IMPLEMENTING A MACHINE LEARNING FUNCTION ORCHESTRATOR

A thesis presented for the degree of
Master in Innovation and Research in Informatics (MIRI)

Title: Implementing a Machine Learning Function Orchestrator
Author: Axel Wassington
Advisor: Luis Velasco
Supervisor: Luis Velasco
Date: January 15, 2022

Abstract

Deployment of Machine Learning (ML) applications requires an Orchestrator to create ML pipelines, where ML functions are connected. The project consists of implementing an Orchestrator and demonstrating the deployment and reconfiguration of ML pipelines. The Orchestrator will run an optimization algorithm to assign the ML functions into datacenters and then coordinate with the Virtual Infrastructure Orchestrator (VIO) and the Software Defined Networking (SDN) controller to create the ML functions.

Keywords: Orchestration, Network Function Virtualization, Intent-based Networking, Software Defined Networking, Virtual Infrastructure Orchestrator, Machine Learning, Kubernetes, Optimization, Edge/Fog computing, Network Automation.

Contents

Acronyms

API Application Programming Interface. 12, 21, 23–25

AWS Amazon Web Services. 8

CDPI Control to Data-Plane Interface. 6

CNI Container Network Interface. 12

CORD Central Office Re-architected as a Datacenter. 8

DB Database. 28, 32

IBN Intent-Based Networking. 1, 2

ILP Integer Linear Program. 9, 15, 21, 24, 33, 34

IM Information Model. 8

IoT Internet of Things. 1

IP Internet Protocol. 22, 24, 25

JSON JavaScript Object Notation. 16, 21, 23, 25

ML Machine Learning. 1, 2, 21, 22, a, 28, 30–32

MLFO Machine Learning Function Orquestrator. 2, 10, 11, 13, 14, 21–23, 25, 26, 30–32, 38, 39

NBI Northbound Interface. 6, 10, 23, 38

NF Network Function. 8

NFS Network File System. 12, 31

NFV Network Function Virtualization. 7

NP Nondeterministic Polynomial. 9

NSO Network Service Orchestration. 7

ODL Open Daylight. 13, 14, 39

ONAP Open Network Automation Platform. 7, 8

OpenMANO Open Management and Orchestration. 7, 8

OS Operating System. 3

OVS Open Virtual Switch. 12, 13, 31

PNF Physical Network Functions. 7

REST Representational State Transfer. 21, 23–25

SDN Software Defined Network. 5–7, 10, 13, 21, a, 29–31, 39

SLA Service-Level Agreement. 6, 8

TOSCA Topology and Orchestration Topology and Orchestration Specification for Cloud Applications. 7, 8

UI User Interface. 12, 28

VIM Virtual Interface Manager. 2

VIO Virtual Interface Orchestrator. 10, 11, 21, a

VL Virtual Link. 7

VLAN Virtual Local Area Network. 10, 12–14, 22, 24–26

VM Virtual Machine. 3, 31

VNF Virtual Network Function. 7

VNI VXLAN Network Ident. 5, 13

VTEP VXLAN Tunnel End Point. 5, 13

VXLAN Virtual Extensible Local Area Network. 5, 10, 13, 22, 24, 39

WF Workflow. 21, 22, 27, 31, 32

WSGI Web Server Gateway Interface. 23

YANG Yet Another Next Generation. 7

Chapter 1

Introduction

A previous publication was done on this work on conference ECOC2021 referenced on [1].

The rise of Machine Learning (ML) algorithms for optical network automation entails analyzing heterogeneous monitoring data collected from monitoring points in network devices. [2] Because network entities can be reconfigured, e.g., lightpath rerouting, it is of paramount importance to link different ML functions (i.e., performance data collection, pre-processing, analysis, storage, visualization, etc.) among them to create an ML Pipeline and to the related network entity (e.g., a lightpath).

In the context of Intent-Based Networking (IBN), where the intents of the business layer are fulfilled by the intent layer, machine learning pipelines can be deployed to ensure that the network continues to deliver the intent (for example by predicting resource usage) based on the policies on it. [3] Because of the amount of data and the almost real-time requirements of the scenario, these ML pipelines, have to be orchestrated to optimize the network utilization and delay.

On the other hand with the recent development of the IoT industry, it's important to have different levels of aggregation for automatic and intelligent decision making. Edge/Fog computing is the concept of making part of these decisions on the edge data centers to have a faster response time. This kind

of complex ML pipeline needs a complex orchestration to place the different ML functions in locations that fulfill certain constraints (for example the response time for an autonomous car). Furthermore, IoT devices are many times mobile, and the ML pipeline must adapt to this mobility, and at the same time optimize resource utilization. [4]

These recent advances in multiple areas have drawn the attention of the ML and 5G community into the definition and development of an orchestrator specifically designed for ML, a Machine Learning Function Orchestrator (MLFO), as shown by the ITU AI/ML in 5G Challenge – "Demonstration of Machine Learning Function Orquestrator (MLFO) via reference implementations (ITU-ML5G-PS-024)". [5]

The MLFO is a component in a more complex IBN framework, where it uses data of the data lake, and interacts with the Multi-VIM Orchestrator and the Network Orchestrator. [6]

The objective of this project is to implement a reference implementation of a MLFO, that can deploy and reconfigure an ML pipeline, and demonstrate its utilization by the orchestration of an example application on an example use case. On top of that, take some reference measurements and design decisions that can help future developments as a baseline to compare to or get ideas.

The example use case consists of the monitoring of a light path that collects data from the monitoring points (on the optical switches) and aggregates it to verify its correct functioning. When the lightpath is reconfigured, the monitoring endpoints change, and the ML pipeline has to be reconfigured to adapt to these changes. This scenario was selected because it makes use of the deployment and reconfiguration, and because of the importance of allocating resources efficiently to avoid impacts on the system's performance.

Chapter 2

Background

2.1 Containerization

Virtualization was first introduced to take better advantage of hardware resources. More than one application can run on a mainframe machine, and different applications have different requirements. By introducing a level of isolation, better control over the resources used by each application can be done, and compatibility issues can be avoided (i.e. by independent library updates). Virtualization is a way to isolate multiple services by abstracting from the underlying hardware. [7]

As shown in figure 2.1, a virtual machine (VM) uses a hypervisor to simulate hardware and runs a fully functional operating system (OS) on top of it. On the other hand, containerization, also known as Operating-system-level virtualization, does not need a hypervisor and keeps a set of advantages by sharing some libraries and the OS kernel with the host system. [8]

While VMs achieve a higher level of isolation, are more portable, and have some features as snapshots; containers start faster, take better advantage of hardware resources, and have smaller, more practical images. So depending on the scenario one or the other option is better. [8]

Kubernetes is an open-source system that automates the deployment and

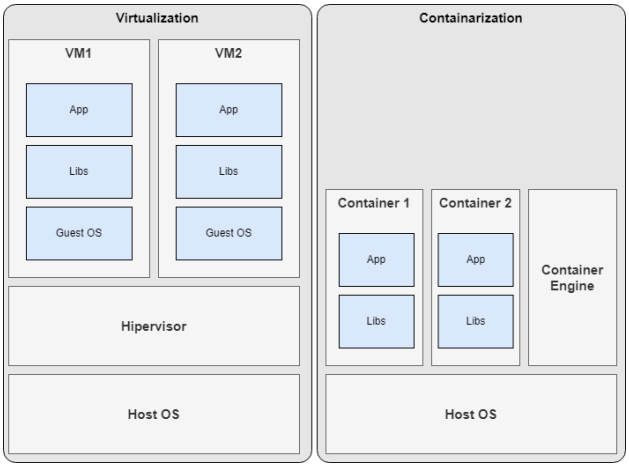


Figure 2.1: Comparison between containerization and virtualization.[8]

management of containerized applications. It is composed of a series of worker nodes where the containers run and a control plane that interacts with the nodes to configure them. The control plane exposes an API through which the different components and external users can communicate. [9]

The Kubernetes API lets an external user manipulate the state of the different resources through it, like the creation of new container instances, the exposure of services, among many others.

2.2 Overlay Network

Overlay networks are a better option to replace Flooding-based systems that do not scale well in bandwidth utilization. An overlay network is a network that is built on top of another network. Nodes in the overlay network can be thought of as being connected by virtual links, that correspond to a path through one or more physical links in the underlying network. Overlay networks do not require or cause any changes to the underlying network. [11]

A way to implement an overlay network is through tunneling. A tunneling protocol is a communications protocol that allows abstracting the communication between two networks through another network that connects them

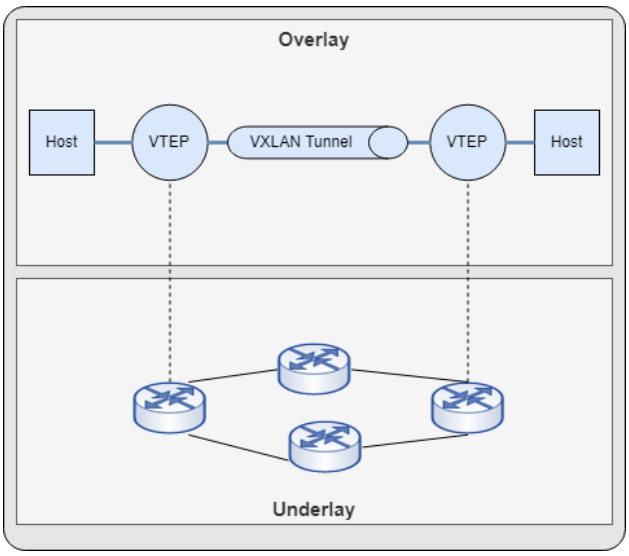


Figure 2.2: Ejemplification of an overlay network using a VXLAN tunnel. [10]

(such as the Internet) by encapsulating the payload.

One tunneling protocol, the Virtual Extensible Local Area Network (VXLAN), was designed for a scenario with Layer 2 and Layer 3 data center network infrastructure in the presence of VMs in a multi-tenant environment. VXLAN Network Ident (VNI) is in an outer header that encapsulates the inner MAC frame. VXLAN is a Layer 2 overlay scheme on a Layer 3 network and could also be called a tunneling scheme to overlay Layer 2 networks on top of Layer 3 networks. The end VXLAN Tunnel End Point (VTEP) can be located on the hypervisor or a switch and is responsible for encapsulating and decapsulating messages as can be seen in Figure 2.2. [10]

2.3 Software Defined Network

Multiple efforts have been made to define programmable networks, but the one that has received the most attention lately is the SDN approach.

In figure 2.3 SDN architecture is divided into four layers. The application

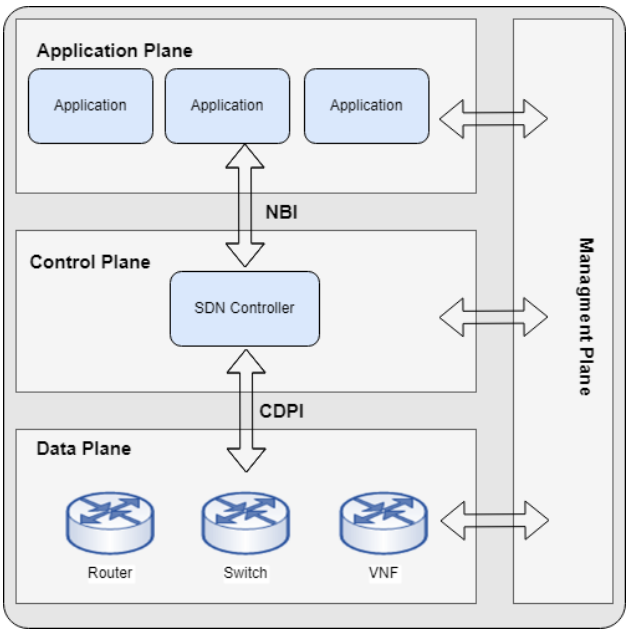


Figure 2.3: SDN architecture overview.2.3

plane communicates network requirements and desired network behavior to the Control Plane through the NBI. The application plane may expose a higher-level interface for network control. The control plane translates the application layer requirements to the Data plane and provides the application with an abstract view of the network. The control plane is sometimes also known as Network Operating System. The data plane consists of network devices that expose control over its forwarding and processing capabilities. The Control to Data-Plane Interface (CDPI) is the interface defined between an SDN Controller and an SDN Datapath. The SDN Northbound Interface (NBI) is the interface between applications and SDN Controllers. The Management plane covers the tasks that are better handled outside the application, control, and data planes (e.g. monitoring, credentials, configurations, SLA, element setup, etc). [12]

The best-known CDPI for switches is Openflow. Openflow is a protocol that standardizes the way the SDN controller communicates with the switches without requiring the vendors to provide a programmable platform or expose information about the internal working. Openflow allows controlling the flow tables on a switch by providing a standardized interface to add and remove

flow entries. Flow entries are composed of a header that is used to match packages, a counter, and an action to apply to the matching packages. When the switch receives a package, it compares it against the flow table and if matching is found the associated action is performed, otherwise the package is forwarded to the controller that is responsible for determining how to handle the package. [13]

2.4 Network Service Orchestration

Deploying and operating end-to-end services has traditionally been hard work. SDN and NFV advances enabled new ways for network operators to create and manage services. Orchestration refers to automatically managing multiple resources, services, and systems to meet certain objectives. [14]

Network Service Orchestration (NSO) can be defined as "the automated management and control processes involved in end-to-end services deployment and operations performed mainly by telecommunication operators and service providers" [14], but there's no full agreement on the term and multiple definitions exist with a focus on different aspects of the orchestration of an end-to-end service.

In general, an orchestrator manages Virtual Network Function (VNF) that is software that runs over a virtual environment over generic hardware, Physical Network Functions (PNF) that is software that runs over specialized hardware, and Virtual Link (VL) that are virtual connections between components. These definitions are based on the Topology and Orchestration Topology and Orchestration Specification for Cloud Applications (TOSCA) template language, which is a standard language used by orchestrators to define network services and their relationships. [15] YANG is a complementary technology with some overlap that is used mainly to define the configuration of network devices. [16]

Two NSO frameworks deserve special attention because they are the strongest in the market: Open Management and Orchestration (OpenMANO) and Open Network Automation Platform (ONAP).

OpenMANO is an open-source project. It follows the ETSI's NFV ISG standardization and is used by Telefonica with a strong presence in Europe. Uses an Information Model (IM) that is infrastructure agnostic, and YANG for service configuration. It has an option to instantiate Kubernetes services as part of a service using Helm charts. [17]

ONAP is a policy-based orchestrator that has more presence outside Europe. It has integration with multiple hardware and virtualization software. It's more mature than OpenMANO. It uses TOSCA for the service design and divides the orchestration labor into three stages: design, deployment, and operations. The design would be the steps involved in the planification and creation of a new TOSCA file. The deployment would be the stage where the coordination with the cloud is done to deploy the corresponding services. And finally, the operation is where the monitoring is done and some actions are taken (ie. self-healing) based on events and the corresponding Event-Condition-Action policies. [18]

Other orchestrators that are also interesting to look into are T-NOVA, Central Office Re-architected as a Datacenter (CORD), and Cloudify. They are not so well known, but each is specialized in a different aspect of orchestration and introduces interesting ideas.

T-NOVA is an open-source orchestrator with a focus on being able to offer network services through a marketplace. It divides the network function (NF) lifecycle in on-boarding/deployment, instantiation, supervision, scaling, and termination. Can define service-level agreement (SLA), and can react in real-time to meet its requirements. [19]

CORD is an orchestrator with a focus on managing a single data center. It has some support for Kubernetes. It is a framework that has multiple implementations for different scenarios.

Cloudify is an open-source multi-cloud and edge orchestration platform. It is focused on cloud services and has integration with Amazon web services (AWS) and Azure. It has complete integration with Kubernetes with different cluster managers. Uses blueprints to describe services, it uses infrastructure as a service. The downside is that the code repository is not independent of the infrastructure, having for example different scripts to start the service in different environments. [20]

2.5 Discrete optimization

An optimization problem is a problem where the objective is to minimize a function by finding the optimal values of a series of variables. Each variable belongs to a domain, and a series of constraints may be defined over the variables. Discrete optimization problems are optimization problems where all variables are discrete, also called combinational problems. [21]

An integer programming problem is a mathematical optimization in which some or all of the variables are restricted to be integers. It is also an integer linear programming (ILP) if the objective function and the constraints are linear. Integer programming is NP-complete. The Simplex method is a popular algorithm to find an exact solution to integer linear programs. [22]

Since integer linear programming is NP-hard, heuristic methods can be used instead for a faster but less optimal solution. Heuristics are a way to find a rapid solution to a problem without any proof that it is the optimal solution.

One of the most basic heuristics is Greedy, which is a constructive algorithm that on each step of the solution construction chooses the local optimal, without caring if that step will prevent better steps later or even render the solution impossible. [23]

Chapter 3

The solution

This section will have details on the solution architecture. How the cluster is configured and the different services involved in the deployment and reconfiguration of the services.

3.1 Cluster Architecture

From Figure 3.1 it can be seen that the MLFO has a NBI to receive the instruction of deployment and reconfiguration. The MLFO has as southbound interfaces the SDN controller, in charge of configuring the tunneling between datacenters, and the VIO services, in charge of configuring the computing instances. Each datacenter has one or more computing nodes where the instances can run and a master switch, that is managed by the SDN controller, where the tunneling between datacenters is configured. Each deployed application can have one or more private networks (VLAN) that can be configured to follow a tunnel (VXLAN). Another important component is the dynamic configuration (ConfigMap) that can be attached to a computing instance and modified on reconfiguration by the VIO.

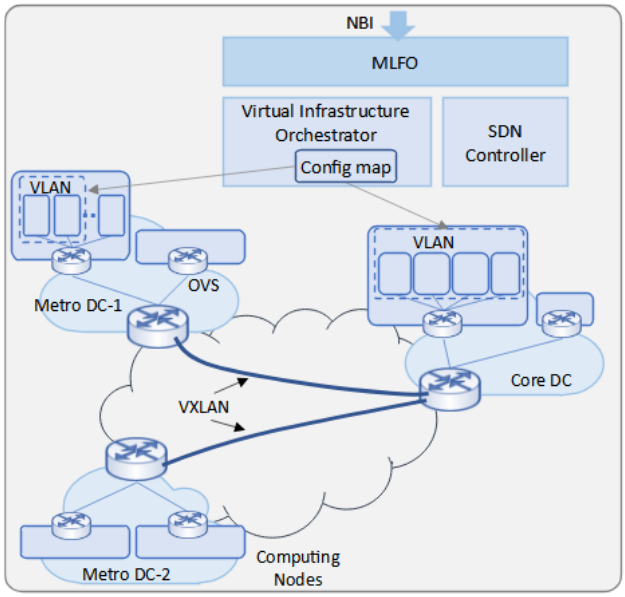


Figure 3.1: Details on the cluster configuration of the proposed solution, showing the different components involved.

3.1.1 Virtual Interface Orquestrator

The VIO is responsible for managing the computing instances and the resources on each cluster. Different technologies can be used, but for this project, Kubernetes [v1.19.2] was used, with docker [v18.09.7] as the container technology. These technologies were selected because of the advantages of using containerization against virtualization, as the near-native performance and the short startup time (that is important for the MLFO, because of the near real-time scenarios), but at the same time, it maintains some of the advantages of virtualization as the isolation and security (that are also important because of the multitenancy characteristics of the MLFO).

The container images were stored on Docker Hub, but a local repository of images may be better, because of the downloading time overhead added to the deployment if the image is not in the cache.

For the dynamic configuration files, the Kubernetes ConfigMap was used, mounted into Kubernetes containers as read-only files. This way of mounting

the configurations allows having a dynamic configuration where the changes made to it are replicated to the files on the containers where it is mounted.

A reverse proxy was configured on the core datacenter using the ingress-nginx controller. The ingress exposes services through the MetalLB load balancer. This allows the deployed applications to have a public User Interface (UI) or Application Programming Interface (API).

For persistence, a Network File System (NFS) server was configured on the core datacenter, and attached to the containers through the Persistent Volume Claim configuration.

The service exposure and the persistence features can only be used from containers deployed to the core datacenter. If a computing instance needs one of these features it can be added as a restriction and will only be able to be deployed to the core datacenter. Other solutions can be found to make this feature available on more than one datacenter, but it escapes the scope of this project.

3.1.2 Software Defined Network

Part of the networking is configured through Kubernetes, part of it is preconfigured and part of it is managed by the SDN controller.

The private networks (VLAN) are configured through the Multus CNI and the Open Virtual Switch (OVS) plugins of the Container Network Interface (CNI) [24] of Kubernetes. Each Kubernetes node has an open virtual switch (OVS) [25] where the different VLANs are configured by Kubernetes. The Multus plugin allows to have more than one network interface for each container, and the OVS plugin allows to configure a VLAN on the OVS switch through Kubernetes.

The switches on each datacenters are configured in a way that allows all the messages generated on the datacenter to reach all other nodes on the datacenter and the master switch. But the message will not be seen by containers that do not belong to the same VLAN, allowing to have isolation between applications.

The tunneling technology used for the inter-datacenter tunnels is VXLAN. Tunnels are preconfigured between datacenters, and each connection is assigned a VNI, then the master switch on each datacenter will decide which packages to send through the tunnel and which not to. The master switch will act as a VTEP, encapsulating and decapsulating the messages that go through the tunnel.

For the SDN controller, OpenDayLight (ODL) [26] was used, which communicates with the master switch on each datacenter through the OpenFlow protocol. The MLFO pairs the VLANs with VXLANs on each of the master switches of the datacenters that are involved in a connection between two containers through the switch OpenFlow tables. The tunnels will only forward the messages belonging to a VLAN that is paired with the VXLAN and that were generated on the datacenter.

A simplified example of how the pairing of the VLAN with the VXLAN is done can be seen in the following code snippets:

```
> ovs-vsctl show
...
Bridge "vtep"
    Controller "tcp:odl.mlfo.gco"
        is_connected: true
    Port "vxpatch"
        Interface "vxpatch"
            type: patch
            options: {peer="patch1"}
    Port "vtep"
        Interface "vtep"
            type: internal
    Port "ovry"
        Interface "ovry"
            type: vxlan
            options: {
                key=flow,
                remote_ip=flow
            }
```

This first command-line snippet shows the "vtep" bridge configuration of the master switch (in this case an OVS switch), showing the configuration of the SDN controller of the switch on line 4, and the VXLAN options that are configured through the OpenFlow table on lines 17 and 18.

```
> ovs-ofctl add-flow vx "priority=20,tun_id=10,actions=output:normal"
> ovs-ofctl add-flow vx "priority=10,dl_vlan=104,actions=resubmit(,1)"
> ovs-ofctl add-flow vx "table=1, actions=set_field:metro1->tun_dst,set_field:10->tun_id,output:300"
> ovs-ofctl add-flow vx "priority=1,actions=output:normal"
```

The second command-line snippet shows the configuration of the OpenFlow tabes, where line 1 says that if a message comes from the tunnel with id 10, it must be forwarded to the datacenter network, line 2 says that if a message belongs to VLAN 104, it must be forward to table 1, where line 3 says that the message will be forwarded through tunnel 10 to the master switch in datacenter "metro1", and line 4 says that the rest of the messages should be treated as internal messages. This configuration allows adding new instructions to forward other VLANs to the tunnel, and to have more than one tunnel.

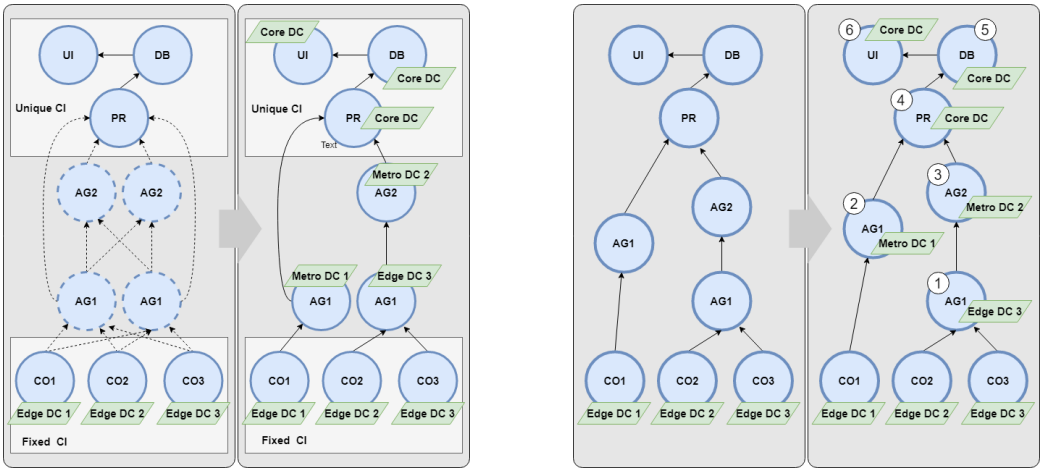
More pairings are added and stale pairings are deleted through the ODL controller by the MLFO when an application is deployed or reconfigured.

3.2 Optimization algorithm

Two optimization algorithms were developed to compare results and set a baseline for future implementations. Each solution has its advantages and disadvantages that will also be shortly discussed.

3.2.1 Optimization problem

The problem we are trying to solve is to find the best pipeline and mapping for a given application and infrastructure. The problem has two versions, the first one is the deployment that is the one we will get into details in this section, and the second one is the reconfiguration, which is almost the same, but can also take into consideration the previous deployment to minimize the number of changes to be made.



(a) ILP. (b) Greedy.

Figure 3.2: Visual representation of the Integer Linear Program (ILP) (3.2a) and the greedy (3.2b) algorithms. To the left of each image, we can see the template with the dotted lines representing optional links and computing instances (greedy does not allow optional resources). To the right of each image is the result of the algorithm.

The problem can be formally stated as:

Given:

- Infrastructure: graph $I = (DC, T)$, where DC represent datacenters and T the tunnels between datacenters, both can have information on constraints, i.e. maximum/available capacity for a tunnel or hardware requirements for datacenters; and costs, i.e. cost by instance or cost by megabyte.
- Pipeline template: graph $PT = (CI, CC)$, where CI represents the computing instances, and CC are the connections between the computing instances, both computing instances and computing connections can be optional, also they may be some constraints on the graph as mutually exclusive arcs or the maximum number of connections for a computing instance.
- Fixed instances: set $F = \{< ci, dc > | \exists < ci, dc > \in CI \times DC\}$, where each pair represents a computing instance that is fixed to a datacenter.

Output: the pipeline P that is a subgraph of PT that satisfies the graph restrictions, and the mapping $M = \{< ci, dc > | \exists < ci, dc > \in CI \times DC\}$, $F \subseteq M$, that satisfies the infrastructure restrictions.
Objective: Minimize some utility function, i.e. a weighted sum (by cost) of the resources used.

The algorithms were developed to solve a subset of the original problem, but a subset big enough to be useful for a real application. Both solutions are based on minimizing the cost of resources utilization.

3.2.2 Integer linear program solution

The python package PULP was used to solve the problem. A preprocessing and postprocessing were implemented to transform the input and output from/to JSON format.

The variables, objective function, and constraints with their explanation are defined below:

- Given variables:
  - N^{CI} number of computing instances.
  - CI list of computing instances.
  - DC list of datacenters.
  - U_{ci} 1 if computing instance ci is a singleton, 0 otherwise.
  - $F_{ci,dc}$ 1 if computer instance ci has a fixed location in datacenter dc, 0 otherwise.
  - CC_{ci_1,ci_2} 1 if ci_2 is a candidate for a mandatory connection from ci_1, 0 otherwise.
  - TC_{dc_1,dc_2} capacity of the tunnel between cd_1 and dc_2, or 0 if no tunnel exists.
  - DCC_{dc} the capacity of datacenter dc.
  - DS_{dc} the cost of running a process in datacenter dc.
  - TS_{dc_1,dc_2} the cost of a connection between dc_1 and dc_2.
- Decision Variables:
  - E_{ci} 1 if computing instance ci will be allocated, 0 otherwise.
  - C_{ci_1,ci_2} 1 if a connection between ci_1 and ci_2 exists.
  - $M_{ci,dc}$ 1 if computing instance ci is allocated on datacenter dc, 0 otherwise.
  - T_{dc_1,dc_2} is the number of connections between dc_1 and dc_2.
  - EC_{ci_1,ci_2,dc_1,dc_2} 1 if connection between ci_1 and ci_2 should go through a tunnel between dc_1 and dc_2, 0 otherwise.

ILP problem statement:

$$min \sum_{ci \in CI, dc \in DC} M_{ci,dc} \times DS_{dc} + \sum_{dc_1 \in DC, dc_2 \in DC} T_{dc_1,dc_2} \times TS_{dc_1,dc_2} \tag{3.1}$$

Subject to:

$$\sum_{dc \in DC} M_{ci,dc} = E_{ci}, \forall ci \in CI \tag{3.2}$$

$$\sum_{ci_2 \in CI} C_{ci_1,ci_2} \times CC_{ci_1,ci_2} = E_{ci_1}, \forall ci_1 \in CI \tag{3.3}$$

$$\sum_{ci_1 \in CI} C_{ci_1,ci_2} \leq E_{ci_2} \times N^{CI}, \forall ci_2 \in CI \tag{3.4}$$

$$\begin{gathered} M_{ci_1,dc_1} + M_{ci_2,dc_2} + C_{ci_1,ci_2} - 2 \leq EC_{ci_1,ci_2,dc_1,dc_2}; \\ \forall ci_1, ci_2 \in CI; dc_1, dc_2 \in DC \mid dc_1 \neq dc_2 \wedge ci_1 \neq ci_2 \end{gathered} \tag{3.5}$$

$$\sum_{ci_1,ci_2 \in CI} EC_{ci_1,ci_2,dc_1,dc_2} = T_{dc_1,dc_2}, \forall dc_1, dc_2 \in DC \mid dc_1 \neq dc_2 \tag{3.6}$$

$$T_{dc_1,dc_2} \leq TC_{dc_1,dc_2}, \forall dc_1, dc_2 \in DC \mid dc_1 \neq dc_2 \tag{3.7}$$

$$\sum_{ci \in CI} M_{ci,dc} \leq DCC_{dc}, \forall dc \in DC \tag{3.8}$$

$$U_{ci} \leq E_{ci}, \forall ci \in CI \tag{3.9}$$

$$\sum_{dc \in DC} F_{ci,dc} \leq E_{ci}, \forall ci \in CI \tag{3.10}$$

$$F_{ci,dc} \leq M_{ci,dc}, \forall ci \in CI, dc \in DC \tag{3.11}$$

The objective function (equation 3.1) is to minimize the cost of the connections plus the cost of the processors, subject to:

- Equation 3.2: If a computing instance is allocated assign to one, and only one, datacenter.
- Equation 3.3: If a computing instance is allocated, it must have all mandatory connections.
- Equation 3.4: If a computing instance has incoming connections then it must be allocated.
- Equation 3.5: Allocate inter datacenter connections.
- Equation 3.6: Allocate datacenter connections for computer instance inter datacenter connections.
- Equation 3.7: Ensure connections do not overpass tunnel capacity and do not use unexisting tunnels.

- Equation 3.7: Ensure that the number of computing instances does not overpass the datacenter capacity.
- Equation 3.9: Force unique computing instances to exist.
- Equation 3.10: Force fixed computing instances to exist.
- Equation 3.11: Force fixed computing instances mapping to the corresponding location.

In Figure 3.2a we can see a visual representation of the MIP algorithm. It can be seen the different sets of unique computing instances and fixed computing instances. It can also be seen how not all the computing instances are on the solution to the right of the figure. Also in the solution, there is always a path between the collectors and the processor, and the result is a tree.

This solution can't model more complex applications with more than one mandatory connection for each computing instance.

On the good side, the result it will generate will be optimal, reducing the cost (resource utilization). The model can easily be extended to use the amount of computation resources and the amount of bandwidth resource needed by each computing instance and its connections instead of counting each computing instance and each connection per unit.

3.2.3 Greedy solution

The greedy algorithm was developed in python and does not need any extra package. It has two parameters that are the penalizer for pending connections ($PENDING\_PENALIZER$) and an incentive for the non-pending connections ($NON\_PENDING\_INCENTIVE$). These parameters control how important it is to reduce the number of pending connections in each step. The parameter $PENDING\_PENALIZER$ was set to 1000 and the $NON\_PENDING\_INCENTIVE$ was set to 100.

Algorithm 1 Greedy algorithm

1: $S \leftarrow \emptyset$
2: $C \leftarrow$ computing instances ids.
3: $D \leftarrow$ datacenters ids.
4: $P \leftarrow$ computing instances that are not assigned to a datacenter.
5: $F \leftarrow$ pair of computing instance id and datacenter id for each fixed location computing instance.

6: $datacenter(c) \leftarrow$ the datacenter assigned to computer instance c
7: $neighbors(c) \leftarrow$ the computing instances that have a connection with c.
8: $datacenter\_cost(d) \leftarrow$ the cost of running a computing instance on datacenter d.
9: $tunnel\_cost(d_1, d_2) \leftarrow$ the cost of using a tunnel or ∞ if no tunnel exist between d_1 and d_2 or 0 if $d_1 = d_2$.

10: **for** $< c_{id}, d_{id} > \in F$ **do**
11: $\quad S \leftarrow S \bigcup \{< c_{id}, d_{id} >\}$
12: **while** $P \neq \emptyset$ **do**
13: $\quad H \leftarrow \emptyset$
14: $\quad$ **for** $c_{id} \in C$ **do**
15: $\quad\quad$ **for** $d_{id} \in D$ **do**
16: $\quad\quad\quad cost \leftarrow datacenter\_cost(d)$
17: $\quad\quad\quad$ **for** $n_{id} \in neighbors(c_{id})$ **do**
18: $\quad\quad\quad\quad$ **if** $n_{id} \in P$ **then**
19: $\quad\quad\quad\quad\quad cost \leftarrow cost + PENDING\_PENALIZER$
20: $\quad\quad\quad\quad$ **else**
21: $\quad\quad\quad\quad\quad cost \leftarrow cost + tunnel\_cost(d_{id}, datacenter(n_{id})) - NON\_PENDING\_INCENTIVE$
22: $\quad\quad\quad\quad H[< c_{id}, d_{id} >] \leftarrow cost$
23: $\quad$ **if** $min_{<c_{id},d_{id}> \in C \times D}(H[< c_{id}, d_{id} >]) \geq \infty$ **then**
24: $\quad\quad$ **return** $INFEASIBLE$
25: $\quad < c_{id}, d_{id} > \leftarrow argmin_{<c_{id},d_{id}> \in C \times D}(H[< c_{id}, d_{id} >])$
26: $\quad S \leftarrow S \bigcup \{< c_{id}, d_{id} >\}$
27: $\quad P \leftarrow P - c_{id}$
28: **return** S

The greedy solution is much faster than the ILP solution but as a drawback, it may not find the solution in all cases. It also solves a subset of the problem, where the template can't have optional instances or connections, so in terms of the problem, $P = PT$, but on the good side can solve the problem for applications that are represented by graphs with cycles (not only trees as the integer program solution).

A visual representation of the algorithm input and solution can be seen in Figure 3.2b, where the numbers represent the order in which the computing instances are placed.

3.3 Application Architecture

To demonstrate how the MLFO would work and take some baseline measurements, an implementation was made. The application has two workflows: deployment and reconfiguration, it was developed in python as a group of microservices and has an external REST API that can be interacted with through JSON messages.

3.3.1 Workflows

Figure 3.3 presents the workflows for the initial ML pipeline deployment and any subsequent externally-triggered reconfiguration. Let us start with the deployment workflow (WF1). The management application initiates WF1 by sending the deployment plan (message WF1/1 in Fig. 4). The descriptor contains a template for the ML functions and the connectivity. Next, the deployment is triggered (2) and the MLFO starts a series of steps. First, the MLFO solves the optimization problem using the constraints, resulting in a mapping between the ML functions and the datacenters, and the connectivity and the deployment plan are computed. A list of iterations is generated that includes the communication of the MLFO with the VIO (e.g., Kubernetes) for the deployment of the ML functions (e.g., encapsulated into containers [5]), and with the SDN controller for managing the connectivity among the ML functions. The list iterations include i) the namespace creation (3); ii) the configuration of an image repository storing the different computing

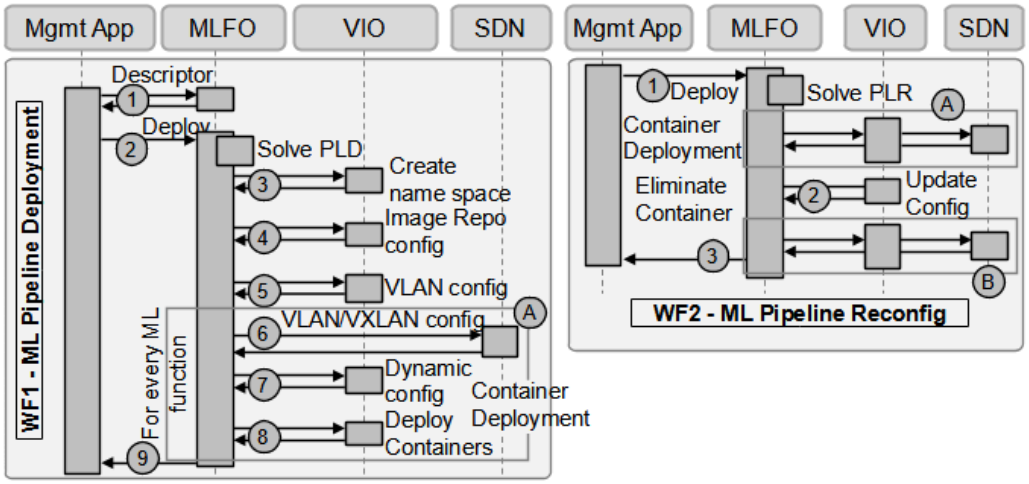


Figure 3.3: Pipeline management showing pipeline template, deployment, and reconfiguration.

images that are retrieved when a new ML function instance is deployed (4); iii) the configuration of the ML pipeline network that entails creating the VLAN (5) and pairing it to the VXLAN tunnels (6); iv) the creation of a volume for the dynamic configuration of the computing instances (7); and v) the deployment of the containers (8). Steps 6-8 are followed for every ML function to be deployed (block A). A reply is eventually sent (9).

WF2 is triggered when the ML pipeline needs to be reconfigured. The management application initiates WF2 by sending a new set of constraints to the MLFO (message WF2/1 in Figure 3.3). The optimization problem is then solved considering the received configuration. Then, the MLFO finds the changes to be performed and prepares a plan with the creation of new ML functions (block A) and the removal of existing ones (block B). Besides, the dynamic configs are updated to reflect the changes in the system (e.g., changes in the IP addresses) (2). When all the steps are executed, the result is sent back to the management application (3).

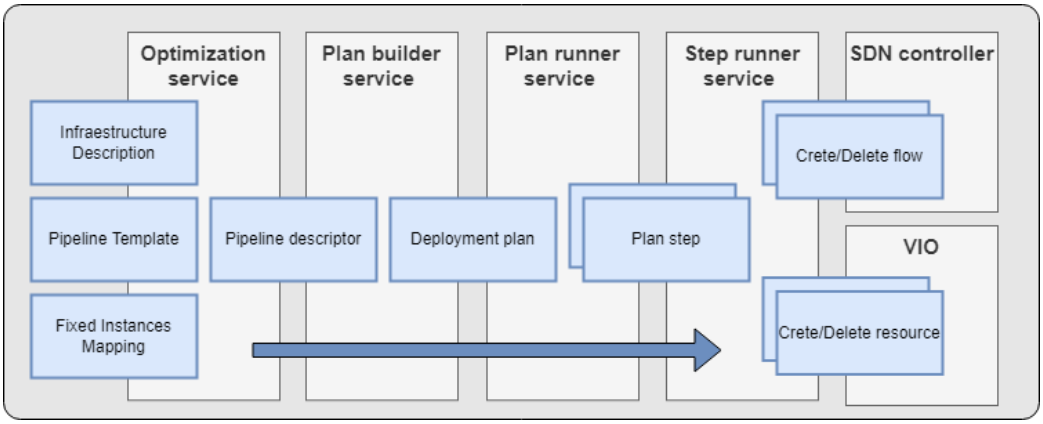


Figure 3.4: Microservice flow of information.

3.3.2 Implementation

In this project an MLFO was implemented using the following technologies: For the MLFO NBI, a REST API that uses short JSON messages was implemented. The MLFO code is developed in python [v3.8.6] and uses Flask [1.1.2] framework and Gunicorn [v20.1.0] as WSGI. The MLFO uses the Kubernetes API as a southbound interface through the python official client library [v11.0.0].

The MLFO application follows a microservice architecture with well-defined interfaces using JSON schemas as shown in Figure 3.4. The microservices are:

- the main service that is the interface with the user and communicates with the rest of the services;
- the optimization service that solves an optimization problem to minimize the cost/use of the resources of the deployment and generates a pipeline descriptor;
- the plan builder service that generates a detailed plan of steps to take to make the pipeline descriptor reality;
- the plan runner service that runs each step of the plan through the step runner service;

- and the step runner service that expands the message using Jinja2 templates and communicates with the SDN controller and the VIO.

Main service

The main service will be the endpoint that the user or client application will communicate with. It offers a REST API to the client that is described in section 3.3.3 and communicates with the rest of the services. It also calculates the difference between one pipeline descriptor and the previous one when deploying an application that was already deployed.

Optimization service

Two implementations were made of the optimization service, one uses a ILP to solve the optimal allocation of computing instances and networks, and the other uses a greedy algorithm. The greedy implementation is faster, but may not find the optimal solution, and may not find a solution at all even if one exists. The problem and the algorithms are explained in section 3.2.1.

Plan builder service

This service will create a detailed plan of steps based on the pipeline descriptor (for first deployments) or the pipeline diff (for subsequent deployments). The steps of the plan can be seen in section 3.3.1.

It will also assign IP addresses and VLAN numbers to the different networks and interfaces of the computing instances and assign the VLANs to the corresponding VXLANs to connect the corresponding computing instances.

It will also render some sections of the model that use Jinja2 templates like the environment variables, dynamic config files, or reverse proxy endpoint paths. This template allows having variables that depend for example on the IP address of a service or the datacenter name that is not known beforehand and are calculated on the optimization or the building of the plan.

Plan runner service

This service will receive a plan and execute through the Step runner service the steps of the plan, it is in charge of coordinating the dependency between steps. In this version, all steps are dependent on the previous step, but this could be changed.

Step runner service

This service acts as a bridge between the MLFO and the southbound interfaces. It is responsible for extending the messages that only contain the important information from the plan, using jinja2 templates to match the format of the southbound interfaces.

It also knows the address of the southbound services and communicates with them. It corroborates that each step took effect before returning.

3.3.3 North Bound interface

The north-bound interface of the MLFO application is a REST API, with well-defined interfaces through JSON schemas. A description of a typical workflow will be described, in which the manager first configures the services and creates an application template, then the management application does the first deployment and after that a reconfiguration, with some references to example messages on the appendix.

Service configuration

The message shown in appendix A.1, the "configure service message" is composed of four sections: the repository configuration that defines the credentials to connect to the image repository ("repositoryconfigs"), the range of IP addresses to be used, and how to divide them between subnetworks ("subnet-generator"), the VLAN range to be used by the applications ("vlan-range") and the connection between datacenters through tunnels ("tunnels"). This

information will then be stored in a database to be used by the different services of the MLFO, and updated accordingly (for example the VLANs are marked as in-use or free).

Template creation

As can be seen in appendix A.2, the "create template message" is composed of an application name ("appname"), a reference to a repository ("repositoryconfig"), and two lists: the "containers-models" list and the "template" list. The container-model list defines the different kinds of functions that an application will use, and the template list defines how to instantiate them and the connections between them.

Each container model is defined by a name ("name") and an image name from the image repository ("image"). They may optionally also include a readiness prove ("readiness" were to request to see if the container is ready to start working), some configurations that may be jinja2 templates that are filled with data from the application ("configs"), a dynamic config file that is a file that is mounted into the container and may change its content when changes on the application are made ("dynamic-config-files" where jinja2 templates can also be used), and/or an external endpoint to be able to access the service from outside the infrastructure ("endpoint").

Each template has a reference to a model from the "container-model" list ("model_name"), a function name ("template_function"), and an instantiation type ("type") that can be: unique (that means that only one instance is permitted), custom (that means that the number of instances and location must be defined by the user at the moment of the deployment) or auto (that means that the number and location of this function will be defined by the optimization function). Optionally they may also have the maximum number of allowed instances ("max" only for the auto instantiation type), a list of connections where only one of them is mandatory ("connections"), and/or a data section that can be used on the container configurations through the jinja2 templates ("data").

Pipeline deployment

For the pipeline deployment, the message defined in appendix A.3 is used. This message defines a mapping between the template functions (defined on the template) and the datacenters, and a data section that is merged with the data section defined on the template. This message will trigger the WF1, as defined in section 3.3.1.

Pipeline Reconfiguration

For the pipeline reconfiguration, the same message as in pipeline deployment is used. The difference is that the application is already deployed and it will trigger WF2 this time.

Chapter 4

Results

4.1 Example application

To experiment and show how the MLFO works a simple monitoring application was developed. The application has five different ML functions:: i) collector (Co) in charge of collecting monitoring data from activated monitoring points (M); ii) aggregator (Ag) that collects measurements from several different collectors and perform some not computationally intensive task, like computing some statistics, e.g., max, min, and average; iii) processor (Pr), which performs a more computational intensive task on the received data; iv) a time series Database (DB), and v) an User Interface (UI) (Figure 4.1).

In the example application, the collector generates random data and pushes it to the aggregator, the aggregator performs a sum aggregation over the values and the number of measurements and push the data to the processor, the processor does a final aggregation over all the data, the DB is a Prometheus [27] that pulls the data from the processor and stores it in a persistent way, and the UI is a Grafana [28] that shows graphs based on the data collected (e.g. the number of messages collected from each source).

The collectors, aggregators, and processors use an output queue that does not need persistence, but that is waited to be empty on the graceful shutdown. The database is configured to use persistence, and the user interface

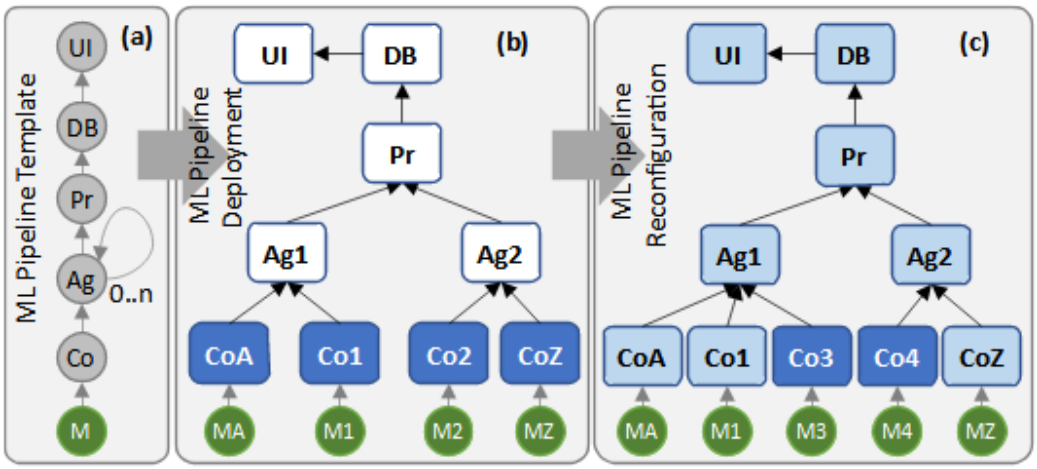


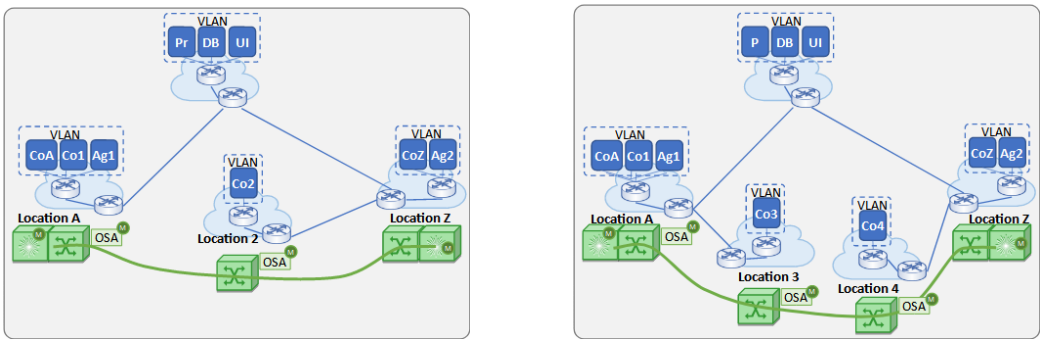
Figure 4.1: Pipeline management showing pipeline template, deployment, and reconfiguration.

is configured to expose itself through the reverse proxy.

4.2 Lightpath scenario

The scenario we will take as an example to demonstrate how the MLFO implementation works, is the monitoring of a lightpath. The monitoring data can then be used to reconfigure the lightpaths. The importance of state information of each node to establish lightpath can be seen for example in [29], [30] and [31]. The data needed to decide on these reconfigurations is of heterogeneous sources and is collected from different sources on the lightpath. All this monitoring information needs to be collected, processed, and then available in a centralized place for the SDN controller to consult it and be able to make the correct decisions on how to manage the lightpaths.

This data need to be collected and aggregated efficiently because of the kind of validations over the data that can be done (for example anomaly detection that need almost real-time data), and because the monitoring data use the network resources (as bandwidth and computation) that if not handled correctly may impact the network performance.



(a) Lightpath setup.

(b) Lightpath rerouting.

Figure 4.2: Example showing the ML pipeline status before and after lightpath reconfiguration.

Furthermore, because the lightpath may be reconfigured the data sources may change with time, the data collection pipeline has to adapt to the lightpath reconfiguration as can be seen in Figure 4.2. In the figure a reconfiguration of the lightpath triggers a reconfiguration on the ML pipeline, and only the necessary changes are made using the same computing instances, and only removing the "Co2" from the old location, and adding the "Co3" and "Co4" to the new locations.

All this justifies the utilization of an MLFO for this scenario. To synthesize, the scenario is the monitoring of a lightpath by collecting data from different endpoints to feed ML algorithm (as an example can be anomaly detection), that makes decisions on the lightpath reconfiguration through an SDN controller. Then the data collection infrastructure has to adapt to the reconfigured lightpath in a way that it does not lose information.

4.3 Empirical demonstration

This first experiment consists in demonstrating the functionality of the developed software by deploying and reconfiguring the example application. As shown in Figure 4.1, first a ML pipeline with four collectors is deployed, and for reconfiguration, two new collectors are added and one is deleted.

	Time	Source	Destination	Protocol	Info
1	*REF*	Application	MLFO	HTTP	POST /appmodels HTTP/1.1 (application/json
1	0.001916	MLFO	Application	HTTP	HTTP/1.1 201 CREATED (application/json)
2	0.007409	Application	MLFO	HTTP	POST /appmodel/mlp/latest/deploy HTTP/1.1
3	0.220487	MLFO	Kubernetes	TLSv1.3	Application Data
3	0.221391	Kubernetes	MLFO	TLSv1.3	Application Data
4	0.241095	MLFO	Kubernetes	TLSv1.3	Application Data
4	0.241958	Kubernetes	MLFO	TLSv1.3	Application Data
5	0.258507	MLFO	Kubernetes	TLSv1.3	Application Data
5	0.259509	Kubernetes	MLFO	TLSv1.3	Application Data
6	0.268173	MLFO	ODL	HTTP	POST /restconf/config/opendaylight-inventor
6	0.302226	ODL	MLFO	HTTP	HTTP/1.1 204 No Content
6	0.306905	MLFO	ODL	HTTP	POST /restconf/config/opendaylight-inventor
6	0.314857	ODL	MLFO	HTTP	HTTP/1.1 204 No Content
7	18.758266	MLFO	Kubernetes	TLSv1.3	Application Data
7	18.759219	Kubernetes	MLFO	TLSv1.3	Application Data
8	18.782397	MLFO	Kubernetes	TLSv1.3	Application Data
8	18.783318	Kubernetes	MLFO	TLSv1.3	Application Data
9	61.135778	MLFO	Application	HTTP	HTTP/1.1 200 OK (application/json)

(a) Deployment Workflow (WF1).

	Time	Source	Destination	Protocol	Info
1	*REF*	Application	MLFO	HTTP	POST /appmodel/mlp/latest/deploy HTTP/1.1 (
A	0.190359	MLFO	ODL	HTTP	POST /restconf/config/opendaylight-inventory
A	0.196360	ODL	MLFO	HTTP	HTTP/1.1 204 No Content
A	0.200539	MLFO	ODL	HTTP	POST /restconf/config/opendaylight-inventory
A	0.209240	ODL	MLFO	HTTP	HTTP/1.1 204 No Content
2	8.371141	MLFO	Kubernetes	TLSv1.3	Application Data
2	8.372115	Kubernetes	MLFO	TLSv1.3	Application Data
B	8.456727	MLFO	Kubernetes	TLSv1.3	Application Data
B	8.457567	Kubernetes	MLFO	TLSv1.3	Application Data
B	8.476126	MLFO	Kubernetes	TLSv1.3	Application Data
B	8.477026	Kubernetes	MLFO	TLSv1.3	Application Data
B	8.485187	MLFO	ODL	HTTP	DELETE /restconf/config/opendaylight-invento
B	8.490304	ODL	MLFO	HTTP	HTTP/1.1 200 OK
B	8.493927	MLFO	ODL	HTTP	DELETE /restconf/config/opendaylight-invento
B	8.502459	ODL	MLFO	HTTP	HTTP/1.1 200 OK
3	8.505793	MLFO	Application	HTTP	HTTP/1.1 200 OK (application/json)

(b) Reconfiguration Workflow (WF2).

Figure 4.3: Message capture during workflows.

The experiment was run using the Greedy version of the algorithm. The Kubernetes cluster was deployed over a set of VMs managed by OpenStack, as well as the SDN controller, the OVS master switches, and the NFS server. Six datacenters were defined inside the Kubernetes cluster, each managed by a different Openstack cluster, and all connected by a physical switch. Each microservice of the MLFO was deployed into a container on the core datacenter.

Figure 4.3a shows the messages exchanged during WF1 to deploy the ML pipeline; the number of the messages is shown in Figure 3.3 for the sake of clarity. Total deployment time was about 1 min, with most of the time used by Kubernetes to deploy containers.

Figure 4.3b shows the exchanged messages during WF2 to reconfigure the

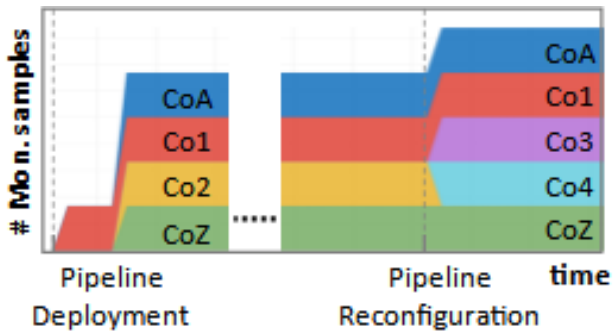


Figure 4.4: Message count after deployment and reconfiguration.

Algorithm	Time(max)	Time(mean)	DC capacity(min)	Infeasible count(mean)
MIP - exact	61.08	24.52	0.24	12
MIP - approx	39.25	25.38	0.26	14
Greedy - Template 1	0.01	0.01	0.44	30
Greedy - Template 2	0.06	0.01	0.33	27

Table 4.1: Algorithm comparison result table.

ML pipeline as in Figure 4.1. Just to mention that WF2 follows a make-before-break approach, i.e., after the creation of new containers, the MLFO waits for the container to be available before continuing with the next steps to avoid losing monitoring samples. Total reconfiguration time was 8.5 sec, most of this time is used waiting for deleted containers to stop gracefully.

Finally, Figure 4.4 8 shows the number of measurements collected in the DB node of the ML pipeline to demonstrate ML pipeline deployment and its reconfiguration

4.4 Algorithm comparison

The objective of this section is to compare the algorithms and set the basis for future algorithms that solve this problem. The experiment consists of an infrastructure shown in Figure 4.5, with 10 metro datacenters, 8 edge datacenters, and one core datacenter. The core datacenter is connected by a

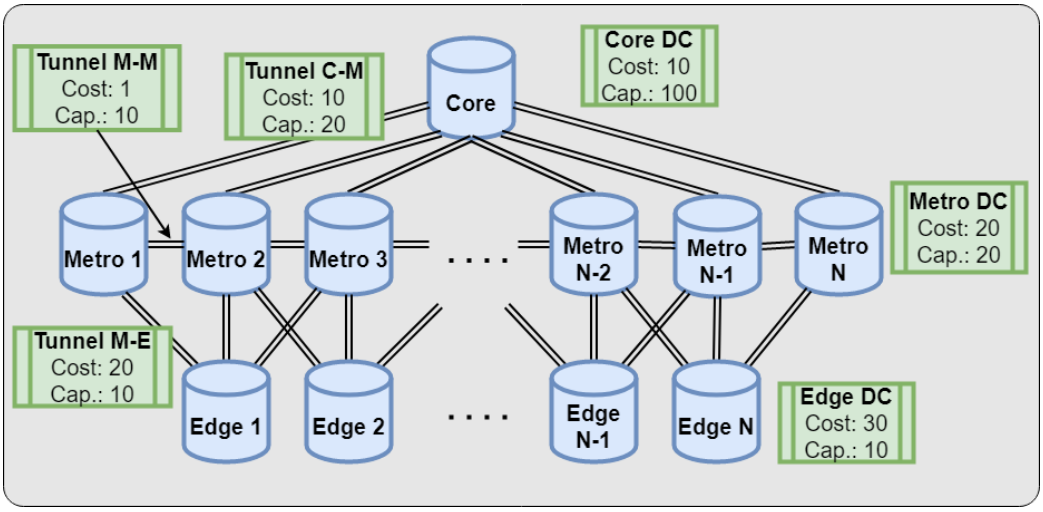


Figure 4.5: Visual representation of the infrastructure used for algorithm comparison, showing the cost and capacity of each datacenter and tunnel.

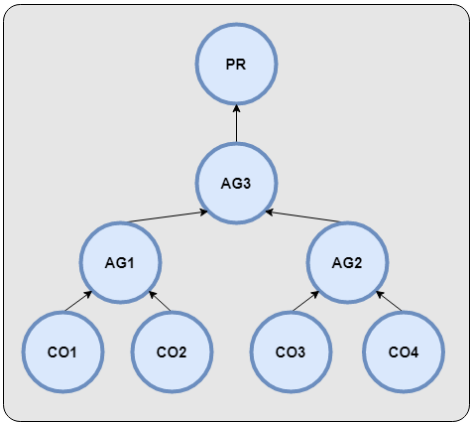
tunnel to each of the metro datacenters, the metro datacenters are connected also to the adjacent datacenters, and the edge datacenters are connected each to 3 metro datacenters. This is a simplistic model of how the datacenters can be distributed in a city.

The experiment consist in deploying 50 applications (one at a time) using each of the algorithms and seeing how the algorithms behave when the infrastructure starts to get saturated. Each application has 4 collectors randomly distributed through the infrastructure datacenters following a uniform distribution.

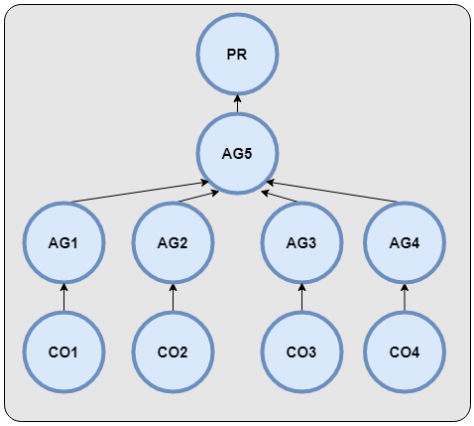
For the ILP CPLEX solver [32] was used, through the Pulp python module [33]. Two versions were considered, the exact version, that searches for the optimal solution, and one with a relative gap tolerance of 0.5, this allows the algorithm to stop as soon as it has found a feasible integer solution proved to be within 50% of optimal.

For the greedy algorithm, two different templates were used, defined in Figure 4.6. The first one uses fewer resources but does not find a solution for every collector distribution.

In Figure 4.7 and Table 4.1 the results of the experiment can be seen. From



(a) Greedy template 1.

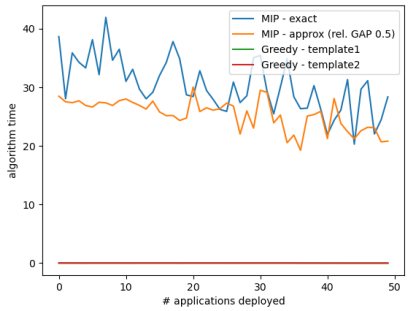


(b) Greedy template 2.

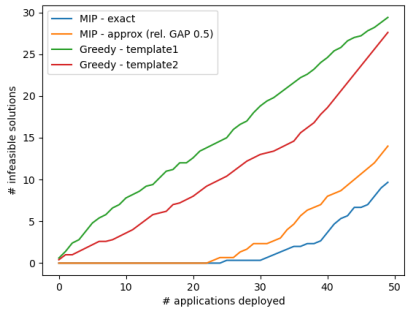
Figure 4.6: The different templates used for the greedy algorithm, showing the disposition of the aggregators.

4.7a we can see that the greedy version is a lot faster than the ILP version, but in Figure 4.7b we can see that it fails to find a feasible solution much frequently. The difference in time taken by the exact and approximate solutions of the MIP is not much (less than 1%), but the exact version for some scenarios takes a lot longer than the approximate (more than 50% more), from the first column of Table 4.1. From Figure 4.7e, it can be seen that the greedy versions start failing to find feasible solutions before the datacenters start to get saturated. Template 2 of greedy is better than template 1 at first, but as it uses more resources it saturates the datacenters first. The exact version is the best performer in finding feasible solutions, and from Figures 4.7d and 4.7c, is the one that can effectively exploit better the resources.

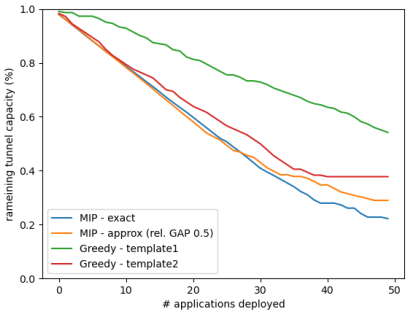
In general, the results from the ILP algorithms take an unacceptable amount of time, but the greedy versions have an unacceptable infeasible count, more work is needed to find an algorithm that uses an acceptable amount of time and infeasible count, candidates are some of the well-known meta-heuristic.



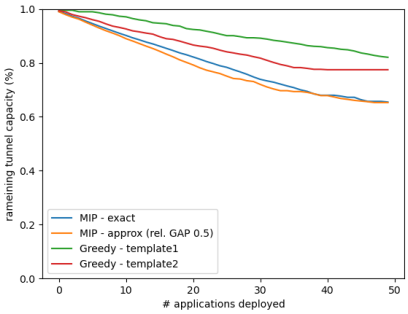
(a) Time taken by algorithm.



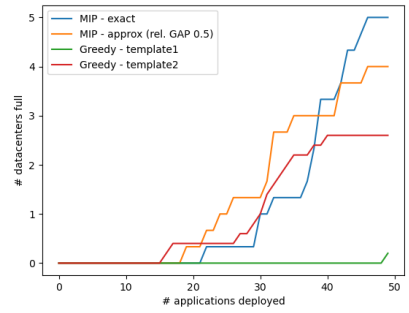
(b) Infeasible solutions count.



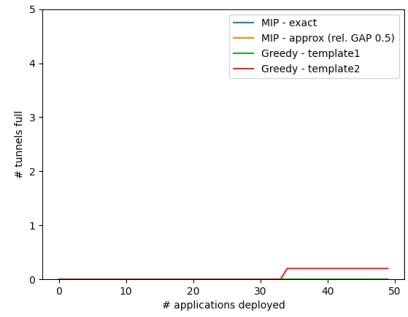
(c) Porcentual datacenter capacity.



(d) Porcentual tunnel capacity.



(e) Full datacenter cumulative count.



(f) Full tunnel cumulative count.

Figure 4.7: Comparison of the different proposed algorithms on the scenario described in Figure 4.5, by deploying fifty applications with four collectors to saturate the infrastructure.

Chapter 5

Planfication and costs

This section will cover a budget estimation for the project.

Role	Annual Salary(€)	Total including SS(€)	Price per hour(€)
Project Manager	40000	52000	29,7
Software Developer	23347	30351,1	17,3
Researcher	33938	44119,4	25,2
Technical writer	25309	32901,1	18,8

Table 5.1: Salary by role estimated using Glassdoor platform.

In Table 5.1 we can see an estimation of the salary for each of the roles considered on the project. Using this estimation the cost of human resources needed for the development of the project will be calculated.

Task ID	Task Name	Hours	Role	Cost(€)
1	**Management**	**200**	**-**	**4523**
1.1	Meetings	20	Manager	594
1.2	Planing	50	Manager	1485
1.3	Writing documentation	80	Writer	1504
1.4	Writing thesis	50	Writer	940
2	**Research**	**240**	**-**	**6048**
2.1	Research networking	70	Researcher	1764
2.2	Research orchestrators	50	Researcher	1260
2.3	Research Kubernetes	50	Researcher	1260
2.3	Research others	70	Researcher	1764
3	**Implementation**	**550**	**-**	**9515**
3.1	Datacenter configuration	100	Developer	1730
3.2	MLFO development	450	Developer	7785
4	**Experimentation**	**210**	**-**	**4897**
4.1	Example application development	50	Developer	865
4.2	Demos execution	100	Researcher	2520
4.3	Analyze results	60	Researcher	1512
-	**Total**	**1200**	**-**	**24983**

Table 5.2: Task management table, with the number of hours dedicated to each task and the cost calculated using Table 5.1

In Table 5.2 a breakdown of the hours invested on the project and an estimation of the cost of the human resources needed to make this investigation is done. The total cost of human resources for the project would be €24983. On top of that, the cost of using the GCO-TESTBED datacenter for the experimentation must be added, plus the use of the lab installations.

Chapter 6

Conclusions

The importance of developing an MLFO was stated, and a reference implementation was developed and its functionality demonstrated.

To develop the application concepts from different areas of computer science where used as optimization, orchestration, networking, machine learning, virtualization, among others.

A NBI was defined for the MLFO, which is practical, well defined, and able to express complex deployments in a declarative way.

The result in terms of deployment and reconfiguration time is good enough, even though they can be further improved, and can be used as a baseline to compare other implementations.

Two algorithms were presented to solve the optimization problem, one that gives an exact solution but takes a lot of time to process and one that gives a fast solution but is not optimal. Some tests were defined to measure how well the algorithms would work on a complex scenario. The developed algorithms can be used as a baseline for future algorithm development.

In general, efforts were made to develop a reference implementation that can give ideas and some measurements to compare against for future implementations or the continuation of this project. A lot of work has to be done to have a production-ready product. Some ideas for future work are given in

the following section.

6.1 Future work and limitations

Some parts of the plan execution could be done in parallel. The current solution executes all the steps sequentially. This improvement could further reduce the time taken on each of the workflows.

Further work could be done on maintaining a consistent state when some step of a deployment fails. In the current solution, only some scenarios were taken into consideration, but not all. This consistency is especially important in this kind of orchestrator because they are used by other applications and not a human that can take actions in case of a failure.

This development assumes that a single VIO and a single SDN controller are used. This can be a problem for very large areas, where the latency for a single controller can be a problem. The solution could also be extended to use multiple controllers distributed geographically to diminish the latency between the controller and the different datacenters.

Right now the solution is not technology agnostic it uses Kubernetes and ODL with VXLAN as tunnels. More work is needed for this solution to become technology agnostic and become a generic framework where different technologies can be used for each component. We selected this initial set of technology because of its wide use and specific properties.

The work that was done on the optimization problem was exploratory, but further work is needed to find an optimization algorithm that scales finding a solution in near real-time and at the same time uses the resources in an almost optimal way.

The security of the MLFO could be improved, encrypting the communication between the Management application and the MLFO and adding a more sophisticated authentication method could be a good start. Security is important in this application especially because of the multitenant characteristic of the scenarios it tries to solve.

Acknowledgments

The research leading to these results has received funding from the Spanish MINECO TWINS project (TEC2017-90097-R) and from ICREA.

Bibliography

[1] Axel Wassington et al. "Implementing a Machine Learning Function Orchestration". In: *2021 European Conference on Optical Communication (ECOC)*. 2021, pp. 1–4. DOI: 10.1109/ECOC52684.2021.9605907.

[2] Danish Rafique and Luis Velasco. "Machine learning for network automation: overview, architecture, and applications [Invited Tutorial]". In: *Journal of Optical Communications and Networking* 10.10 (2018), pp. D126–D143.

[3] L. Velasco et al. "Intent-based networking and its application to optical networks [Invited Tutorial]". In: *Journal of Optical Communications and Networking* 14.1 (2022), A11–A22. DOI: 10.1364/JOCN.438255.

[4] Mohammad Saeid Mahdavinejad et al. "Machine learning for Internet of Things data analysis: A survey". In: *Digital Communications and Networks* 4.3 (2018), pp. 161–175.

[5] Shagufta Henna. "Demonstration of machine learning function orchestrator (MLFO) via reference implementations". In: *ITU AI/ML in 5G Challenge* (2020).

[6] Luis Velasco et al. "End-to-End Intent-Based Networking". In: *IEEE Communications Magazine* 59.10 (2021), pp. 106–112. DOI: 10.1109/MCOM.101.2100141.

[7] NM Mosharaf Kabir Chowdhury and Raouf Boutaba. "A survey of network virtualization". In: *Computer Networks* 54.5 (2010), pp. 862–876.

[8] Rajdeep Dua, A Reddy Raja, and Dharmesh Kakadia. "Virtualization vs containerization to support paas". In: *2014 IEEE International Conference on Cloud Engineering*. IEEE. 2014, pp. 610–614.

[9] *Kubernetes Documentation*. Tech. rep. Version 1.19. The Linux Foundation ®, 2020. URL: https://v1-19.docs.kubernetes.io/docs/home/.

[10] Mallik Mahalingam et al. "Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks." In: *RFC* 7348 (2014), pp. 1–22.

[11] Jianguo Ding, Ilangko Balasingham, and Pascal Bouvry. "Management of overlay networks: A survey". In: *2009 Third International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*. IEEE. 2009, pp. 249–255.

[12] *SDN Architecture Overview*. Tech. rep. Version 1.0. Open Networking Foundation, 2013. URL: https://opennetworking.org/wp-content/uploads/2013/02/SDN-architecture-overview-1.0.pdf.

[13] *OpenFlow Switch Specification*. Tech. rep. Version 1.3.0. Open Networking Foundation, 2012. URL: https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf.

[14] Nathan F Saraiva de Sousa et al. "Network service orchestration: A survey". In: *arXiv e-prints* (2018), arXiv–1803.

[15] *TOSCA Simple Profile for Network Functions Virtualization (NFV)*. Tech. rep. Version 1.0. TOSCA, OASIS, 2015.

[16] Martin Bjorklund et al. "YANG-a data modeling language for the network configuration protocol (NETCONF)". In: (2010).

[17] A Reid et al. "OSM Scope, Functionality, Operation and Integration Guidelines". In: *ETSI, White Paper* (2019).

[18] *Open Network Automation Platform*. Tech. rep. Version 9.0.0 (Istambul). The Linux Foundation ®, 2019. URL: https://docs.onap.org/en/istanbul/.

[19] G. Xilouris et al. "T-NOVA: A marketplace for virtualized network functions". In: *2014 European Conference on Networks and Communications (EuCNC)*. 2014, pp. 1–5. DOI: 10.1109/EuCNC.2014.6882687.

[20] *Cloudify documentation*. Tech. rep. Version 6.1.0. 2021. URL: https://docs.cloudify.co/6.1.0/.

[21] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998. Chap. 1.

[22] J. A. Nelder and R. Mead. "A Simplex Method for Function Minimization". In: *The Computer Journal* 7.4 (Jan. 1965), pp. 308–313. ISSN: 0010-4620. DOI: 10.1093/comjnl/7.4.308. eprint: https://academic.oup.com/comjnl/article-pdf/7/4/308/1013182/7-4-308.pdf. URL: https://doi.org/10.1093/comjnl/7.4.308.

[23] Paul E. Black. *greedy algorithm*. Ed. by Dictionary of Algorithms and Data Structures (NIST). Accessed: 8 January 2022. 2005. URL: https://www.nist.gov/dads/HTML/greedyalgo.html.

[24] *The Container Networking Interface Specification*. Tech. rep. The Linux Foundation ®. URL: https://www.cni.dev/docs/.

[25] Ben Pfaff et al. "The design and implementation of open vswitch". In: *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 2015, pp. 117–130.

[26] *OpenDaylight Documentation*. Tech. rep. OpenDaylight Project, 2020. URL: https://docs.opendaylight.org/en/stable-silicon/.

[27] Björn Rabenstein and Julius Volz. "Prometheus: A Next-Generation Monitoring System (Talk)". In: Dublin: USENIX Association, May 2015.

[28] Grafana Labs. *Grafana Documentation*. 2018. URL: https://grafana.com/docs/.

[29] Hui Zang et al. "Dynamic lightpath establishment in wavelength routed WDM networks". In: *IEEE Communications Magazine* 39.9 (2001), pp. 100–108. DOI: 10.1109/35.948897.

[30] Hui Zang, Jason P Jue, Biswanath Mukherjee, et al. "A review of routing and wavelength assignment approaches for wavelength-routed optical WDM networks". In: *Optical networks magazine* 1.1 (2000), pp. 47–60.

[31] Luis Velasco et al. "In-operation network planning". In: *IEEE Communications Magazine* 52.1 (2014), pp. 52–60. DOI: 10.1109/MCOM.2014.6710064.

[32] CPLEX User's Manual. "Ibm ilog cplex optimization studio". In: *Version* 12 (1987), pp. 1987–2018.

[33] Stuart Mitchell et al. *Optimization with PuLP*. Tech. rep. URL: https://coin-or.github.io/pulp/.

Appendices

Appendix A

Message details

A.1 Cofigure services

```
{
  "repositoryconfigs":[
    {
      "id": "regcred",
      "cred": "XXX",
      "type": "dockerconfig"
    }
  ],
  "subnet-generator":{
    "network":"192.168.30.0/24",
    "subnets-mask":"29"
  },
  "vlan-range":{
    "from": 1000,
    "to": 2000
  },
  "tunnels":[
    {
      "id": 200,
      "connection":["metro2","core"]
    },
    {
      "id": 201,
      "connection":["metro1","core"]
```

```
    },
    .
    .
    .
  ]
}
```

A.2 Create template message

```
{
   "appname": "mlpn",
   "repositoryconfig": "regcred",
   "containers-models": [
       {
           "name": "pr",
           "image": "gco/mlf-collector:2.1",
           "readiness": {
                "path": "/ping",
                "port": 80
            },
           "endpoint": {
               "host": "main.k8s.gco",
               "path": "/{{ appname }}/V{{ appversion }}(/|$)
   (.*)",
               "targetport": 80
           },
           "configs": [
               {
                   "name": "CONFIGS_SERVER_ID",
                   "value": "processor"
               },
            .
            .
            .
       },
       {
           "name": "ag",
           "image": "gco/mlf-collector:2.1",
           "dynamic-config-files":[{
               "filename": "servers.yaml",
               "path": "/etc/configs",
               "data": [{
```

```
                    "name": "push-url",
                    "value": "http://{% set subnetsids =
container.subnets | map(attribute='id')  | list %}{{
containers | selectattr('template_function','in',
container.data.pushfunction) | map(attribute='subnets') |
flatten | selectattr('id','in',subnetsids ) | map(
attribute='ip') | list | first }}/reg"
                }]
            }],
            .
            .
            .
        },
        {
            "name": "co",
            "image": "gco/mlf-collector:2.1",
            .
            .
            .
        }
    ],
     "template":[
        {
            "template_function": "pr",
            "model_name": "pr",
            "type":"unique"
        },
        {
            "template_function": "ag2",
            "data":{
                "pushfunction": ["pr"]
            },
            "model_name": "ag",
            "type":"auto",
            "connections": ["pr"],
            "max": 5
        },
        {
            "template_function": "ag1",
            "data":{
                "pushfunction": ["pr", "ag2"]
            },
            "model_name": "ag",
            "type":"auto",
            "connections": ["pr", "ag2"],
```

```
            "max": 10
        },
        {
            "template_function": "co",
            "data":{
                "pushfunction": ["ag1"]
            },
            "model_name": "co",
            "type":"custom",
            "connections": ["ag1"]
        }

    ]
}
```

A.3 Deploy application message

```
{
  "dataceter_mappings":[
    {
      "template_function":"co",
      "data":{
        "name": "CoA"
      },
      "datacenterid": "metro2"
    },
    {
      "template_function":"co",
      "data":{
        "name": "Co1"
      },
      "datacenterid": "metro2"
    },
    .
    .
    .
  ]
}
```