTP Server	Webhook	Low	Webhook management
Polling	None	High	-

5.2.4 Packages Architecture

The entire module is structured in a modular way, with each package responsible for a specific task, as visible in Figure 5.6. This allows for better maintainability and scalability, as each component can be developed and tested independently.

- **parser**: this package is responsible for executing the *Jinja2* templates and generating the final configuration files.
- **translator**: this package is responsible for reading the single *Translator* configuration, executing the associated *Parsers*, and exposing them to the upper layer.
- **controller**: this package contains the main logic for orchestrating the translation process, executing all the associated *Translators* and managing the GIT repositories.

- **server**: this package is used only in the *HTTP Server* mode and in the *Polling* mode. It contains the logic for handling incoming requests and triggering the appropriate actions in the *controller* package when a request is received.
- the **main** module starts the right *server* based on the flags provided at runtime, or directly starts the *controller* if *Singular Invocation* mode is chosen.

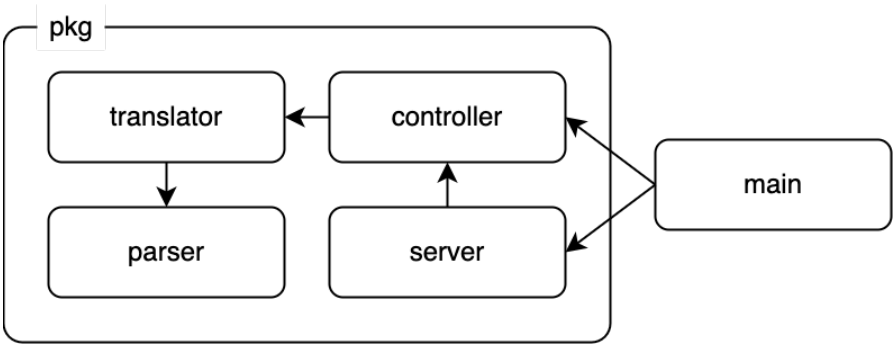

Figure 5.6: Config Translator Packages Architecture

5.2.5 Obtaining Infrastructure Information

The ability to obtain information from the nodes is crucial for the translation process, as discussed in subsection 4.3.3, and is implemented through *Jinja2 filters*.

pongo2 [27] enables the creation of these filters by providing a simple interface. Each function that aims to become a filter must adhere to this specific signature:

```
1 func(in *Value, param *Value) (out *Value, err *Error)
```

Filter Signature

- **in** is the input value, the value written before the pipe (|) in the template, which can be a string, a number, or any other type;
- **param** is an optional parameter that can be passed to the filter, allowing to insert in the template some other information needed to specify some behavior.
- **out** is the output value, the result of applying the filter to the input value;
- **err** is an error value that can be returned if something goes wrong during the filtering process.

The main issue with how filters must be defined is that they need to be declared globally, which prevents registering them differentiated on a per-template basis. They

are then defined in the *controller* package, and able to scan all the configurations of the nodes and obtain the wanted information.

The following filters are defined:

- **nodeLoopback**: retrieves the node loopback IPv4 address given its name;
- **nodeZone**: retrieves the node zone given its name;
- **indent**: changes the indentation level of the output.

5.2.6 Module Runtime Configuration

The module accepts only one argument, that is the working mode required:

- `./config-translator immediate`: this is the *Singular Invocation* mode, where the module runs once and then exits;
- `./config-translator server`: this is the *HTTP Server* mode, where the module runs as a long-lived process, waiting for incoming requests;
- `./config-translator polling`: this is the *Polling* mode, where the module runs as a long-lived process, periodically checking for new requests.

All the module configuration is instead provided through environment variables, as shown in the following table. Please note that all the variable that have a '*' in the default value column in the table need to be explicitly defined by the user and, if not defined, the module would not be able to start.

Table 5.1: Config Translator Environment Variables

Variable Name	Description	Default
GIT_USERNAME	the username to use for authenticating with the GIT server	*
GIT_PASSWORD	the password to use for authenticating with the GIT server	*
IN_REPO_PATH	the local path to the upstream GIT repository	*
TRANSLATORS_SUBPATH	the subpath within the *in* repository where the configuration translators are located	translators
SERVICES_SUBPATH	the subpath within the *in* repository where the services are located	services
NODES_SUBPATH	the subpath within the *in* repository where the nodes configurations are located	nodes
OUT_REPO_PATH	the local path to the downstream GIT repository	*
OUT_SUBPATH	the subpath within the *out* repository where the output files should be saved	configurations
INTERNALDB_SUBPATH	the subpath within the *in* repository where the internal database files are located (configurations logs and last applied commit SHA)	translator-db

Chapter 6

Validation

The aim of the entire framework is to manage services within a network with thousands of nodes and millions of configurations. Simulating such a complex environment is a challenging task, as it would require not only a large number of nodes but also a diverse set of configurations to accurately reflect real-world scenarios.

If we simulate these nodes using lightweight containers, we can achieve a higher level of scalability and flexibility in our testing environment. However, this approach also comes with its own set of limitations: each container would require its resources (some Gigabytes of RAM for each one), and containerized instances do not have the full set of capabilities that physical devices possess.

The validation is then focused on two major aspects:

- the entire framework should work, with all its components, in a small-scale environment, composed of a limited number of nodes and configurations.

- each element should be able to scale independently to assure the overall system performance and reliability.

Several components used in the framework have a large adoption in the world, and we can rely on existing studies and benchmarks to assess their scalability ([20] [21] for *GIT*, [22] for *ArgoCD*). The *SDCIO* module, even if not yet completely developed, is designed with scalability in mind, and the community is actively working on improving its performance and resource management. In any case, with the scalability approach described in subsection 4.4.2 the module can be scaled on several zones, making each instance more resilient and capable of handling increased workloads. The *Config Translator* module, as entirely new, must be evaluated with a greater attention, so that its scalability can be properly assessed.

6.1 Basic Laboratory Deployment

All the validation activities but section 6.4 have been conducted with the environment depicted in Figure 6.1, that contains a mix of physical and virtual devices.

The *Laboratory Machine* is a virtual machine deployed on *vmware* with *Red Hat Enterprise Linux 9.5* Operating System with 16 cores and 32 GB of RAM.

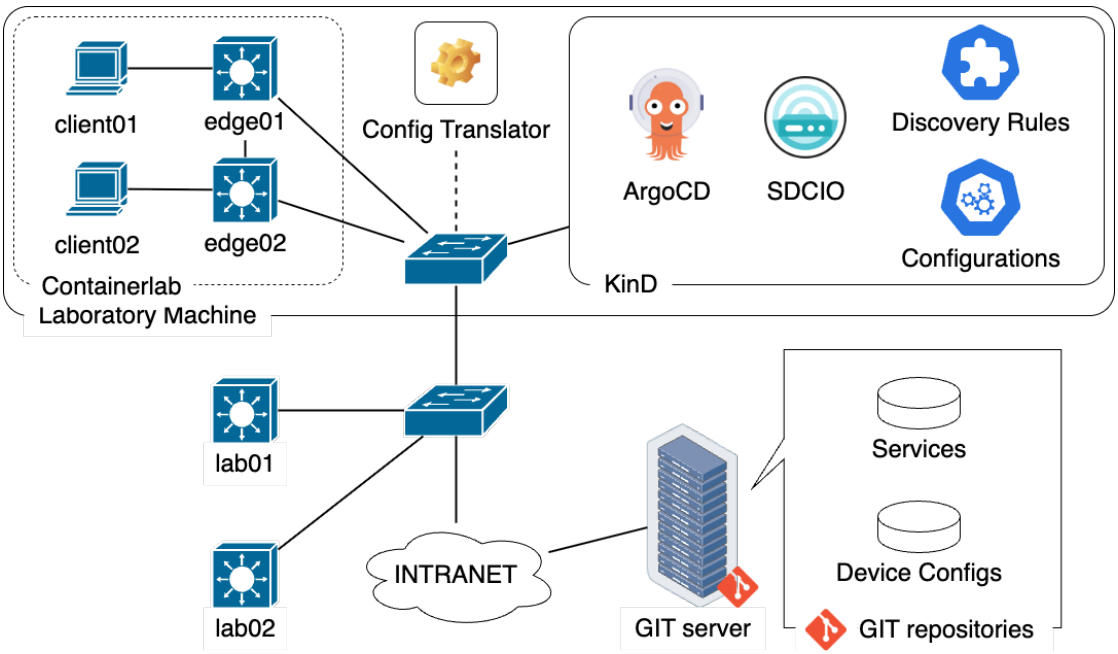

Figure 6.1: Physical Laboratory Infrastructure

- **Kubernetes resources** are available in a *KinD* cluster.
- **ArgoCD** is deployed on the same cluster. It is configured to run an application that points to the `applications` folder of the *Device Configs* repository. This folder then includes two sub-applications: one pointing to the nodes config directory and the other pointing to the services configurations directory, both in the *Device Configs* repository.
- **SDCIO** is deployed on the same cluster, and all the needed k8s resources but *DiscoveryRules* are statically applied on the cluster.
- **Containerlab** is running on the same physical machine with two *Cisco XRd* routers (**edge01** and **edge02**), and two clients (**client01** and **client02**). Each node in *Containerlab* is accessible executing commands via `docker exec` command, while only the routers are directly connected to the host management network

and available for SSH connections. The clients are linked to one of the *edge* routers, and the two *edge* routers are linked together. OSPF and MPLS protocols are enabled.

- **Config translator** is running as a system process directly on the host machine (not containerized).
- two *Cisco ASR9k* physical nodes are available in the same datacenter of the laboratory machines (**lab01** and **lab02**). These nodes are shared with other projects in the company, and have OSPF and MPLS already enabled.
- a **GIT server** is available on the company intranet, but is not physically located on the same site of the laboratory. Both the *Services* and the *Device Configs* repositories are hosted on this server.

6.2 Framework Validation with Different Services

To comprehensively validate the functionality of the entire framework, we conducted tests with different types of services. This approach allows us to verify the complete workflow depicted in Figure 4.1: from the moment a user or an upper-layer tool creates a service configuration, through all the intermediate processing steps, to the actual application on network devices, and finally ensuring that the service works as expected.

We selected two different service configurations for our validation purposes:

- A simpler L2 MPLS Tunnel service, which can be deployed on virtual nodes, described in subsection 3.3.1
- A more complex L2 Tunnel with Bandwidth Control and Monitoring, which requires physical nodes, described in subsection 3.3.2.

This dual approach provides us with a comprehensive validation strategy. The simpler service, deployed on virtual nodes, allows for greater control over the testing environment, enabling us to closely monitor each step of the workflow and manipulate conditions as needed to verify the framework's behavior. Virtual nodes provide greater flexibility for debugging and allow us to validate that all components of the framework interact correctly.

The more complex service, resembling a real-world telecommunications service with bandwidth control and monitoring capabilities, requires physical nodes to properly validate its functionality. This approach helps us ensure that the framework can handle complex service requirements in real-world scenarios.

6.2.1 L2 MPLS Tunnel Service

The logical configuration of this service is explained in subsection 3.3.1. The virtual nodes on *Containerlab* are used to deploy this service.

Abstract Configuration

The configuration shown in subsection 3.3.1 should be deployed on both the edge nodes, and has been mapped to the following *Jinja2* template, where `metadata` and `spec` variables come directly from the *Service* resource, while `device` and `other` variables are mapped respectively to the device in which the configuration is applied and the other end of the service:

```
apiVersion: config.sdcio.dev/v1alpha1
kind: Config
metadata:
  name: service-{{ metadata.name }}-{{ device.deviceName }}
  namespace: default
  labels:
    config.sdcio.dev/targetName: {{ device.deviceName }}
    config.sdcio.dev/targetNamespace: default
    config-translator.telecomitalia.it/targetDevice: {{ device.deviceName }}
    config-translator.telecomitalia.it/kind: service
spec:
  priority: 20
  config:
  - path: /interface-configurations/interface-configuration[interface-name={{
    device.interface }}.{{ spec.vlan }}][active=act]
    value:
      description: "{{ spec.description }}"
      interface-mode-non-physical: l2-transport
      ethernet-service:
        encapsulation:
          outer-tag-type: match-dot1q
          outer-range1-low: "{{ spec.vlan }}"
        rewrite:
          rewrite-type: pop1
  - path: /l2vpn/database/xconnect-groups
    value: {...}
```

Jinja2 Configuration Template

The following is an example of a valid *Service* description, which creates a link between the interfaces *GigabitEthernet0/0/0/2* of both *edge01* and *edge02*, exposing the service on VLAN 200 for the customer, internally using a pseudowire with identifier 120.

```
apiVersion: service.telecomitalia.it/v1alpha1
kind: L2Tunnel
metadata:
  name: serviceA
  namespace: default
spec:
  pseudowire: "120"
  vlan: "200"
  description: "desc1"
  firstEndpoint:
    deviceName: edge01
    interface: GigabitEthernet0/0/0/2
  secondEndpoint:
    deviceName: edge02
    interface: GigabitEthernet0/0/0/2
```

L2Tunnel service example

Testing Procedure

1. Create a new *L2Tunnel* Service resource as shown in the example.
2. Commit and push the resource to the *Services* Git repository.
3. Wait for all the elements in the chain to propagate the configuration until SDCIO applies the configuration to the target devices. Check logs of all the involved modules in the meantime.
4. Log into the virtual routers and verify that the tunnel is correctly configured and running.
5. Log into each client, configure an appropriate IP address on the VLAN interface, and test connectivity between the two clients using `ping`.
6. Delete the *L2Tunnel* Service resource from the Git repository.
7. Commit and push the changes to the Git repository.
8. Wait for the deletion to propagate and verify that the tunnel is no longer active.

All the above procedure is repeated for each configuration deployed, and also with multiple configurations applied at the same time, paying attention to use not overlapping VLANs and pseudowire identifiers:

1. GigEthernet2 @ edge01 - GigEthernet2 @ edge02, VLAN 200, pseudowire 120

2. GigEthernet2 @ edge01 - GigEthernet2 @ edge02, VLAN 201, pseudowire 121

3. GigEthernet2 @ edge01 - GigEthernet3 @ edge01, VLAN 202, pseudowire 122

Testing Results

Table 6.1: Test Cases and Results for L2 MPLS Tunnel Service

Test Case	Config 1	Config 2	Config 3	Result
1	Applied	-	-	✓
2	-	Applied	-	✓
3	-	-	Applied	✓
4	Applied	Applied	-	✓
5	Applied	-	Applied	✓
6	-	Applied	Applied	✓
7	Applied	Applied	Applied	✓

For all test cases, the framework demonstrated correct operation, handling both the creation and deletion of the required configurations. Connectivity between the clients was successfully verified in all application cases, confirming the correct functioning of the L2 MPLS Tunnel service. Furthermore, the framework correctly managed situations where multiple configurations were applied simultaneously, without generating conflicts or errors.

6.2.2 L2 Tunnel with Bandwidth Control and Monitoring Service

This service, described in subsection 3.3.2, is more complex as involves bandwidth control and monitoring capabilities for the L2 tunnel. Differently from the previous one, frames are sent untagged (without VLANs) to the destination device: no more than one service can be active on a single interface at a time.

The application of *Cisco SLA Profiles* and *Y.1731* protocol [29] requires physical devices: we will use *lab01* and *lab02* routers, as depicted in Figure 6.1, that have only one port each available for the validation of this service.

Abstract Configuration

The basic configuration is similar to the previous service, with several other modules enabled and configured.

In particular, the service needs a *policy map* configured on the router that defines the bandwidth limits for the L2 tunnel. Since this policy should be unique on the device and is lightweight, we chose to implement it using a dedicated configuration file with

orphan deletion policy[1]. The resulting configuration has then four configuration files (two for each end of the tunnel).

The following is an example of a valid *Service* description, which creates a link between the interfaces *GigabitEthernet0/0/0/1* of *lab01* and the interface *GigabitEthernet0/0/0/22* of *lab02*, with a bandwidth limit of 100 Mbps, internally using a pseudowire with identifier 150.

```
apiVersion: service.telecomitalia.it/v1alpha1
kind: L2Link
metadata:
  name: lablink
  namespace: default
spec:
  pseudowire: "150"
  description: "L2 Link between two machines in lab"
  cos: "7"
  bandwidth: "100" # Mbps
  firstEndpoint:
    deviceName: lab01
    interface: GigabitEthernet0/0/0/1
  secondEndpoint:
    deviceName: lab02
    interface: GigabitEthernet0/0/0/22
```

L2Link service example

Testing Procedure

1. Create a new *L2Link* Service resource as shown in the example.
2. Commit and push the resource to the *Services* Git repository.
3. Wait for all the elements in the chain to propagate the configuration until SDCIO applies the configuration to the target devices. Check logs of all the involved modules in the meantime.
4. Log into the routers and verify that the link is correctly configured and running.
5. Verify that the monitoring process and the bandwidth control are up and running.
6. Delete the *L2Link* Service resource from the Git repository.
7. Commit and push the changes to the Git repository.
8. Wait for the deletion to propagate and verify that the link is no longer active.

[1] see section 2.4.1

Testing Results

The testing procedure was successfully completed, and no issues were found. The statistics are collected by the routers, and the monitoring process is working as expected, as verified by the following commands executed on the target router:

```
> show ethernet sla statistics history brief profile DMM_7 interface
    GigabitEthernet0/0/0/1.0
Wed Jul 30 16:50:20.475 CEST
Source: Interface GigabitEthernet0/0/0/1.0, Domain GEAMEFPTR-PDI
Destination: Target MEP-ID 3401
================================================================================
Profile 'DMM_7', packet type 'cfm-delay-measurement'
Scheduled to run every 1min first at 00:00:09 UTC for 1min

Round Trip Delay
~~~~~~~~~~~~~~~~
5 probes per bucket
No breached stateful thresholds.

                            Results (ms)                      Info
Bucket started     ----------------------------------  --------------------
(CEST)                  Min      Max     Mean       SD  Suspect Result Count
-----------------  -------- -------- -------- --------  ------- ------------
16:45 30 Jul 2025     0.071    0.078    0.072    0.001                   300
> show ethernet sla statistics history brief profile SLM_7 interface
    GigabitEthernet0/0/0/1.0
Wed Jul 30 16:48:45.824 CEST
Source: Interface GigabitEthernet0/0/0/1.0, Domain GEAMEFPTR-PDI
Destination: Target MEP-ID 3401
================================================================================
Profile 'SLM_7', packet type 'cfm-synthetic-loss-measurement'
Scheduled to run every 1min first at 00:00:09 UTC for 1min
Frame Loss Ratio calculated every 1min

One-way Frame Loss (Source->Dest)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
5 probes per bucket
No breached stateful thresholds.

                        Results (%)                    Info
Bucket started     ------------------------  ------------------------------
(CEST)                 Min     Max Overall  Suspect        Sent       Lost
-----------------  ------- ------- -------  ------- ----------- ----------
16:40 30 Jul 2025    0.000   0.000   0.000                  300          0
```

Commands to verify the monitoring and bandwidth control

6.3 Config Translator Scalability

The *Config Translator* module is designed to run multiple instances in parallel, as described in subsection 4.4.1. This section explores the performance of each instance in different conditions, including varying the number of configuration files processed simultaneously and the total size of the upstream and downstream repositories.

These performances depend also on external factors, such as the available system resources and the network conditions, and in particular the load of the used GIT server. Several measurements for each condition have been taken, with a minimum of ten, and the average values are reported.

The used translator requires to create two configurations for each service description applied, and the numbers in the following sections refer to the number of services processed. So, the number of files written in the downstream repository is $\sim 3\times$ *number of services* (two configurations and the log file).

6.3.1 Performance on Clean Repository

Starting from repository in which no service is deployed, with the only exception of the nodes configurations, the performance is measured in terms of the time taken to process the configuration files. The time is measured on creation, update and delete request, with several number of configurations applied at the same time.

Table 6.2: Config Translator performance with different numbers of configurations

# of Configurations	Create (ms)	Update (ms)	Delete (ms)
1	1937 ± 179	1791 ± 144	2014 ± 183
5	2257 ± 372	2370 ± 1245	2092 ± 116
10	2418 ± 260	2141 ± 157	2006 ± 217
50	2062 ± 141	1939 ± 115	2398 ± 704
100	3148 ± 1678	2914 ± 1235	3338 ± 781
500	5658 ± 1533	5148 ± 1049	4487 ± 1087
1000	7544 ± 1569	6845 ± 1196	6982 ± 2012

The Table 6.2 shows the obtained results for each operation (create, update, delete) with different numbers of configurations, including both the average time in milliseconds and the standard deviation. The results are also graphically represented in Figure 6.2.

The linear relation between the number of configurations and the time taken for each operation is evident from the results. There are not significant differences in the performance of the three types of operations, with all of them showing a similar trend as the number of configurations increases. There is a constant initial delay of $\sim 2s$ due to the fetch and push of the remote repositories.

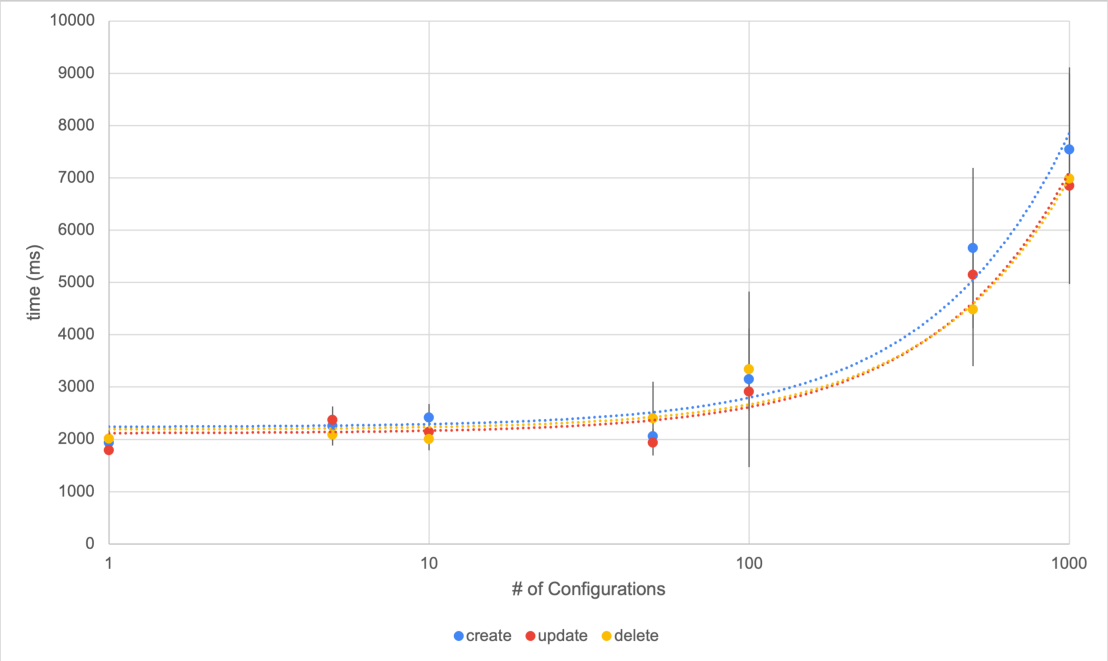

Figure 6.2: Config Translator Performance on Clean Repository

6.3.2 Performance on Loaded Repository

The module will work with huge repositories. Understanding how it works when several configurations are already present is crucial for ensuring its scalability and performance.

Table 6.3: Config Translator performance with different numbers of configurations on a loaded repository

# of Configurations	Create (ms)	Update (ms)	Delete (ms)
1	6208 ± 105	6080 ± 131	6264 ± 362
5	6620 ± 338	6475 ± 109	6466 ± 210
10	6984 ± 348	7013 ± 1009	6892 ± 344
50	6479 ± 174	6449 ± 186	8764 ± 360
100	6779 ± 253	6928 ± 660	11259 ± 478
500	8375 ± 531	7953 ± 590	30803 ± 359
1000	10083 ± 437	10133 ± 357	56991 ± 443

The same tests of subsection 6.3.1 are then repeated with a repository in which 10.000 services are already deployed (so, ~30.000 configuration files are already present in the repository). The results are shown in Table 6.3 and graphically represented in Figure 6.3. Since there is a significant difference in the performance between create/update and delete operations, the Figure 6.4 plots the performance of

create/update operations only.

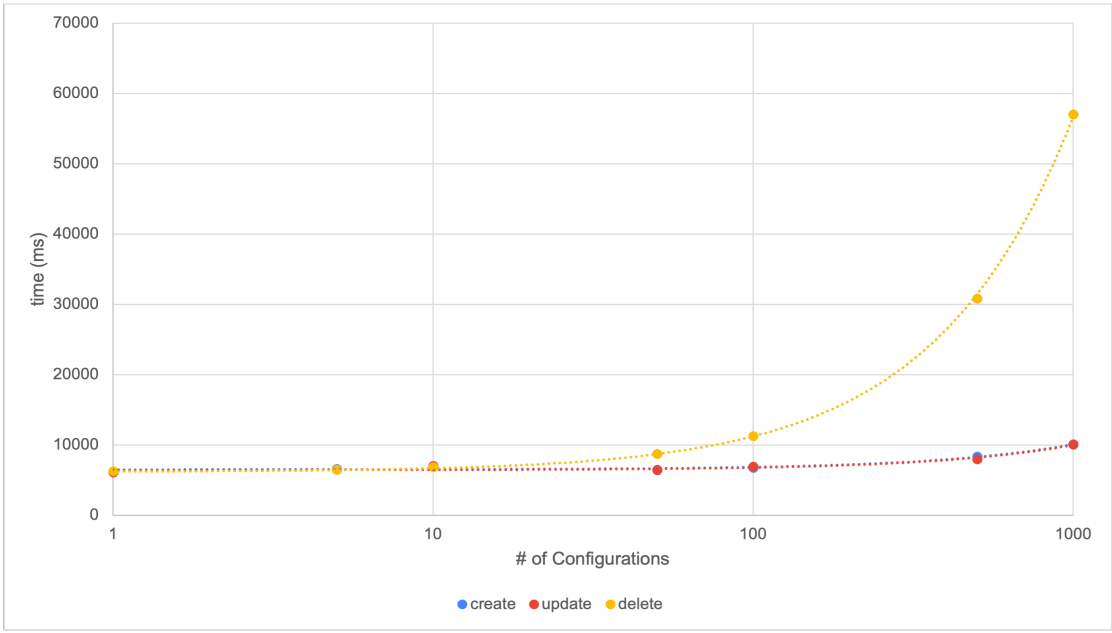

Figure 6.3: Config Translator Performance on Loaded Repository

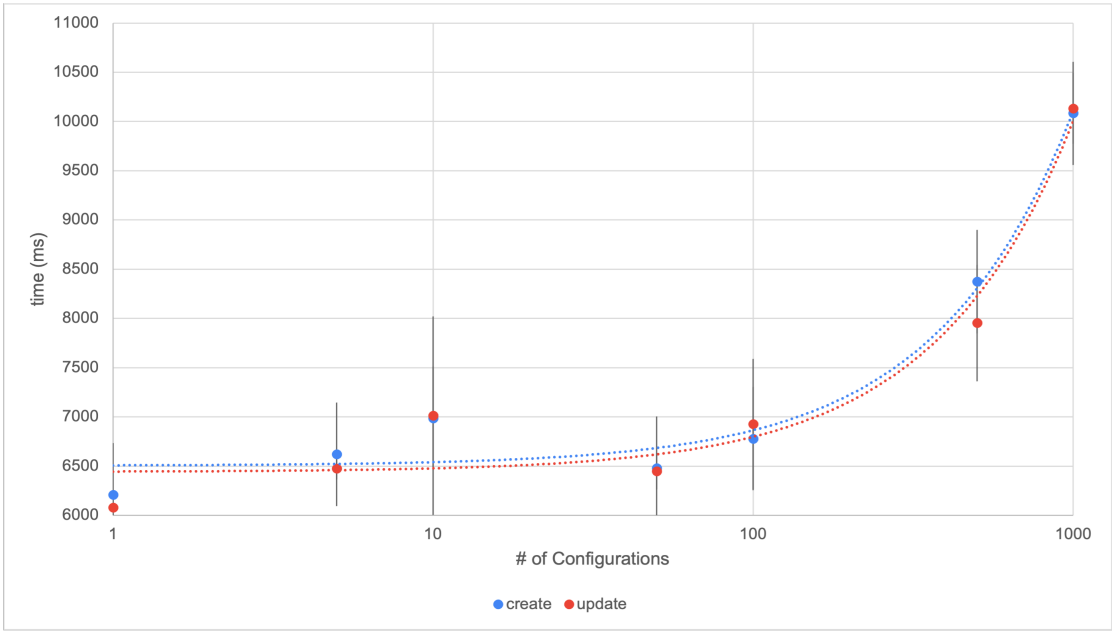

Figure 6.4: Config Translator Performance on Loaded Repository (Create/Update Only)

Differently from the previous scenario, the performance of create and update operations is significantly better than that of delete operations. This should not be a problem in real-world scenarios, as deletions are expected to be less frequent than create and update operations.

Figure 6.5 shows the comparison between the performances obtained in subsection 6.3.1 and in this section on the *create* operation only. The difference in timing between applying a single configuration and 100 configurations is similar in both scenarios, while it increases more rapidly when applying 1000 configurations in the case of clean repository, demonstrating the efficiency when applied to a loaded one.

However, the constant delay related to the fetch and push of the repository is much more significant in this scenario, being $\sim 6s$ instead of $\sim 2s$.

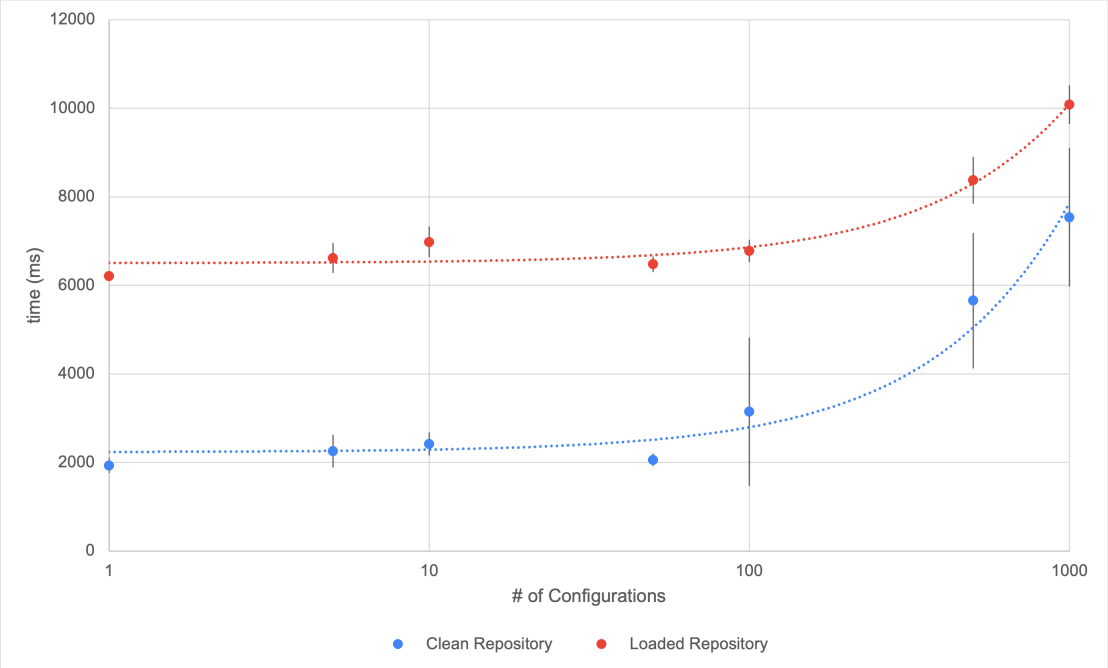

Figure 6.5: Comparison of Config Translator Performance on Clean and Loaded Repositories

6.3.3 Performance with Incremental Repository Load

The difference in performance between the two scenarios (clean vs loaded repository) highlights the impact of existing configurations on the processing time. As the number of configurations increases, the time taken for create, update, and delete operations also rises significantly, especially in a loaded repository.

To better understand this scenario, we can analyze the performance metrics in

more detail with an incremental approach. The Table 6.4 shows how the processing time changes as the repository size grows incrementally, measuring both the time to apply a single configuration and the time to apply 1000 configurations.

The Figure 6.6 graphically represents these results and the time delta between applying a single configuration and 1000 configurations. This time is almost constant across different repository sizes, confirming that the overhead introduced by additional configurations does not vary significantly with the number of existing elements.

The time needed to apply a single configuration increases linearly with the number of existing elements. The increase is $< 5s/10.000configs$, which is acceptable for the intended use case.

Table 6.4: Config Translator performance with increasing repository size

Existing Elements	1 Config (ms)	1000 Configs (ms)	Delta (ms)
0	2820 ± 328	5224 ± 329	2404 ± 657
1000	2667 ± 92	5934 ± 625	3267 ± 717
2000	3307 ± 105	5861 ± 366	2554 ± 472
3000	3665 ± 197	6817 ± 1247	3152 ± 1445
4000	4165 ± 114	7016 ± 160	2851 ± 274
5000	4695 ± 300	7895 ± 658	3200 ± 958
6000	4971 ± 202	8732 ± 2292	3761 ± 2494
7000	5558 ± 153	8631 ± 418	3073 ± 572
8000	6018 ± 232	9020 ± 308	3003 ± 540
9000	6345 ± 163	9324 ± 161	2979 ± 324
10000	7157 ± 601	9928 ± 126	2771 ± 727
11000	7368 ± 181	10581 ± 422	3213 ± 604

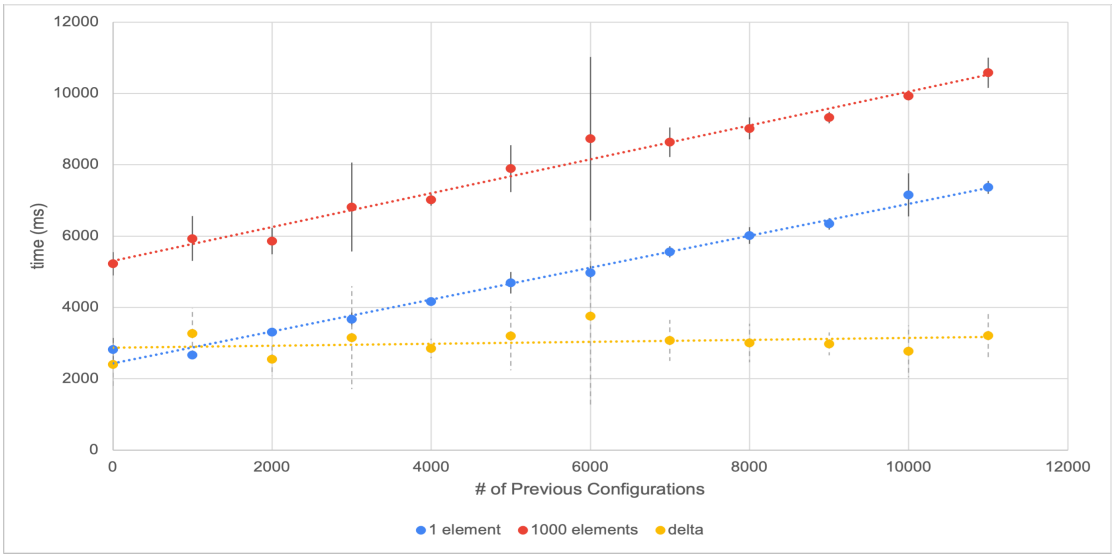

Figure 6.6: Config Translator Performance with Increasing Repository Size

6.4 Scalability of SDCIO over Multiple Zones

The last step is to validate the scalability of the *SDCIO* module, deploying a different *SDCIO* instance for each zone. We are not interested in performance, but rather in the ability to manage multiple instances effectively. For this reason, the entire infrastructure is virtualized.

6.4.1 Modified Laboratory Deployment

The topology depicted in Figure 6.1 is incremented with some additional components to support the multi-zone architecture, and Figure 6.7 shows the resulting infrastructure.

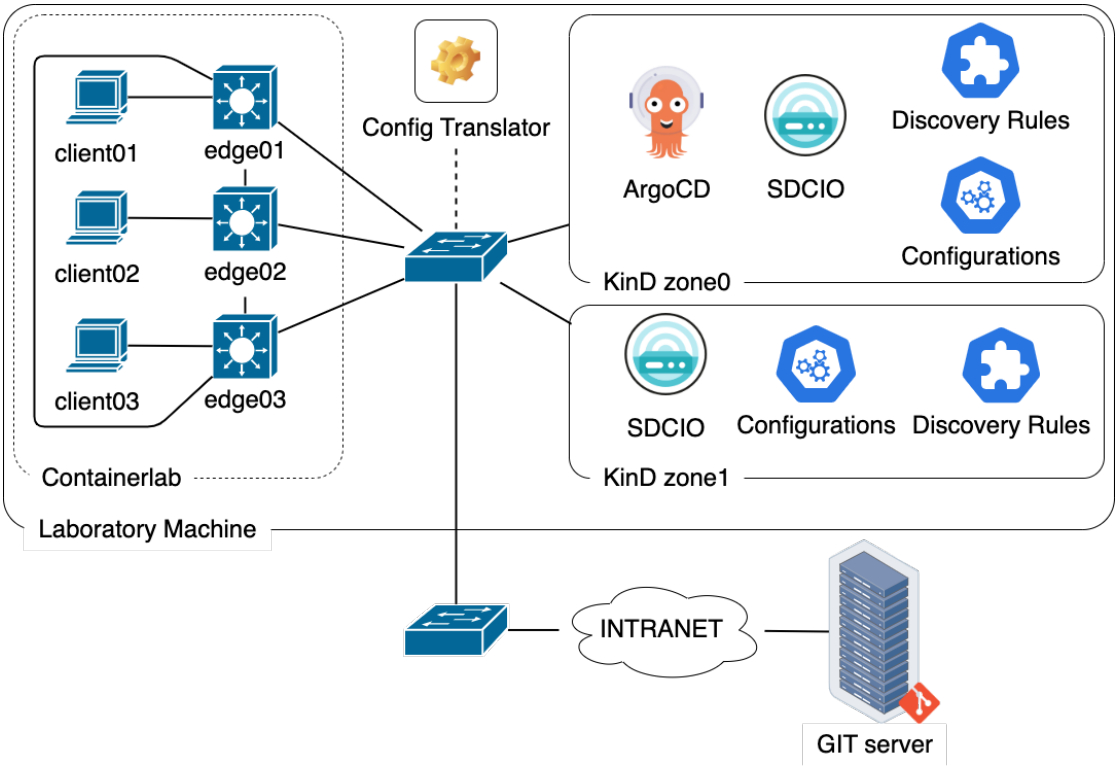

Figure 6.7: Physical Laboratory Infrastructure for Multiple Zones

- Two distinct Kubernetes clusters are deployed using KinD, each representing the controller for a different zone, and an instance of *SDCIO* is deployed in each one.
- Another edge *Cisco XRd* router (**edge03**) is added to the *Containerlab* topology, connected to **edge01** and **edge02**. *edge01* and *edge02* belong to the *zone0*,

while *edge03* belongs to the *zone1*. The nodes are directly connected one another.

- A single instance of **ArgoCD**, deployed on the cluster of *zone0*, controls both the clusters.

- The rest of the topology remains unchanged.

6.4.2 Repository Structure and ArgoCD Applications

Each zone has two ArgoCD Applications: one for the nodes configuration, and one for the services configuration. There are then four applications defined in the appropriate folder into the *Device Configs* repository, and the configurations are inserted by *Config Translator* in different folders based on the zone the node belongs to.

```
repository
├── applications
│   ├── zone0-infra.yaml
│   ├── zone0-services.yaml
│   ├── zone1-infra.yaml
│   └── zone1-services.yaml
├── configurations
│   ├── zone0
│   │   ├── infra
│   │   │   ├── edge01-discoveryrule.yaml
│   │   │   └── edge02-discoveryrule.yaml
│   │   └── services
│   │       └── service-edge01.yaml
│   └── zone1
│       ├── infra
│       │   └── edge03-discoveryrule.yaml
│       └── services
│           └── service-edge03.yaml
└── translator-db
    └── …
```

6.4.3 L2 MPLS Tunnel Service over Multiple Zones

The same service configuration and testing procedure described in subsection 6.2.1 is applied to the multi-zone scenario. The tunnel should connect *edge01*, which is located in *zone0*, to *edge03*, which is located in *zone1*.

Results

The service is correctly deployed: the `service-edge01.yaml` is created in the *zone0/services* folder and applied by *ArgoCD* on the cluster of *zone0*, while the `service-edge03.yaml` is created in the *zone1/services* folder and applied by *ArgoCD* on the cluster of *zone1*.

The configurations are then applied by the two instances of *SDCIO* on the edge nodes, and the tunnel is finally established, providing connectivity between *client01* and *client03*.

Chapter 7

Conclusion

This thesis explored the already present and growing framework for centralized network management and monitoring. By adopting a *cloud native* approach for problem solving, we were able to create a framework that combines the best of the tools widely used in cloud environments with some under-development specific tools, and completing the missing pieces with our own implementations.

The result is a modular framework that has the following characteristics:

- **Stateless**: each component has no state inside it and all the static information are stored in GIT repositories. The only status is the one obtained by the devices and stored into the *SDCIO* module, but it can be downloaded again at any time from the devices.
- **Modular**: each component is designed as a microservice independent from the others, allowing an independent lifecycle and easy replacement if needed. The interfaces between modules are standard in the market and so easily replicable from other tools.
- **Scalable**: each component of the framework can be scaled horizontally in different ways, as demonstrated by the validation performed on the modules.

This work demonstrated the feasibility of the application of *cloud native* principles to the network management and monitoring field, and that it offers concrete advantages.

The new approach, based on a distributed architecture, permits to go beyond the limitations of the single database architecture and its lock problems. The framework will acquire a more robust stability from this point of view with the implementation of a distributed database in the *SDCIO* module, already planned for future works by the community.

Thanks to the most modern technologies and protocols, the module is able to maintain the local device status aligned to the real one, being notified in real time of any change happening on the devices. This does not solve completely the problem of technicians working directly on the devices, but it can guarantee that the configurations applied by the central module takes care of the current device status and not of a stale one.

Even if not yet stable enough for a production environment, and still open to several improvements as presented in chapter 8, the framework demonstrates that walking towards a *cloud native* approach is a viable solution for the network management and monitoring field, and that it can bring several advantages to *Telco* operators.

Chapter 8

Future Work

The status of the presented framework is that of a prototype, and as such there are many aspects that can be enhanced or added to improve it. In this chapter, there is the starting point for some of these future works that arose during the ideation and implementation of the framework.

8.1 Closing the Loop with Service Monitoring

The current framework enables the deployment of services starting from an abstract description and pushing them down to the physical devices. However, there is no feedback that enables the user to monitor the status of the intended service, neither if the service has been correctly deployed, nor if it is still running as intended.

In the overall architecture of SDN in *TIM*, this monitoring is partially covered by another stack in charge of monitoring the network, but there is no correlation between the abstract service description and the actual status.

This is particularly relevant in a production environment to provide a better experience to the final users, because it allows the *Telco* operator to have a clear view of the status of the services and possibly react in case of failures in a more accurate and faster way, either automatically or with human intervention.

The abstract service description in the *Services* repository has a Kubernetes-like structure, with *YAML* manifest files, but that includes only *spec* fields, without any *status*. The loop might be closed adding this auto populated field in the original repository, adding there all the needed information about the deployed service.

New challenges arise in this scenario, as the *Services* repository is currently intended to be a static repository from the framework point of view, since it is only modified by the upper layer. Adding such a dynamic behavior might require a study about the synchronization and conflict management between the user and the automated

system.

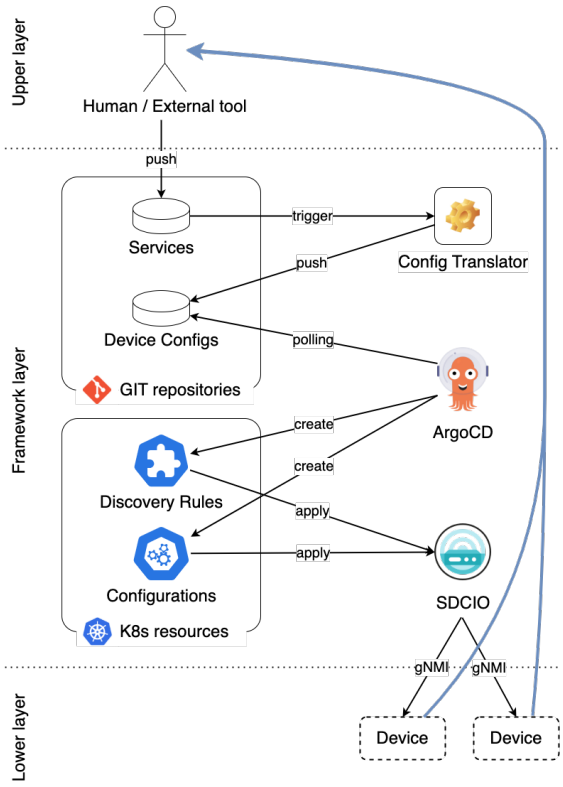

Figure 8.1: Closing the Loop with Service Monitoring

8.2 Awareness of Available Resources on Nodes

The current framework enables the user to inject in the configurations some specific information about the nodes where to deploy the services, such as the loopback IP address, starting from its name. The data that are not always the same in the node, such as the pseudowire ID to use, must be provided in the abstract service description.

While some of these data depends on the willing of the final customer, others are purely technical and there is no need to ask the upper layer to take care of them. The framework might be improved by adding a specific module able to understand the available resources on the wanted infrastructure, and automatically provide them to the *Config Translator* that can then be able to inject more intelligence in the final configurations.

This is exactly what *KUID* does in the *Kubenet* framework, as presented in subsection 2.4.2. There are also other tools that can be used to achieve this goal, and a further

study might be done to understand which one is the most suitable to integrate in the current framework as an external module, or if it is better to develop a custom solution based on the specific needs of the operator.

8.3 Adaptive Services

The Services, as they are currently defined, are always deployed in the same way, only with different parameters. If we consider the example of the *L2 MPLS Tunnel* described in subsection 3.3.1, in consideration with the corner case illustrated in Figure 3.7, we can see that the two edge nodes are physically the same device.

In this case, the current framework will deploy the full service, with the pseudowire and the VLAN interfaces, even if they are not needed. A more intelligent framework might be able to understand this and adapt the service, without the creation of a pseudowire, but simply connecting the two VLAN interfaces directly. This can help in reducing the complexity of the final configuration, with a reduction of the unnecessary overhead and resources consumption.

Jinja2 templating language can be exploited to achieve this goal, adding some conditional statements in the templates, and providing the needed information to the *Config Translator* to understand if a configuration should be applied or not. In the example presented before, we can then deploy a single configuration in the node that is both the edge nodes, with the settings of both the endings of the tunnel, instead of two separate configuration files.

8.4 Performance Improvements

As presented in section 6.3, the performances of the *Config Translator* module are influenced by the size of the two involved repositories. While the timing for deploying a service is not critical even in a production environment, being able to reduce the time needed to process the configurations could help in reducing the load on the system, and to reduce the possibility of errors and inconsistencies.

GIT is used in heavy production environments, and is the subject of different research works to improve its performances. Diving into these works might help in understanding if there are some improvements that can be applied to the current framework, or if there are some alternative solutions that can be used to achieve the same goal with better performances.

8.5 Topology Visualization

Currently, the framework is able to deploy services starting from an abstract description, creating the needed configurations for each involved node. However, there is no way for the user or the upper layer to visualize *how* the single service has been deployed, and *where* each component is located in the network. This could be helpful in some scenarios of troubleshooting.

The architecture of the framework is designed to be modular and extensible, making it possible to add new modules to enhance its functionalities. A new module might be added to provide a visualization of the network topology, combining the information about nodes obtained from the *discovery phase* of SDCIO, and the information about all the deployed services created by the *Config Translator* module.

Acronyms

API

Application Programming Interface, a set of rules and tools for building software applications

CIDR

Classless Inter-Domain Routing, a method for allocating IP addresses and IP routing that replaces the older system based on classes A, B, and C

CLI

Command Line Interface, a text-based interface for interacting with software applications

CR

Custom Resource, an extension of the Kubernetes API that allows users to define their own resource types (see section 2.1.1)

CRD

Custom Resource Definition, a way to extend Kubernetes capabilities by defining new resource types that can be managed by the Kubernetes API (see section 2.1.1)

GIT

A distributed version control system for tracking changes in source code during software development

gRPC

Google Remote Procedure Call, an open source remote procedure call system initially developed by Google, using HTTP/2 for transport and Protocol Buffers as the interface description language

IPv4

Internet Protocol version 4, the fourth version of the Internet Protocol (IP) used to identify devices on a network through an addressing system

ISP

Internet Service Provider, a company that provides access to the Internet

JSON

JavaScript Object Notation, a lightweight data interchange format

k8s

Kubernetes

MPLS

Multiprotocol Label Switching, a method for speeding up and shaping network traffic flows

OS

Operating System, software that manages computer hardware and software resources and provides common services for computer programs

OSPF

Open Shortest Path First, a routing protocol for Internet Protocol (IP) networks that uses a link state routing algorithm and falls into the group of interior gateway protocols (IGP)

REST

Representational State Transfer, an architectural style for designing networked applications

RFC

Request for Comments, a type of publication from the leading technical development and standards-setting bodies for the internet

SDN

Software-Defined Networking, an approach to computer networking that allows network administrators to manage network services through abstraction of lower-level functionality

SHA

Secure Hash Algorithm, a family of cryptographic hash functions used to ensure data integrity, used in GIT for uniquely identifying commits

SLA

Service Level Agreement, a commitment between a service provider and a client that defines the level of service expected from the service provider

SSH

Secure Shell, a cryptographic network protocol for operating network services securely over an unsecured network

TLS

Transport Layer Security, a cryptographic protocol designed to provide communications security over a computer network

VLAN

Virtual Local Area Network, a logical subnetwork that groups a collection of devices from multiple networks into a single broadcast domain

XML

Extensible Markup Language, a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable

YAML

YAML Ain't Markup Language, or Yet Another Markup Language, a human-readable data serialization standard

Bibliography

[1] Paolo Fasano. "Software Driven Network Operations." Politecnico di Torino. 2024. url: https://www.polito.it/ateneo/comunicazione-e-ufficio-stampa/appuntamenti/news?idn=22154 (cit. on p. 2).

[2] *The Kubernetes Project*. url: https://kubernetes.io/ (cit. on p. 5).

[3] *KinD documentation*. url: https://kind.sigs.k8s.io/ (cit. on p. 10).

[4] *ArgoCD documentation*. url: https://argo-cd.readthedocs.io/en/stable/ (cit. on p. 10).

[5] Martin Björklund. *The YANG 1.1 Data Modeling Language*. RFC 7950. Aug. 2016. doi: 10.17487/RFC7950. url: https://www.rfc-editor.org/info/rfc7950 (cit. on p. 13).

[6] *GoYang package description*. url: https://pkg.go.dev/github.com/openconfig/goyang (cit. on pp. 13, 47).

[7] Rob Enns, Martin Björklund, Andy Bierman, and Jürgen Schönwälder. *Network Configuration Protocol (NETCONF)*. RFC 6241. June 2011. doi: 10.17487/RFC6241. url: https://www.rfc-editor.org/info/rfc6241 (cit. on p. 14).

[8] Rob Shakir, Anees Shaikh, Paul Borman, Marcus Hines, Carl Lebsack, and Chris Morrow. *gRPC Network Management Interface (gNMI)*. Internet-Draft draft-openconfig-rtgwg-gnmi-spec-01. Work in Progress. Internet Engineering Task Force, Mar. 2018. 9 pp. url: https://datatracker.ietf.org/doc/draft-openconfig-rtgwg-gnmi-spec/01/ (cit. on p. 14).

[9] *Cisco Network Services Orchestrator*. url: https://www.cisco.com/c/en/us/products/cloud-systems-management/network-services-orchestrator/index.html (cit. on p. 14).

[10] *OpenDaylight Platform*. url: https://www.opendaylight.org/ (cit. on p. 15).

[11] *Open Network Operating System (ONOS)*. url: https://opennetworking.org/onos/ (cit. on p. 15).

[12] *Kubenet Framework*. url: https://learn.kubenet.dev/ (cit. on p. 15).

[13] *SDCIO description*. url: https://docs.sdcio.dev/ (cit. on p. 16).

[14] *KUID description*. url: https://docs.kuid.dev/ (cit. on p. 20).

[15] *Choreo description*. url: https://choreo-docs.kform.dev/ (cit. on p. 20).

[16] *Pkgserver description*. url: https://docs.pkgserver.dev/ (cit. on p. 22).

[17] *Containerlab documentation*. url: https://containerlab.dev/ (cit. on p. 25).

[18] *Cisco XRd documentation*. url: https://www.cisco.com/c/en/us/support/routers/ios-xrd/series.html (cit. on p. 25).

[19] *Jinja2 Templates documentation*. url: https://jinja.palletsprojects.com/en/stable/templates/ (cit. on p. 38).

[20] Yucheng Low, Rajat Arya, Ajit Banerjee, Ann Huang, Brian Ronan, Hoyt Koepke, Joseph Godlewski, and Zach Nation. "Git Is For Data." In: *CIDR*. 2023. url: https://www.cidrdb.org/cidr2023/papers/p43-low.pdf (cit. on pp. 42, 60).

[21] Taylor Blau. *Scaling monorepo maintenance | The Github Blog*. Apr. 2021. url: https://github.blog/2021-04-29-scaling-monorepo-maintenance/ (cit. on pp. 42, 60).

[22] Ajayi Abiola Samuel, Oriyomi Badmus, Godwin Okechukwu Iheuwa, Lucky Ehizojie, and Shokenu Emmanuel Segun. "Comparative Analysis of GitOps Tools and Frameworks." In: (2025). url: https://www.researchgate.net/profile/Abiola-Ajayi-12/publication/392363693_Comparative_Analysis_of_GitOps_Tools_and_Frameworks/links/683f0df4df0e3f544f5cb812/Comparative-Analysis-of-GitOps-Tools-and-Frameworks.pdf (cit. on pp. 42, 60).

[23] *Proposed Pull Request for SDCIO Config Server*. url: https://github.com/sdcio/config-server/pull/348 (cit. on p. 47).

[24] *Pull Request for SDCIO Config Server with the needed changes*. url: https://github.com/sdcio/config-server/pull/352 (cit. on p. 47).

[25] *Proposed Pull Request for SDCIO Data Server*. url: https://github.com/sdcio/data-server/pull/303 (cit. on p. 49).

[26] *Kind Local Registry documentation*. url: https://kind.sigs.k8s.io/docs/user/local-registry/ (cit. on p. 50).

[27] *Pongo2 package description*. url: https://pkg.go.dev/github.com/flosch/pongo2 (cit. on pp. 50, 57).

[28] *go-git package description*. url: https://pkg.go.dev/github.com/go-git/go-git/v5 (cit. on p. 53).

[29] *Cisco Ethernet CFM, Y.1731 Basic Concepts, Configuration, and Implementation*. url: https://www.cisco.com/c/en/us/support/docs/asynchronous-transfer-mode-atm/operation-administration-maintenance-oam/117457-technote-cfm-00.html (cit. on p. 65).