Automation in Multi-Domain Software-Defined Networking: Overview and Use Cases

Auteur : Collin, William
Promoteur(s) : Donnet, Benoît; 6116
Faculté : Faculté des Sciences appliquées
Diplôme : Master en sciences informatiques, à finalité spécialisée en "computer systems security"
Année académique : 2020-2021
URI/URL : http://hdl.handle.net/2268.2/11665

University of Liège - Faculy of Applied Sciences

Master Thesis

Automation in Multi-Domain Software-Defined Networking: Overview & Use Cases

A thesis submitted in fulfilment of the requirements for the degree of Master in Computer Science

Author:
William Collin

Supervisor:
Benoit Donnet

Advisors:
Hugues De Pra
Wouter De Prins

Academic year 2020-2021

Acknowledgments

Throughout the writing of this dissertation, I have received a great deal of support and assistance.

I would first like to thank people from Cisco that made this traineeship possible. Hugues de Pra for his supervision and support who gave me insights on the business side of the IT industry, and made me felt like a member of the team despite working remotely all the year. Wouter de Prins for his invaluable mentorship that influenced the career path I've chosen, and for always encouraging me to discover and learn new technologies.

Then, I would also like to thank my supervisor Benoit Donnet for his insightful feedback in writing my thesis, and more generally for his guidance and teaching during my studies.

Lastly, I would also like to thank my friends, especially my partner in crime Thomas Carlisi for the smooth collaboration and the fun times during all those years.

Abstract

Software-defined networking (SDN) is a solution designed to make networks more flexible and easier to manage. Cisco has designed the Application-Centric Infrastructure (ACI) for data center networks, and Software-Defined Access (SDA) for campus networks. In addition, Cisco is currently developing integrations between its Software-Defined Networks. This project aims at building a use case for the integration between ACI and SDA to show how end-to-end segmentation can be performed between the two domains, and to build a solution to automate the deployment of configuration to ACI. The latter was extended and also demonstrates how Infrastructure-as-Code and DevOps can be used to manage IT infrastructures that include the container orchestrator Kubernetes, and VMware vSphere.

Contents

List of Figures

Chapter 1

Introduction

Over the last ten years, Software-Defined Networks (SDN) and programmability have introduced a different approach to building and managing computer networks. SDN is an approach aiming to make networks more flexible and agile. The Open Networking foundation originally defined SDN as a solution that decouples the control plane and the data plane, enabling the control plane to become directly programmable[15]. As of today, SDN is globally recognized as an architecture that opens the network to applications. This integrates the two following aspects: enabling applications to program the network to accelerate its deployment; providing a better visibility over the network. There is therefore no single SDN with a single solution or protocol. Every vendor defines their SDN according to their own interests[11].

An SDN typically involves a central control point for managing the network called a controller, through which, all network operations are performed either using a GUI, or an interface that it exposes. This interface (Northbound API) is commonly a REST API and enables programmability of the network and integration with other systems. Likewise, the controller communicates with the network devices with an interface called the Southbound API using protocols such as OpenFlow, RESTCONF, OpFlex, just to name a few.

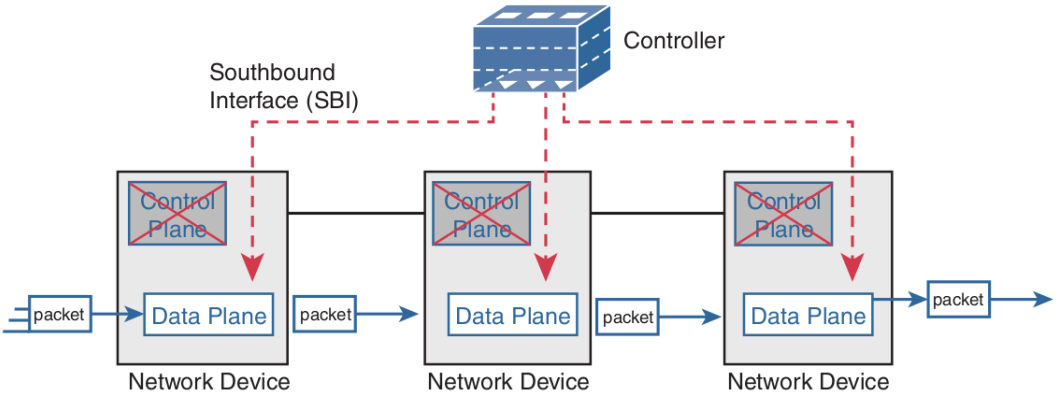

Figure 1.1: Controller Communicating with Network Devices with the Southbound API using the OpenFlow protocol. By W. Odom, 2020, CCNA 200-301 Volume 2, p. 364[42]

The degree of control and the type of control performed by SDN controllers vary widely. There are 2 types of control: imperative and declarative control. Indeed, in early approaches to software-defined networking, it was a question of programming the data plane of the switches via protocols such as OpenFlow and having the controller conduct all the control plane operations, as it is shown on Figure 1.1. In such a system, the controller explicitly tells the switches how to handle network traffic by sending them down the exact commands or instructions to make the change. The protocol Openflow uses an imperative approach. This model endorsed by the Open Networking Foundation allows to completely decouple the data

plane and the control plane.

Newer SDN solutions like the Application-Centric Infrastructure (ACI) are based on declarative control and do not completely strip intelligence out of the devices. In this model, policies are sent from a controller to networking devices, the latter responsible to implement them. Recently, Cisco has developed several SDN solutions, including Software-Defined Access designed for campus networks, and Application-Centric Infrastructure for the data center environment. Both solutions are part of the Multidomain strategy that Cisco has recently been pushing, which consists of integrating multiple SDNs.

By providing an open Northbound interface, SDN provides a different way to operate networks. It allows network engineers to program the network using scripts or automation tools such as Terraform or Ansible. More generally, providing a management platform with an API also applies to other areas of the IT infrastructure such as the configuration of bare metal servers, virtualized environments, public clouds, storage, etc. This concept of managing the IT infrastructure with code refers to as "Infrastructure-as-Code" (IaC) and implies that it should be treated the same way as any other code, for instance, be uploaded to a version control system (VCS) acting as a single source of truth.

IaC is also one of the key enablers of the DevOps movement, a set of ideas, practices, and tools that increases an organization's ability to deliver applications and services at high velocity. An important concept of DevOps is the CI/CD pipelines commonly integrated within the VCS software, it allows developers to automate any task upon some event such as the push of new code into a Git repository. As a result, IT operations can also be carried out directly from the VCS leveraging the pipelines. Gitlab is an example of VCS providing CI/CD pipeline capabilities.

In today's highly competitive environment where applications are the key element that supports the business[66]. Businesses need to quickly deploy and run new applications within their networks and react quickly to the ever-changing needs of their users. The problem today is that there is a gap between the world of infrastructure and the world of applications: it can take several weeks between the arrival of a new application and its availability to users, particularly because of the time required to provision networks, storage, and server resources[66]. In traditional networks, network operations involve many different devices to be configured manually, which is time-consuming, repetitive, and vulnerable to error. This growing need for simplifying network operations using automation and thus accelerating the deployment of applications is the main driver behind the adoption of SDN, a different approach to building and managing computer networks. This explains why Software-Defined Networks and IaC are two key elements of the infrastructure of tomorrow.

This work is focused on the integration between ACI and SDA, and on programmability. The main objective is to create and implement a use case for this integration. In particular, this integration provides seamless and consistent identity and endpoint mapping between the campus and data center networks, allowing administrators to create cross-domain policies. In the meantime, as Cisco is putting a lot of effort into the Infrastructure-as-Code territory, a second objective is to build a programmability demonstration to show how ACI can be programmed following the DevOps principles.

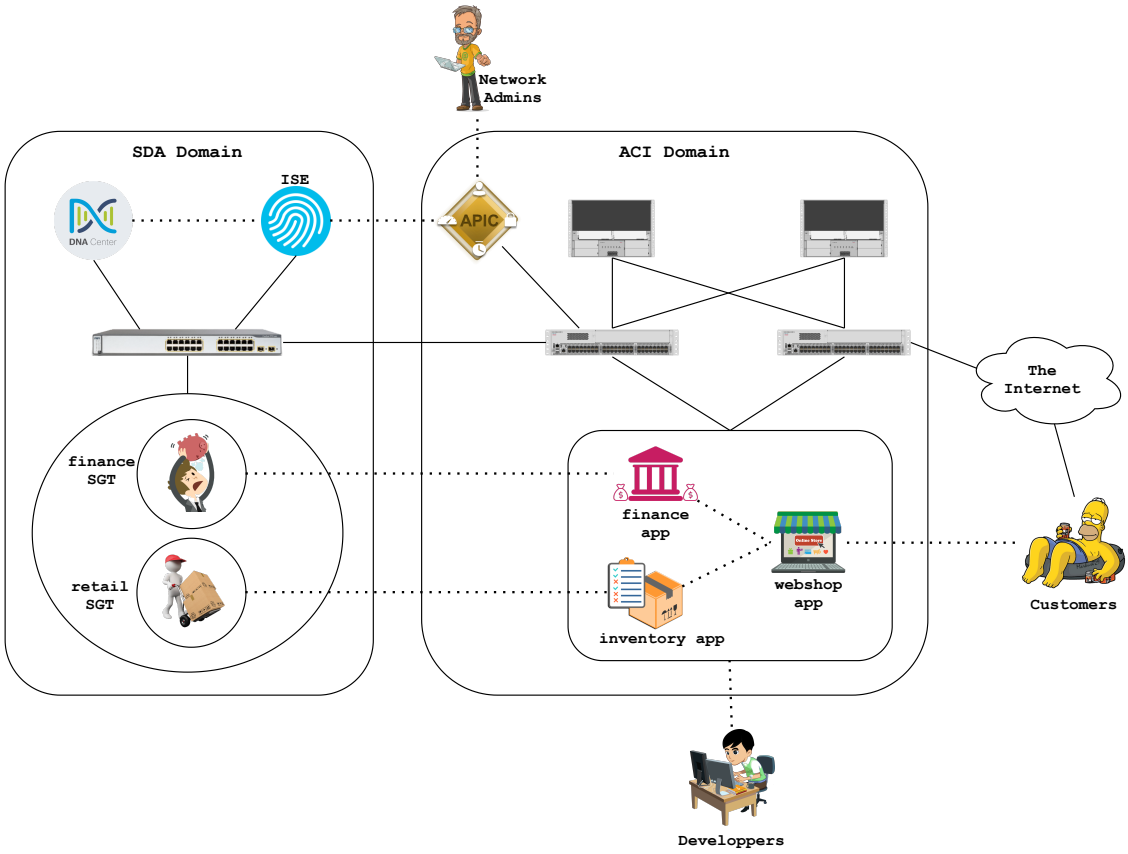

Figure 1.2: use case Overview

The use case takes place in the beer industry with a company named Cisco's Beer Factory. It involves three applications developed with a "microservice" architecture, and three types of users: the finance and retail employees sitting in the campus network, and the customers from the Internet. Each of them has access to a particular application running in the data center. A finance application for the finance employees, an inventory application destined to retailers, and a webshop to buy beers from the company. This use case allows us to show how end-to-end segmentation between the ACI and the SDA domain can be carried out. The big picture is presented in Figure 1.2

Developing the applications, deploying them, and configuring the network around them brings us to the second objective of this internship. Efforts were made to perform all IT operations with IaC from the DevOps platform Gitlab by leveraging CI/CD pipelines. It includes the configuration of ACI, the deployment of the applications to Kubernetes, and the provisioning and configuration of virtual machines in a virtualized environment such as vSphere. Initially, it was planned to perform the ACI-SDA integration with the applications running in a Kubernetes cluster, but sadly, the integration does not allow it. This explains why everything had to be migrated to virtual machines.

The structure of this thesis is the following. It starts by providing general information about ACI in Chapter 2. Then, Chapter 3 explains the basics of Kubernetes and how container networking is managed by ACI. Afterwards, the project objectives and the deliverables are summarized in Chapter 4. Next, Chapter 5 and 6 explains what is Infrastructure-as-Code, and how it was carried out in the lab. Last but not least, Chapter 7 summarizes the main features of SDA. Finally, How the ACI-SDA integration works and how it is configured for the use case is explained in Chapter 8.

Chapter 2

Application Centric Infrastructure

2.1 Application-Centric Infrastructure: Overview

On July 31, 2014, Cisco released its software-defined solution for data-center networks to the market: Application Centric Infrastructure [65]. An overview of ACI is given in this section.

When reimagining networking for the data center, the designers of ACI focused on the applications running on the data center and their requirements[43]. As a consequence, they have built networking concepts around application architectures, hence the name 'Application Centric Infrastructure'. ACI is illustrated in Figure 2.1.

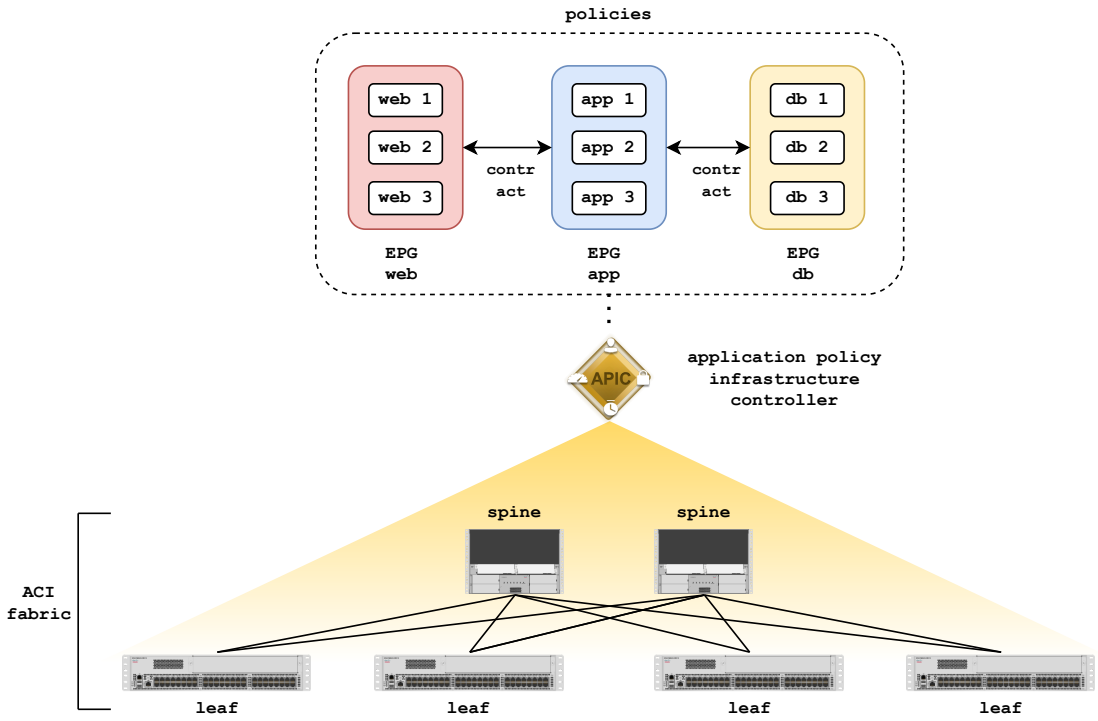

Figure 2.1: ACI Overview

ACI is based on a policy model that decouples the network design from the physical infrastructure and focuses on the needs (networking, security, and services) of the applications, and allows application requirements to be defined as policies. In ACI, every aspect of network configuration is defined by policies describing how the system should behave.

For example, assume that we have a typical 3-tier web application consisting of a web frontend, a backend, and a database. In ACI, instead of worrying about the details of how the endpoints find each other (VXLAN, routing, etc.), a network engineer will define groups (called endpoint groups (EPG) based on application attributes. Therefore, he will create three endpoint groups (Web, App, and DB for instance). Once this is done, he will define policies called contracts, which dictates the communication rules between the endpoints (you can think of a contract as a traditional access-list). For instance, the EPG Web is allowed to communicate with the EPG App over HTTP.

As we saw in the example above, ACI abstracts traditional network constructs (such as VLANs, VRFs, IP subnets, and many more) and introduces new constructs including application profiles, endpoint groups, contracts, objects associated with external connectivity such as the L3out, as well as service graphs for the integration of L4-L7 services like load balancers, firewalls and many more[47]. For more details about policy objects, see Section 2.4.

The two main components of ACI are the Nexus L3 switches wired in a leaf-spine topology for the physical switching infrastructure, and the Application Policy Infrastructure Controller (APIC) for centralized network management and real-time network health monitoring. The APIC exposes the functionalities of the ACI fabric (an ACI fabric is the set of leaf and spines nodes under the control of the same APIC domain) through a REST API supporting JSON and XML payloads, and allows integration with cloud software (e.g Google Cloud Platform, AWS), virtualization software (e.g VMware vCenter) and automation software (e.g Terraform, Ansible). Engineers can also communicate with the APIC through a web-based GUI offering a single pane of glass of the whole ACI fabric (see Figure 2.2) or via custom scripts.

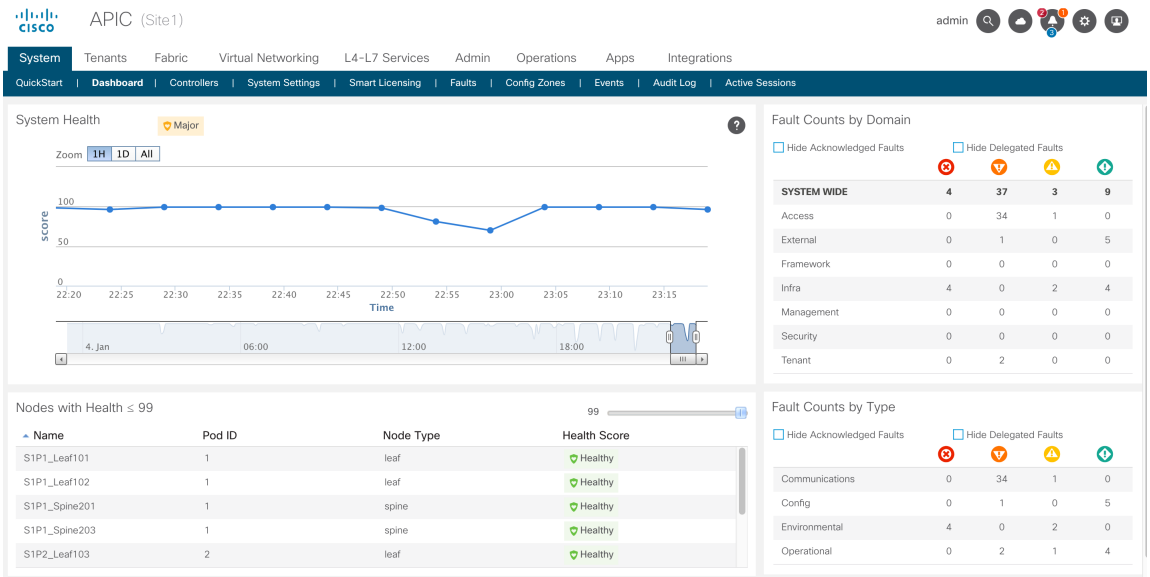

Figure 2.2: APIC Graphical User Interface

ACI is also hypervisor agnostic, meaning that policies can be applied to any workload (bare-metal servers, virtual machines, containers).

2.2 Declarative Control with OpFlex in ACI

This section is dedicated to the southbound protocol OpFlex used for communications between the APIC and the Nexus switches in the ACI fabric. It describes the OpFlex protocol

and points out the benefits of OpFlex and why Cisco has chosen OpFlex for ACI.

2.2.1 Differences Between Declarative Control and Imperative Control

Before delving into the details of OpFlex, it is preferable to explain the differences between declarative and imperative control applied to networking.

In an imperative control system, the controller receives requests from the northbound API and instructs the network devices on how they have to be configured. In other words, the controller explicitly tells the switches how to handle network traffic by sending them down the exact commands or instructions to make the change. The protocol Openflow uses an imperative approach.

Declarative control uses concepts of promise theory, a model based on intelligent objects that take care of the configuration state changes initiated by a central controller. These intelligent objects are also responsible for informing the central system of anomalies and exceptions. In declarative control, the switches are asked by the controller to reach a desired state but are not being told precisely how to do so. One way to explain how declarative control functions outside of the networking world is to look at how air traffic controllers operate. They tell pilots to land on particular runways but do not explains how to reach them. The job of flying and landing the plane is left to the pilot[63].

To sum up, imperative control focuses on the 'how' whereas declarative control focuses on the 'what'. As we will in section 5.1, the concepts of declarativeness and imperativeness can also be applied to automation tools.

2.2.2 The OpFlex Control Protocol

In its broadest sense, OpFlex is a policy-driven system used to manage a large set of devices[72]. It works by abstracting policies and relies on intelligent, autonomous devices to interpret them, hence the declarative control.

As shown in Figure 2.3, this system is composed of several logical units:

- the policy repository.
- the endpoint registry.
- the observer.
- the policy elements.

As its name suggests, the policy repository is a logically centralized entity containing all the policy definitions created by users of the system through an interface (API, graphical user interface, etc.). The policies are stored in a tree data structure called the management information tree (MIT) or management information model. For more information about the MIT in ACI, refer to section 2.4.1.

The endpoint registry stores information (identity, location, etc.) about the endpoints connected to the system. the policy elements register the endpoints to the endpoint registry. In ACI, the endpoint registry is implemented as a distributed database.

The Observer is in charge of providing analytics of the system. It contains the state and performance data of the managed devices.

The policy elements are the units enforcing the policies. The MIT in the policy repository has to be kept in sync with the policy elements, so the policy elements are automatically notified when changes to the policy model occur. Moreover, only the managed objects relevant for the policy elements are synchronized, therefore, only a subset of the logical model is stored. In order to apply the policies to the hardware, the policies are rendered into a concrete model that is analogous to compiled software; it is the form of the model that the switch operating system (NX-OS in the Nexus switches) can execute[8]. This approach is called a model-driven framework.

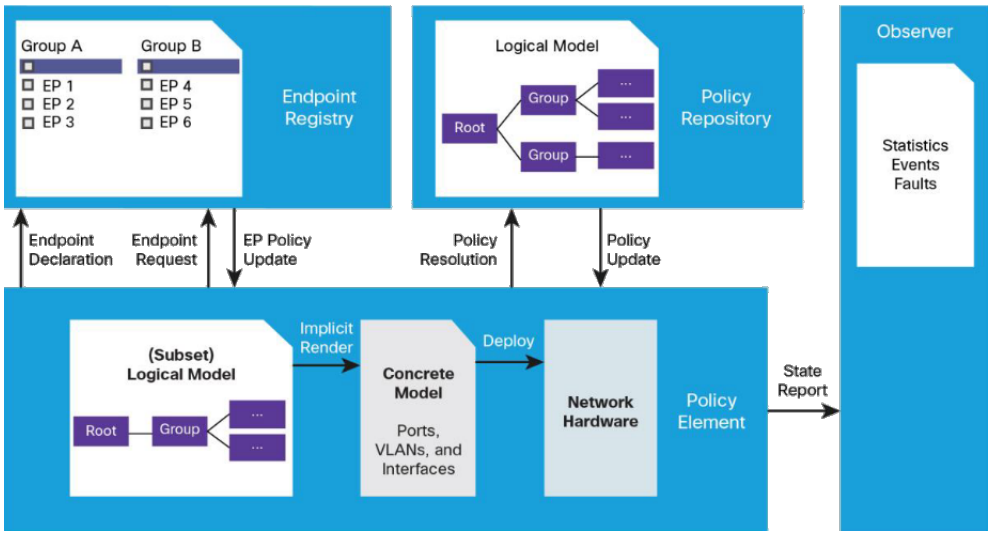

Figure 2.3: OpFlex Logical Model. By Cisco Systems, 2014, OpFlex: An Open Policy Protocol White Paper[63]

Messages in an OpFlex system are transferred using remote procedure calls (RPC) and the data is encoded in JSON or XML.

OpFlex was co-authored by the big players in the IT industry (Cisco, Microsoft, IBM, Citrix, and Sungard Availability Service) and has been designed for interoperability between systems from multiple vendors. Moreover, OpFlex was submitted to the IETF standardization process. To illustrate the interoperability of OpFlex, we can look at the integration between ACI and the container orchestrator Kubernetes. ACI can push policies to an OpFlex agent running in a container inside Kubernetes, which in turn renders the network policies to OpenFlow rules to program the virtual switches (Open vSwitch) of Kubernetes also running inside containers.

2.2.3 Why OpFlex?

Declarative control systems like OpFlex offer a number of advantages over imperative control systems.

In declarative control, new policies can no longer be pushed to the network devices if the controller goes down, however, the data forwarding can continue to happen since the APIC is removed from the data path. Indeed, after device configurations are completed, communication between the APIC and the network devices may be only required only in the event of policy updates or fault scenarios. Therefore, the APIC controller is no longer a catastrophic single point of failure.

The controller in Openflow networks can also be a performance bottleneck as the network grows and becomes more complex. The complexity is distributed among the network devices in ACI, offering better scalability.

Policies also adds a layer abstraction over the network, making it easier to configure since the operators have to deal less with low-level constructs, and thus hides the details from people like application developers that can reuse the groups and contracts that the network engineer has define. Therefore, ACI with its abstract policy model introduces a new way of interacting with the application teams that makes much more sense to them and allows them to work closer together[45].

One of the main reasons behind the introduction of OpFlex as a southbound protocol by Cisco is also economical. Cisco has spent decades and invested billions of dollars pushing intelligence into its network devices, they were simply not going to throw years of work in the trash and adopt a model in which the intelligence is stripped from the network devices.

2.3 ACI Physical Design

For many years, engineers have used a three-tier architecture for data center designs, consisting of a core, aggregation, and access layer. But the leaf-spine has become the standard in today's deployment of data center networks, overcoming some of the limitations of traditional three-tier architectures.

This section is focused on the physical architecture of the application-centric infrastructure, discusses why the topology of the spine-leaf was picked, and briefly outlines the ways ACI can be deployed.

2.3.1 About The Application Policy Infrastructure Controller

The APIC is implemented as a replicated synchronized clustered controller. The recommended minimum configuration for an ACI fabric is a cluster of three replicated controllers[8]. The policy repository is actually a distributed database that is divided into several parts called shards. Each shard is assigned to a master APIC and is broken into three replicas that are distributed among the controllers. Changes of policies in a Shard are performed by the master APIC of that shard, and thus allowing to balance the workload among the controllers. If one of the APIC goes down, the remaining APICs will negotiate who will be the master of the shards that the down APIC was in charge of. Having such system allows for better resiliency in case of failure scenarios.

2.3.2 Spine-Leaf Topology

The spine-leaf architecture is adapted from the clos network named after Charles clos in 1952[67]. As its name implies there are spine and leaf switches in a spine-leaf architecture. In this design:

- Each leaf switch is connected to every spine switch.
- The leaf switches are not connected to each other.
- The spine switches are not connected to each other.
- Endpoints connects only to the leaf switches.

In ACI, a controller is considered as an endpoint and is thus connected to a leaf switch. Endpoints can be bare-metal servers, hypervisors, or any kind of networking devices such as routers or firewalls. In particular, the part of the lab we are working on consists of three ESXi hypervisors installed on Hyperflex servers connected to two Fabric Interconnects, themselves connected to the ACI leaves. This is shown on Figure 2.4.

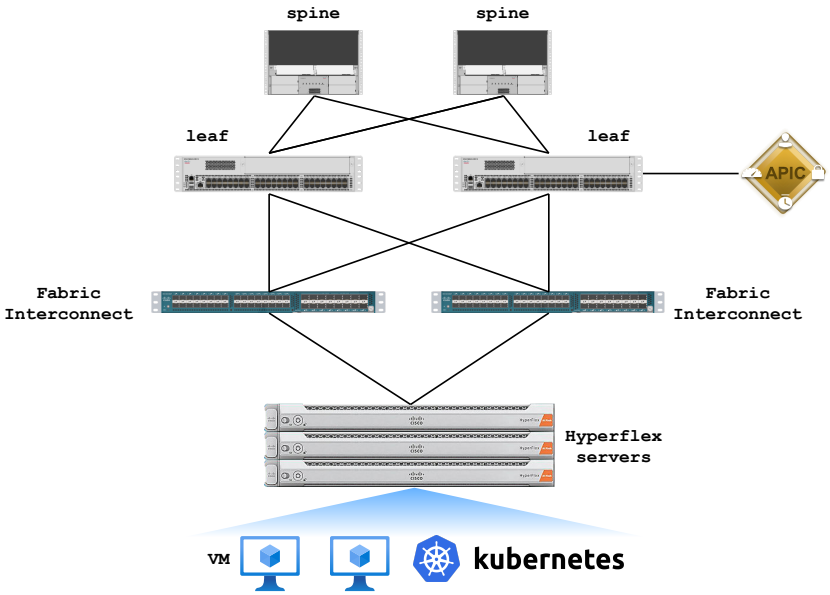

Figure 2.4: ACI Lab Topology

To understand why this topology was selected for the physical design of ACI, we need to look at traffic patterns in data center networks.

East-West traffic in Data Center Networks

Traffic patterns in the data center can be divided into two categories: north-south (or vertical) and east-west traffic (or lateral).

North-south traffic is traffic entering (resp. leaving) the data center from (resp. to) the outside world, whereas east-west traffic indicates traffic flows among endpoints within the data center.

East-west traffic dominates in data center networks, this is highlighted in a study from Samar Raza Talpu[64] and also by Cisco in the Global Cloud Index 2015-2020[55]. This increase in east-west traffic is due to multiple factors.

First of all, the way applications are developed has changed. For a long time, the typical pattern was the monolithic application. This is a single large application consisting of multiple sub-systems or capabilities. With monolith application, the traffic is generally north-south[24]. Nowadays, applications tend to be broken into multiple components that are called services, allowing each service to be developed and deployed independently of the others. This is the microservice pattern. If a web application is deployed in a data center and is built around the microservice architecture, a single query to the application may require multiples services to communicate with each other, and thus generating lateral traffic flows since the services may probably reside in different endpoints.

Then some applications like cloud storage requiring data replication, or distributed applications (e.g Hadoop, Apache Spark, etc.) are also driving the increase in traffic in data center networks.

Next, commonly used techniques in cloud environments are also drawing a large number of network resources. One of them is live VM migration, it allows the transfer of running virtual machines across multiple physical machines to balance the workload.[21]

Advantages of the Spine-Leaf Topology For East-West Traffic

A spine-leaf topology best manages lateral traffic flows and provides easier scalability.

This architecture shortens the communication paths between two endpoints. Indeed, the traffic originated by an endpoint is always the same number of hops from its destination (unless the destination endpoint is located on the same leaf) since the leaves are fully meshed with the spines. Therefore, latency in east-west traffic is lower and predictable[16]. While in a three-tier architecture, the number of hops can vary depending on the location of the endpoints. In some cases, the data flows may only have to move up to the aggregation layer before reaching destination. In others, traffic may have to reach the core routers. It creates a discrepancy in latency, which is problematic for time-sensitive applications.

The spine-leaf design allows for easy network scalability as expanding the network is straightforward. An additional spine switch can be added along with uplinks to every leaf switch, and thus creates a new network path, which is used to load-balance the traffic. And bandwidth scales linearly with the addition of spine switches in a spine-leaf topology[54].

2.4 ACI Policy Model

As we have learned in the previous sections, the entire ACI fabric is managed by policies recorded in the management information tree.

This section details the structure of the policy model in ACI and presents common policy objects used to configure the ACI network.

2.4.1 Management Information Tree

Physical and logical components that comprise the ACI fabric are recorded in a hierarchical structure called the management information tree.

Each node of the MIT is a managed object (MO) that is an abstraction of a fabric resource. The policy model is an object-oriented data structure comprised of managed objects that are instances of classes. An MO can represent a physical object such as a switch, or a logical object like an endpoint group. Each MO is identified by a distinguished name (DN) and contain properties (child/parent relationships, object attributes, etc.) making up the object. On a practical note, users can browse managed objects via a web-based GUI called Visore.

The northbound REST API of the APIC is actually an interface to the MIT allowing CRUD (create, read, update and delete) operations to managed objects.

MOs can be placed in different categories. For instance, all MOs related to policies configured under a tenant (contracts, endpoint groups, etc.) are direct or indirect children of a Tenant object. Similarly, access policies and fabric policies are children of the Infra object. A logical view of the management information tree is provided in figure 2.5

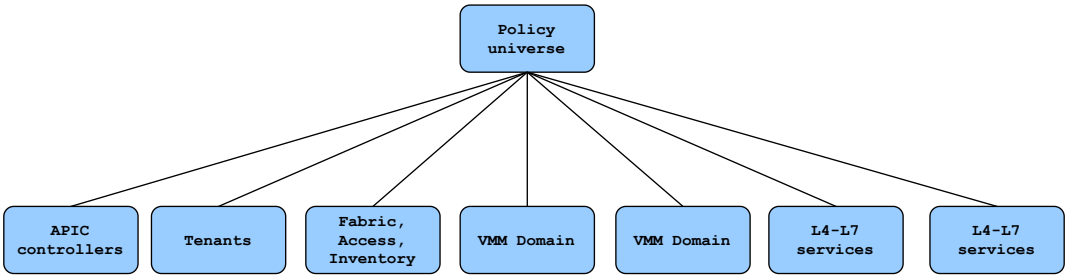

Figure 2.5: logical view of the Management Information Tree

Tenant, Access, and L4-7 service policies are discussed below. The goal of the following subsections is to give some insights on how policies are employed and fit together for some simple use cases.

2.4.2 Tenant Policies

A tenant is a logical container in which network engineers will define the application requirements such as application profiles, endpoint groups, etc. Figure 2.6 depicts the Tenant object as well as its relations with other objects commonly used to write application policies. In this figure, solid lines indicate that objects contain the ones below, dotted lines indicate a relationship. For instance, we can infer from the graph that a tenant may contain multiple application profiles due to the one-to-many cardinality relationship.

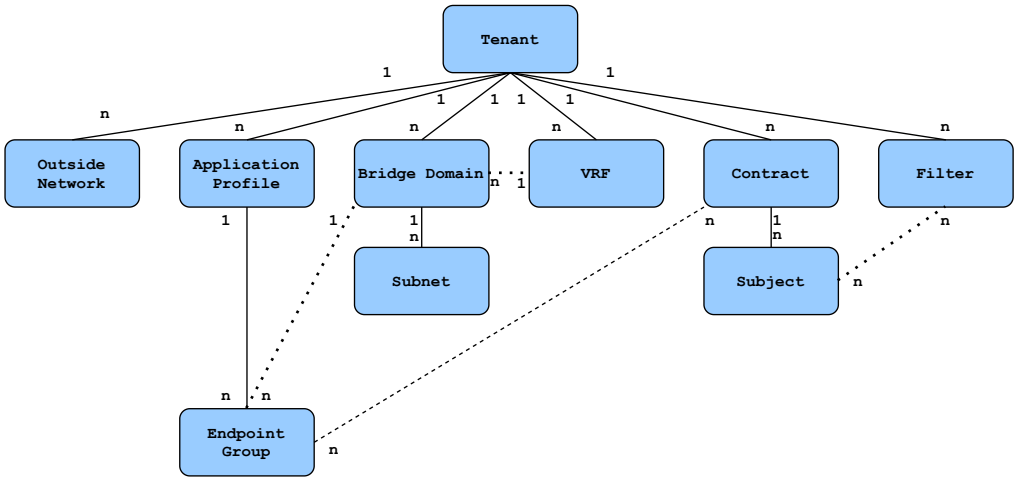

Figure 2.6: Tenant Managed Object and Some of its Children

From a policy perspective, a tenant is a unit of isolation, it allows network infrastructure administration to be segregated out. The notion of a tenant is quite flexible, it can represent business units inside an organization or customers for a cloud provider. For instance, at Cisco, engineers sometimes have to show demos running in a lab to their customers. They build a tenant per demo and I make sure to perform experiments in my tenant so that I do not mess up with what they have built. Policies can be shared among tenants if they are defined in the *Common* tenant.

The Big Picture

Before digging into the details of some of these objects, let me illustrate 2.6 with an example. Figure 2.7 depicts the policy objects and their relationships with each other used to configure the network for a three-tier application in ACI.

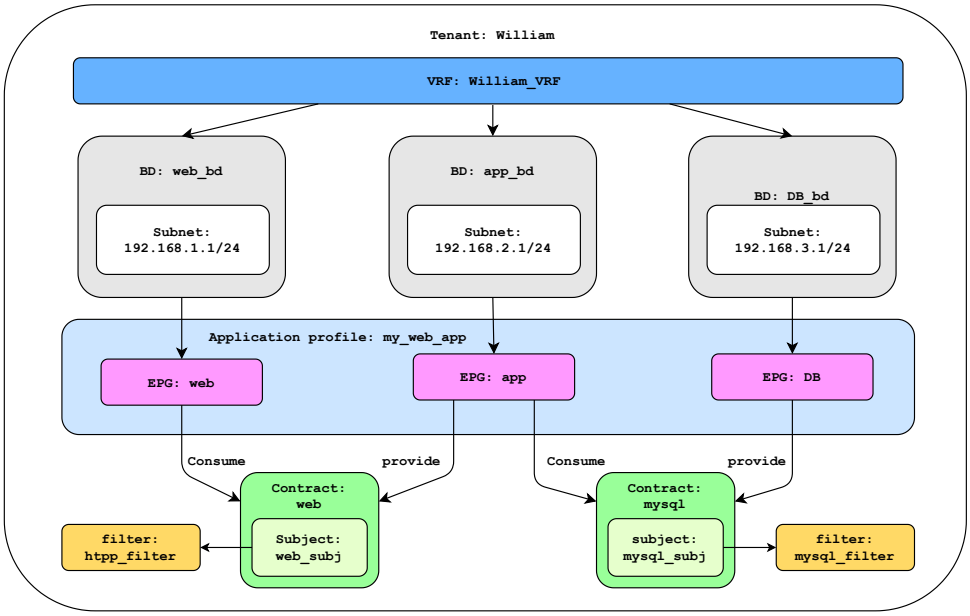

Figure 2.7: Tenant Policies for a Basic Web Application

Virtual Routing Forwarding

Tenants also provide Layer 3 network segmentation through virtual routing forwarding (VRF). This is a technology allowing to create multiple instances of routing tables within a single network device, and thus enable several L3 networks to co-exist. As a consequence, ACI supports multi-tenancy since IP address spaces can be duplicated in separate VRFs. Note that VRFs are nothing new, Layer 3 virtual private networks (L3VPN) providers have been using this technique for years to segregate customer traffic.

Communication between multiple VRFs remains possible through a process called VRF route leaking, so inter-tenant communication can still occur as long as the IP addresses do not overlap.

A VXLAN network identifier (VNID) is always associated with a VRF, its use is discussed in section 2.5.2.

Application Profiles

An application profile is simply a container for endpoint groups. Assume that we want to build an e-commerce application using the microsevice pattern. First, we would create an application profile named *my-webshop* for instance, and then proceed to create several endpoint groups for the required services that we would place into the *my-webshop* application profile.

Endpoint Groups

An endpoint group is a collection of endpoints providing a similar function. It is used to logically group objects requiring similar policy. Endpoint examples include bare-metal servers, containers, virtual machines. EPGs can also refer to endpoints located outside the ACI fabric, in this case, we speak of external EPGs. Note that the concept of external EPGs is used later on for the ACI-SDA integration.

Bridge Domains

A bridge domain (BD) represents a layer 2 broadcast domain in ACI. While a VRF defines a unique address space, that address space can be divided into multiple subnets. those subnets are defined in one or more BDs referencing the corresponding VRF[9]. The bridge domain is often referred to as being "like a VLAN", which is close but not entirely exact. BDs are not subjected to the same limitations of VLANs such as the 4096 segments limits.

When a subnet is defined under a BD, a gateway is automatically created on all leaves of the fabric where the bridge domain exists, this is called a pervasive or anycast gateway. As we will see, the concept of anycast gateway is also used in Software-Defined Access. The IP address and the MAC address of the bridge domain's gateway is the same on all the leaves so that endpoints can seamlessly move between leaf nodes. A VNID, as well as a multicast IP address are associated with a bridge domain on its creation.

Contracts and Filters

ACI follows a zero-trust model, which means that inter-EPG communication is not allowed unless there is a contract. However, endpoints belonging to the same EPG can freely communicate with each other by default. Contracts define the types of traffic that can pass between EPGs, including the protocols and ports allowed.

The relationship between an EPG and a contract can be of two types: an EPG can provide or consume a contract (or both). And to be of any use, a contract needs to have at least a consumer and a provider. When an EPG provides a contract, communication with that EPG can be initiated from other EPGs according to the rules of the contract. When an EPG consumes a contract, endpoints in the consuming EPG are able to begin a communication with any endpoints of an EPG providing this contract.

The policy object specifying the specific port and protocol is called a filter in ACI. As you can see in Figure 2.6, contracts are not directly linked to filters. Instead, a contract can contain one or more child objects called subjects, which are linked to filters.

2.4.3 Access Policies

So far, the fundamental constructs for configuring the network with an application centric approach have been covered in the previous section. In fact, access policies have to be set up beforehand, otherwise, tenant policies are not activated. They specify how endpoints such as bare-metal servers, virtualization platforms such as VMware vSphere, or even Kubernetes clusters are attached to the ACI fabric.

As with Tenant policies, access policies also make use of many policy objects. Some of those are presented in Figure 2.8

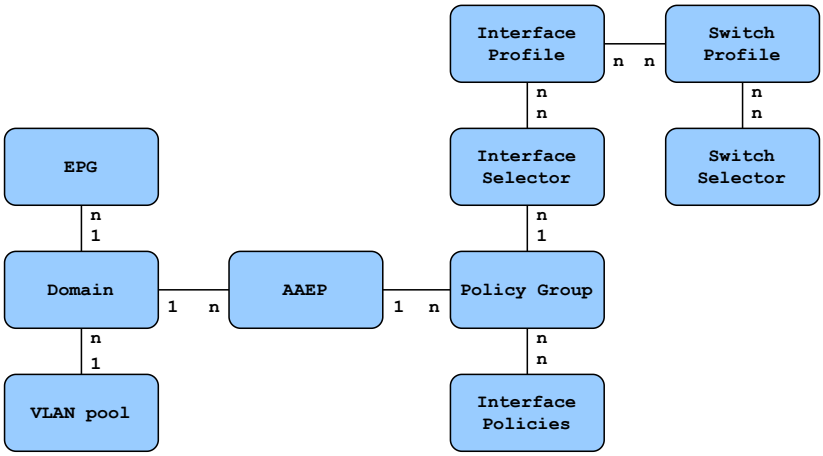

Figure 2.8: Access Policy Objects

Illustrating Access policies

Assume that a network engineer wants to deploy virtual port channels to connect his virtual environment to ACI through a Fabric Interconnect on the interface 1/1 of the leaves 1 and 2, and links the EPGs of its three-tier web application to the VMM domain that is allowed to pick VLANs tags in the 100 to 200 range. The logical constructs are presented in Figure 2.9.

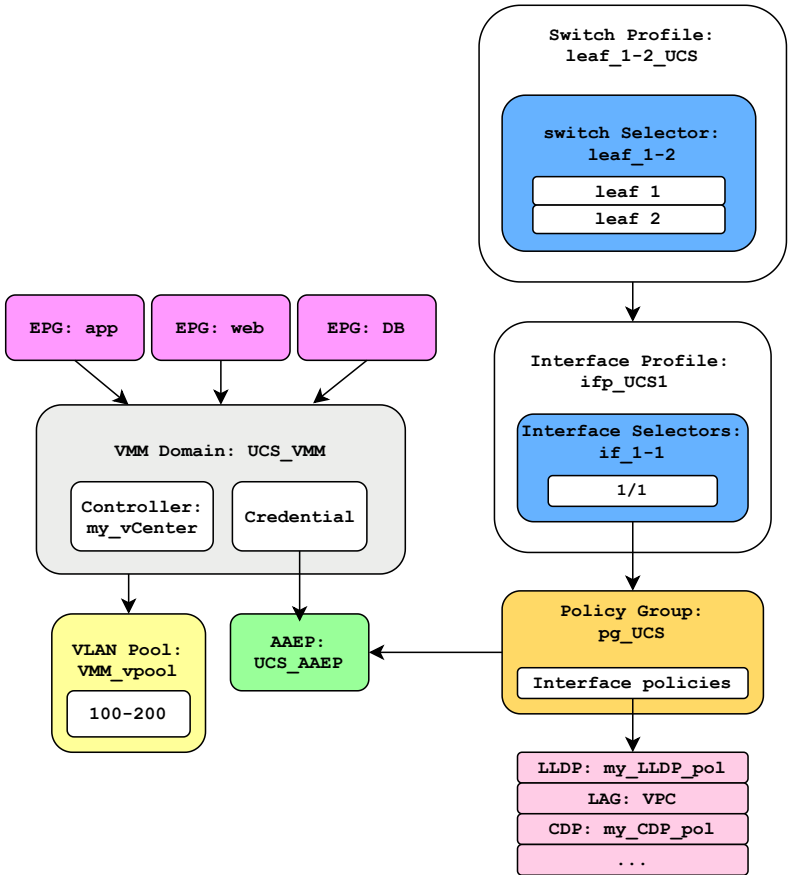

Figure 2.9: Illustrating Access Policies

We can say that access policies and tenant policies work hand-in-hand to define where and how endpoints or applications are connected.

Domain

All endpoints groups require a domain, which simply defines the scope of a VLAN pool (i.e where the pool will be applied). A domain can either be physical, virtual, or external. To illustrate, a VMware vSphere environment is an example of a virtual domain.

VLAN Pool

Why are VLANs needed in the first place? VLANs are still needed on the link between a leaf node port and the device that connects to it because the leaf must know the EPG of the endpoint originating the traffic. Therefore, VLANs are used by the leaves to identify endpoints. It is relatively easy if a device like a laptop is directly plugged into a leaf node port. In this case, Per Port VLAN can be used, which means that traffic reaching a leaf node from that port will be automatically mapped to a VLAN. However, complexity arises when a virtualization platform is connected to the ACI fabric as traffic initiated by virtual machines belonging to different EPGs reaches leaf nodes using the same link (or group of links for resiliency). Therefore, there must be an integration between the APIC and the management software (e.g VMware vCenter) of the virtual environment so that network policies can be applied to virtual workloads.

Attachable Access Entity Profile

The attachable access entity profile (AAEP) is the logical construct binding VLAN pools defined in a domain to leaf switch interfaces.

Interface Profile, Interface Selector, Policy Group, and Interface Policies

Configuring link-layer policies and applying them to switch interfaces is performed with several logical constructs. Interface policies are individual link layer policies, among them: link speed, link aggregation type (regular access port, virtual port channels, etc.) and LLDP. A policy group is simply is a collection of interface policies. An Interface profile is a container of interface selectors. An interface selector specifies one or multiple interfaces (e.g interface 1/45) and references a policy group for the selected interfaces.

Switch Profile and Switch Selector

Similarly, There are switch profiles and switch selectors. The configuration is applied to the interfaces of the switches when a switch profile is linked to an interface profile.

2.4.4 Integration of ACI and vCenter

The lab in Diegem runs a vSphere virtual environment managed by vCenter and is integrated with ACI.

Each VMM domain maps to a vSphere Distributed Switch (VDS) [69] provisioned by vCenter. A VDS functions as a single virtual switch across all the hypervisors (ESXi) and enables the VMs to maintain consistent network configurations as they migrate across multiple hosts, through vMotion for instance. Then, Each EPG linked to VMware VMM domain is associated with a distributed port-group automatically created by vCenter, which can be understood as a separate network within vSphere. The VM is then placed appropriately in the distributed port-group by the VMware admin.

The technique employed by VMware to tag the frames is Virtual Switch Tagging[68]. With this technique, the frames are tagged when leaving the ESXi host using the 802.1Q

protocol. Note that VXLAN can be used to overcome the 4096 VLAN limit. However, it comes to the price of larger headers.

2.4.5 Layer 4-7 Services

L4-7 services such as firewalls or load-balancers can easily be integrated with ACI. As one might expect, a multitude of policy objects are employed to insert such services, among others, L4-7 devices, service graph templates, contracts, and Policy-Based Routing (PBR) objects.

An L4-7 device is essentially the device applying functions to the network, it can be a virtual (e.g a Cisco ASAv) or a physical device. When the device is virtualized, it is referred to as a Virtualized Network Function (VNF).

A service graph is composed of function nodes referencing the L4-7 devices and terminal nodes (i.e a provider and a consumer node). A chain of function nodes can be formed, this process is called "Service Function Chaining" (SFC) and is thoroughly explained in RFC 7665. For instance, an engineer may want to filter then load-balance network traffic.

While traffic is routed using routing tables, PBR provides a mechanism to route traffic using policies, and there is a policy object for that. Therefore, you could for example redirect all HTTP traffic from an EPG to a particular location of your choice. in our case, an interface of the L4-7 device starting the chain of function nodes.

2.5 Routing and Forwarding in the ACI fabric

The routing and forwarding of packets in the ACI fabric are discussed in this section. It starts with the major protocols used by ACI, then some routing scenarios are presented.

2.5.1 VXLAN: an Overlay technology

Overview of VXLAN

Any multi-tenants infrastructure requires network isolation. Years ago, the method commonly used to provide this isolation in the data center was the VLAN (802.1Q).

the VLAN ID field included in the Ethernet header is 12 bits long, allowing 4096 possibilities. It is therefore possible to use 4096 different VLANs on the same device. However, it induces some limitations for data center networks. 4096 VLANs seems like a lot, but it is not enough in a cloud environment hosting the network infrastructure of hundreds of customers. Then, using traditional approaches like the Spanning Tree Protocol to build a loop-free topology can result in a large number of disabled links for an L2 network covering the whole data center. Hence, the introduction of Virtual Extensible LAN (VXLAN) available in RFC 7348[73].

VXLAN is a tunneling protocol, which makes it possible to "stretch" an L2 network over an L3 network. In other words, VXLAN is a Layer 2 overlay scheme on top of Layer 3 underlay network. A VXLAN tunnel (or segment) is identified by its VNI (VXLAN Network Identifier) that is encoded on 24 bits and gives over 16 million possibilities.

The functioning of VXLAN is rather simple. The Ethernet frame of the source endpoint is encapsulated in a UDP datagram (Mac-in-UDP encapsulation) at the entrance of the tunnel, then the packet is forwarded to the exit of the tunnel where it is de-encapsulated and forwarded to the destination endpoint. The elements in charge of this encapsulation (and

de-encapsulation) are called VTEPs, for VXLAN Tunnel Endpoints. The VXLAN packet format is shown in Figure 2.10

The source (resp. destination) IP address of the outer IP header is the address of the source (resp. destination) VTEP. Since the VXLAN packet is routed, a routing protocol is needed in the underlay network (IS-IS in ACI). Also, routers knowing nothing about VXLAN can be part of the underlay network; the only role they achieve is routing UDP datagrams.

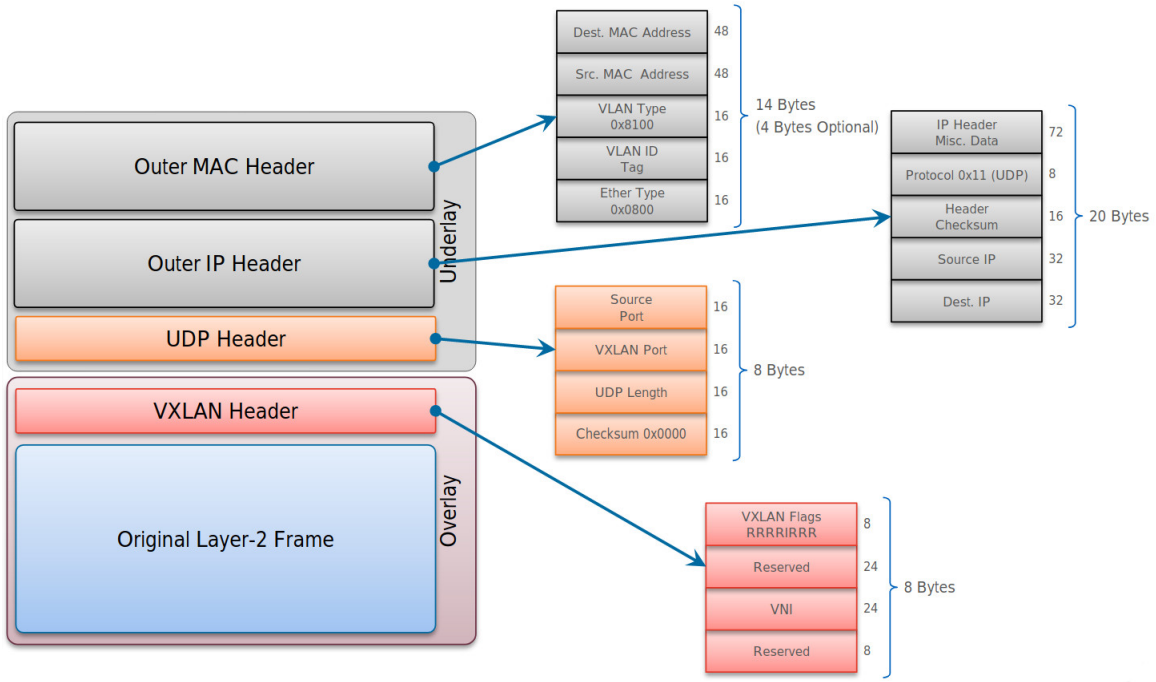

Figure 2.10: VXLAN Packet Format. By Bradley Wong, 2015, BRKAPP-9004: Data Center Mobility, VXLAN and ACI Fabric Architecture.

Endpoint Learning

How does the source VTEP know the destination VTEP it has to send the packet to? And, How are MAC addresses of the endpoint learned? This subsection aims to answer these questions. Essentially, there are two solutions to these problems: data plane learning and control plane learning.

First, the location of an endpoint from a VTEP perspective has to be clear. An endpoint's location is a mapping of its MAC address and the IP address of the VTEP to reach it.

In data plane learning, The VTEPs learn the location of endpoints a bit like regular L2 switches with a flood-and-learn approach. However, VXLAN uses a few tricks to make it more efficient.

One method is to use a multicast group for each VNI (or group of VNIs) that VTEPs join or leave depending on the endpoints connected to them. A VNI could for instance be an L2 network of a tenant. As a result, multi-destination traffic is forwarded to the VTEPs using multicast.

The other way to handle broadcast traffic is to perform Head-End Replication, but it is less efficient and scales poorly.

Unknown unicast (i.e destination MAC and IP addresses known by the source endpoint but the source VTEP does not known the destination VTEP to contact), broadcast and

multicast traffic (BUM traffic) is encapsulated in multicast VXLAN packets and sent to VTEPs via a multicast underlay network[61].

Control plane learning is a more recent solution and relies on MP-BGP with the EVPN address family combining MAC and IP information for learning the location of endpoints behind the VTEPs, which allows to reduces flooding traffic (e.g unknown unicast, ARP requests) in the network. In this model, the VTEPs run the MP-BGP protocol and exchange endpoint reachability information.

2.5.2 VXLAN in ACI

The leaf switches of the ACI fabric serve as VTEPs, so as traffic reaches a leaf switch and is destined for a remote leaf and assuming that the destination VTEP is known, the traffic is encapsulated in VXLAN packets and forwarded across the fabric through a spine switch chosen randomly with ECMP allowing ACI to automatically load balance the traffic and to share the workload among the spine switches. Then, the packets are de-encapsulated upon exiting the fabric and are delivered to the destination endpoint.[10]. With this model, ACI uses a full mesh, single hop, loop-free topology without the need to use STP[47]. This is illustrated in Figure 2.11. Note that as we will see with the COOP protocol, the spines are also able to encapsulate and de-encapsulate VXLAN packets.

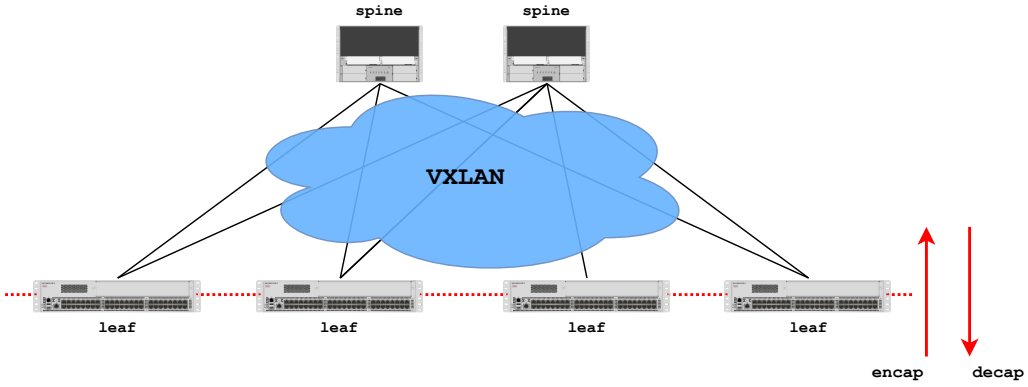

Figure 2.11: VXLAN in ACI

When source and destination endpoints are reachable via the same VTEP, packets do not travel over VXLAN tunnels.

Depending on whether the traffic is switched (e.g virtual machines belonging to the same LAN) or routed (e.g inter-EPG traffic), the VNID used by the source VTEP will be different. the VNID carried by the VXLAN header for switched traffic identifies a bridge domain (L2VNID), while the VXLAN header for routed traffic carries a VNID identifying a VRF (L3VNID).

Moreover, The ACI VXLAN header contains the source EPG for policy enforcement.

Also, one important selling point of ACI is that VXLAN is configured out-of-the-box and entirely managed by the APIC.

2.5.3 Council Of Oracles Protocol and Endpoint Learning

The approach taken by the designers of ACI regarding endpoint learning is a bit different than the solutions discussed in section 2.5.1. It can use the Council Of Oracles Protocol

(COOP) for this purpose, which greatly reduces broadcast traffic in the data center network. Despite its fancy name, COOP is a relatively simple protocol run by the leaf and spine switches of the fabric, where the spines are the said oracles forming a council.

Whenever a leaf switch learns about a new endpoint, it records its location (i.e MAC and IP addresses) in its Local Station Table. Then, this information alongside the leaf's IP address is reported to one of the spine selected randomly using COOP. The spine switches maintain a distributed database containing the location of all the endpoints, so whenever the location of an endpoint is learned by a spine, this information is stored in the spine's Global Proxy Table (or COOP database) then relayed to the other spines. Therefore, each spine has a complete record of every endpoint in the system.

Because this database is accessible, a leaf does not necessarily need to use multi-destination traffic to send packets to a remote endpoint for which it does not know the location. If a leaf does not know about a particular remote endpoint, it can forward packets to a spine that will forward packets to the remote leaf based on the information in its COOP database[52]. This explains why the oracles are also called spine proxies. This process is shown in Figure 2.13.

2.5.4 BUM traffic

L2 unknown unicast

One of the configuration options of a bridge domain is *L2 Unknown Unicast* that can be set to either *Flood* or *Hardware Proxy* mode.

When set in *Flood* mode, the leaves operate in a multicast-based flood-and-learn approach for unknown unicast L2 traffic. the traffic is sent from the source leaf along a multicast tree using the multicast IP address associated with the bridge domain to all the leaves of the fabric, it is then flooded in the bridge domain. Therefore, COOP is not used in this case. Figure 2.12 illustrates this behavior.

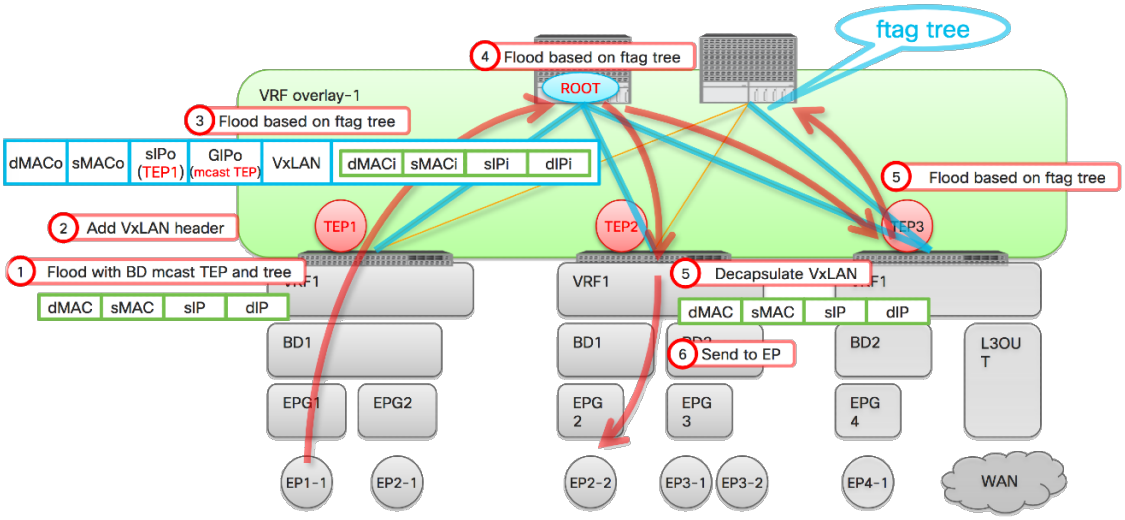

Figure 2.12: L2 Unknown Unicast with Flood Mode Enabled. By Takuya Yishida, 2020, BRKACI-3545: Mastering ACI Forwarding[74]

The Figure speaks for itself, however, a few comments can be made concerning the multicast tree. To begin with, all multi-destination (e.g broadcast, multicast) is carried as multicast traffic in the ACI fabric. ACI uses Forwarding Tag (FTAG) trees rooted in the spine switches to load-balance multi-destination traffic. There are 12 different FTAG trees in ACI used for user traffic. The last four bits of the multicast IP address assigned to a bridge domain are always set to zero, this room is for the FTAG that is part of the destination multicast address ("GIPo" for Global IP outer) and is selected based on the hash of the initial packet's content.

In *Hardware Proxy* mode, the leaves leverage the COOP databases of the spine switches; L2 Unknown unicast traffic is sent to a spine and a lookup is performed to find the destination VTEP, then the packet is forwarded to the leaf switch. the packet is dropped if no mapping is found in the COOP database. It is obvious, a mapping has to exist in the COOP database since the source endpoint knows the destination MAC address.

L3 unknown unicast

L3 unknown unicast is handled the same way as L2 unknown unicast in *Hardware Proxy* mode using the spine proxies (setting the *Unknown L2 Unicast* to *Flood* mode does not affect how L3 unknown unicast traffic is carried in the fabric). Figure 2.13 depicts how this is performed in the ACI fabric

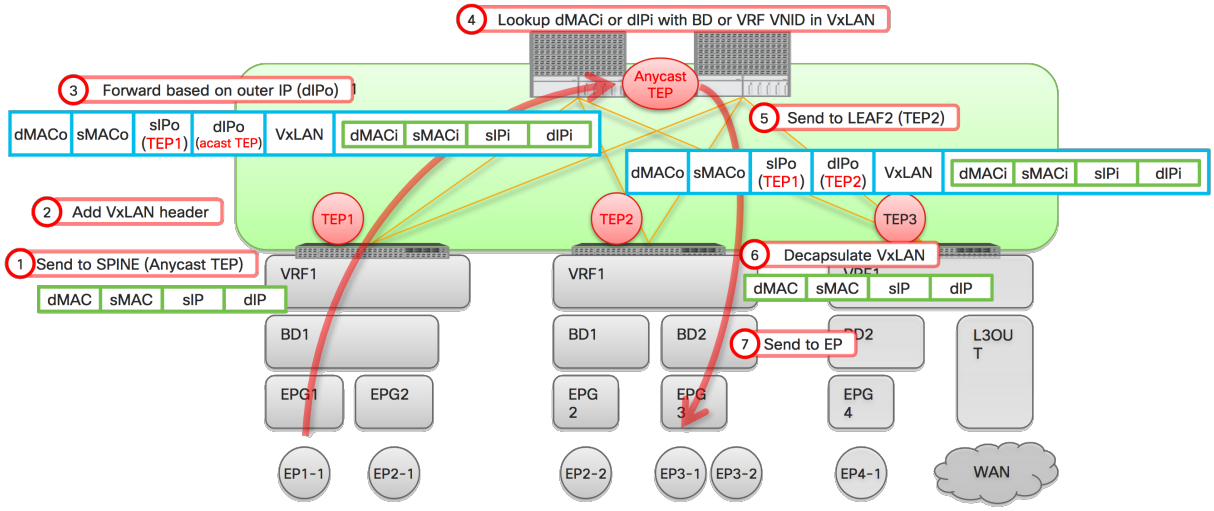

Figure 2.13: L3 Unknown Unicast. By Takuya Yishida, 2020, BRKACI-3545: Mastering ACI Forwarding[74]

As we can see, the packet from TEP1 to one of the spine is still encapsulated using VXLAN (step 3). What is interesting is the destination IP address ("dIPo" for destination IP outer) used in this packet. This anycast IP address is installed on each switch and leverages ECMP capabilities enabling load sharing. The destination MAC address ("dMACo") of the packet is also an anycast address since the leaf and the spine switches are directly connected.

Once the packet is received by the spine, it is de-encapsulated and the lookup is performed against the COOP database (step 4). If successful, the spine encapsulates the initial packet in a VXLAN packet whose destination IP address ("dIPo") is the IP address of the destination VTEP (TEP2 in this example) retrieved in the lookup (step 5).

Assuming that the COOP lookup for and endpoint failed, the packet is dropped and a technique called ARP gleaning is used in order to retrieve the location of the destination endpoint. ARP glean packets are sent to all the leaves of the fabric, which instruct them to generate ARP requests inside the BD subnet of the destination endpoint provided that the BD subnet is present on the leaves. The destination endpoint responds to the ARP request and thus allows the leaf switch to relay the location of the endpoint to an spine switch that will record it in the COOP database. ARP gleaning is shown in Figure 2.14

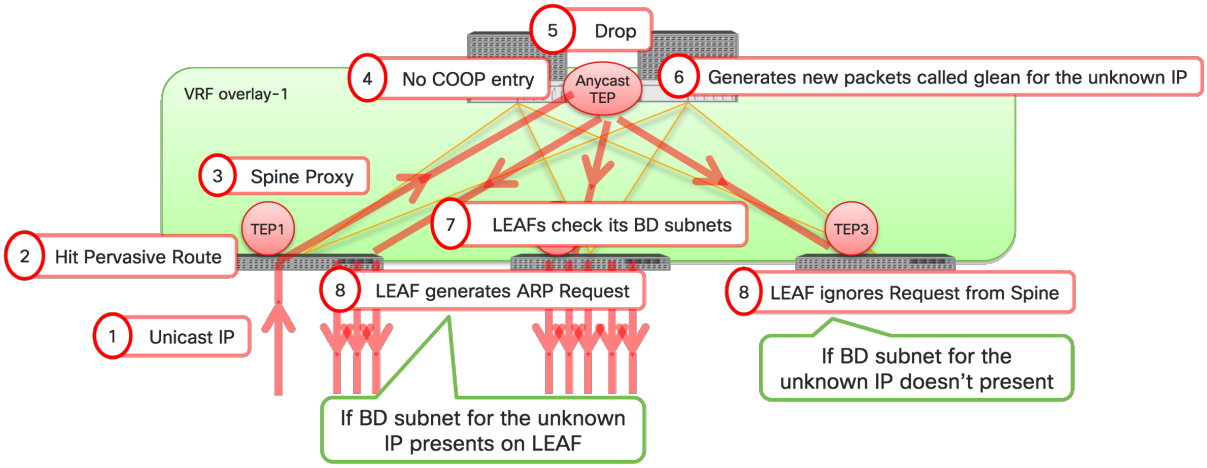

Figure 2.14: ARP Gleaning. By Takuya Yishida, 2020, BRKACI-3545: Mastering ACI Forwarding[74]

Chapter 3

Kubernetes

Applications of the beer business are deployed in Kubernetes[2]. This chapter is dedicated to the understanding of containers, Kubernetes, and the integration of Kubernetes and ACI.

3.1 Containers

Just as the transportation industry uses containers to isolate the various goods to be transported on ships, trains, trucks and airplanes, software development is increasingly using the concept of containerization.[39]

Container technology eliminates the "it works on my machine" syndrome as it provides a way to package software code and all its dependencies so that it can run the same, regardless of the infrastructure. Once packaged, container-based applications can be easily deployed in any environment, whether it's a private or public cloud or a developer's laptop. They also help reduce conflicts between the development and operations teams by separating areas of responsibility; developers can focus on the logic and dependencies of their application while operation teams focus on deployment and management, without having to worry about details like software version or application-specific configuration.[20] Therefore, it allows enterprises to build and deploy applications faster.

Containers are one of the key enablers of the DevOps practices discussed later on, and are closely related to microservices. Having a basic understanding of how they operate is thus valuable.

3.1.1 Container terminology

Container technology comes with its own terms, the most important ones are explained below.

- **Container image:** a container image is an immutable file, which packages an application or a service, its configuration, and its dependencies. Often, a container image is built upon another image, with additional customization. For instance, if one wants to containerize a Python application, he will build his image upon an already available Python image or a minimalist Linux image on which he will install Python.
- **Container registry:** a registry is a storage and distribution system for container images. Container images can be pushed (resp. pulled) to (resp. from) the registry, it can be seen as a bookshelf for container images[38]. Container images are usually associated with a tag for versioning purposes. We are using an on-premise container registry in the lab named Harbor. So, one might create his images or use images created by others published in a registry.

- **Container engine:** a container engine is a piece of software used for pulling, pushing and building container images, and instantiating containers. Docker is an example of a container engine, others include CRI-O, RKT, LXC, etc. More details about the Docker engine are provided in section 3.1.2.
- **Container:** a container is the running instance of a container image.

3.1.2 Docker

Docker is technically a container engine but it can be seen as a complete platform for running containerized applications. It enables developers to package up an application with all its dependencies, to deploy it in any environment where Docker is installed, and to share it publishing it to a registry. As we will see in the following sections, Kubernetes leverages container engines, it is therefore important to have basics understandings

As shown in figure 3.1, Docker relies on a traditional client-server architecture. Clients can communicate with the Docker daemon using a REST API.

The Docker daemon is responsible for listening to requests and does all the heavy lifting such as running containers, pulling images, etc.[27]

When one installs Docker on its laptop, it is shipped with a Docker client. The communicates with the Docker daemon using the client accepting commands such as *docker run* (for running a container), the client then sends these commands to dockerd.

Note that one can communicate directly to Docker daemon by scripting API calls.

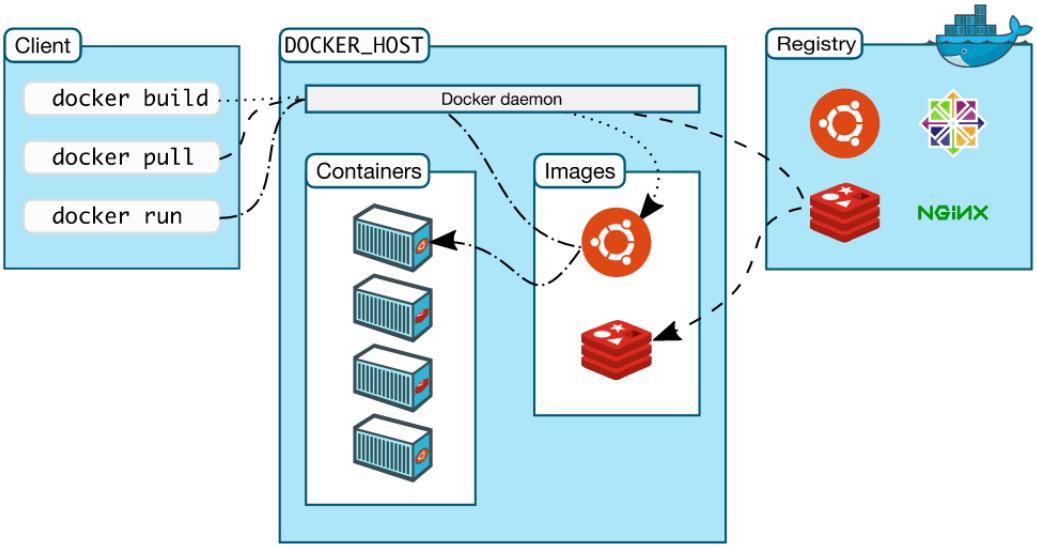

Figure 3.1: Docker architecture. by Docker.inc[29]

3.1.3 Containers vs. Virtual Machines

Like virtual machines, containers are isolated packaged computing environments that can be spun up and torn down on a whim[28].

Containers leverage OS-level virtualization whereas virtual machines depend on hardware virtualization. Indeed, containers running on a host share the same underlying host's operating system kernel, this is shown in Figure 3.2. Therefore, they do not embed a copy of the operating system and rely on a container engine to create an isolated computing environment.

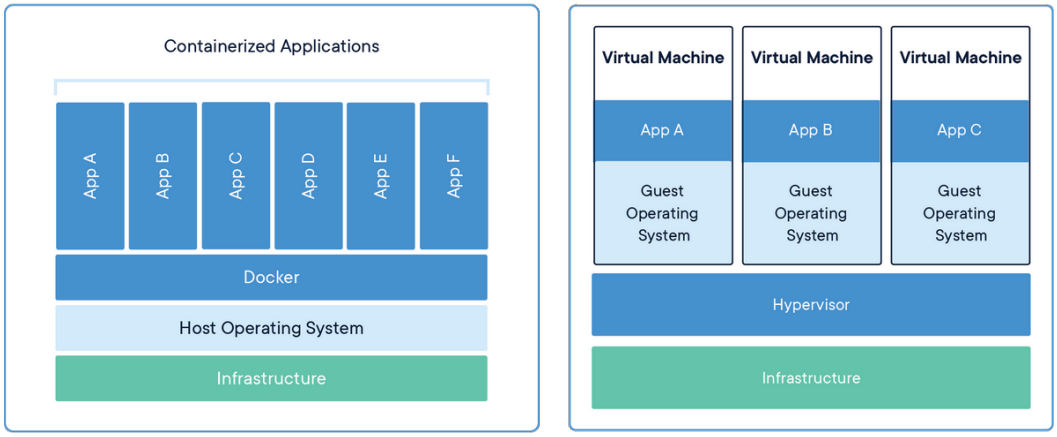

Figure 3.2: Containers vs. VMs. by Docker.inc[30]

On Linux systems, containerization uses kernel Linux features called namespaces to provide the isolated workspace and control groups (cgroups) to control the resource usage of a container. From the Linux Programmer's manual, "*a namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource*"[32]. Thanks to namespaces, containers believe that they exist in completely isolated environments within the host. Namespaces come in different flavors in the Linux OS. For example, there is the PID namespace for process isolation, the mount namespace for the setup of a container filesystem and volumes, and the network namespace that will be covered in more details in the next section.

Containers will in no way replace virtual machines, they are complementary technologies that help improve the use of IT resources, with their advantages and disadvantages.

Containers are lightweight entities that can be spun up in a matter of milliseconds, unlike virtual machines that require to boot a whole operating system. Besides, the size of container images is measured in megabytes and is therefore much smaller than virtual machine images, which typically consume gigabytes. However, from a security perspective, VMs provide complete resource isolation, which allows for better security.

Overall, virtual machines solve infrastructure problems by enabling enterprises to get more out of servers. Whereas containers solve application problems by increasing portability as they are decoupled from the underlying infrastructure, enabling microservices and DevOps practices, and improving resource utilization.

3.1.4 Docker networking

To understand how the network of a Kubernetes cluster is managed by ACI, it is first necessary to understand the building blocks of container networking. the main elements are the network namespace, bridges, virtual Ethernet interfaces.

Whenever a container is started, it runs in its network namespace, so the container is isolated from the host network. A Linux network namespace virtualizes the network stack

and can be considered a VRF in an ordinary network; the network namespace has its routing table, network interfaces, IP addresses, etc.

A Linux bridge functions as a virtual network switch, it is used to forward packets between interfaces that are connected to it within a single IP subnet. For example, Docker creates a bridge (named *docker0* by default) for container packet switching when installed on a machine.

As mentioned earlier, a container operates in its network namespace. Communication between different network namespaces is made possible by virtual Ethernet interfaces, these interfaces also called veth interfaces are created in pairs. A veth pair can be seen as a cable connecting a container to a bridge or a container to another container. By default, Docker creates a veth pair between the newly created container and the default bridge *docker0*.

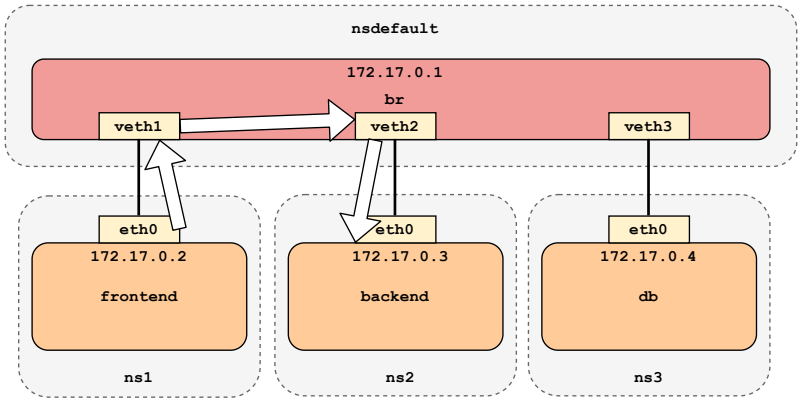

Figure 3.3: Containers networking constructs

These constructs are illustrated in Figure 3.3 for three containers (i.e. *frontend, backend* and *db*), each of them are attached to the their namespace (i.e. *ns1, ns2* and *ns3*). In this example, traffic from the frontend container to the backend container would first be forwarded to the bridge lying in the host namespace named *nsdefault* through the veth pair *veth1@eth0*, then be directed to the backend container via the second veth pair *veth2@eth0*.

3.2 Kubernetes

Managing a containerized application is a challenging task, especially when it is broken down into a large number of entities. To illustrate, Uber adopted a microservice architecture around 2012 and their platform has grown to approximately 2200 services[19]. Container orchestrators such as Kubernetes aim to solve this challenge[36].

Kubernetes also abbreviated K8s (for the 8 letters between the first and the last letter of "Kubernetes"), is a system for automating the deployment, scaling, and management of containerized applications. For instance, Kubernetes comes with self-healing and can automatically restart containers that fail, it can scale the number of containers based on CPU utilization, and evenly distribute the network traffic to the containers.

This project has become open-source since 2014 and was started at Google around 15 years ago. It is is now maintained by the Cloud Native Computing Foundation[2].

The Kubernetes cluster was deployed using the Cisco Container Platform (CCP), it nicely integrates with vCenter and ACI.

3.2.1 Kubernetes Architecture

A working Kubernetes deployment also referred to as a Kubernetes cluster, is a distributed system composed of master and worker nodes. The master node implements the control plane functionalities that control the K8s cluster, much like the APIC for ACI. And, the worker nodes or compute nodes are in charge of running the containerized workloads named Pods. The different elements of Kubernetes architecture are presented in Figure 3.4.

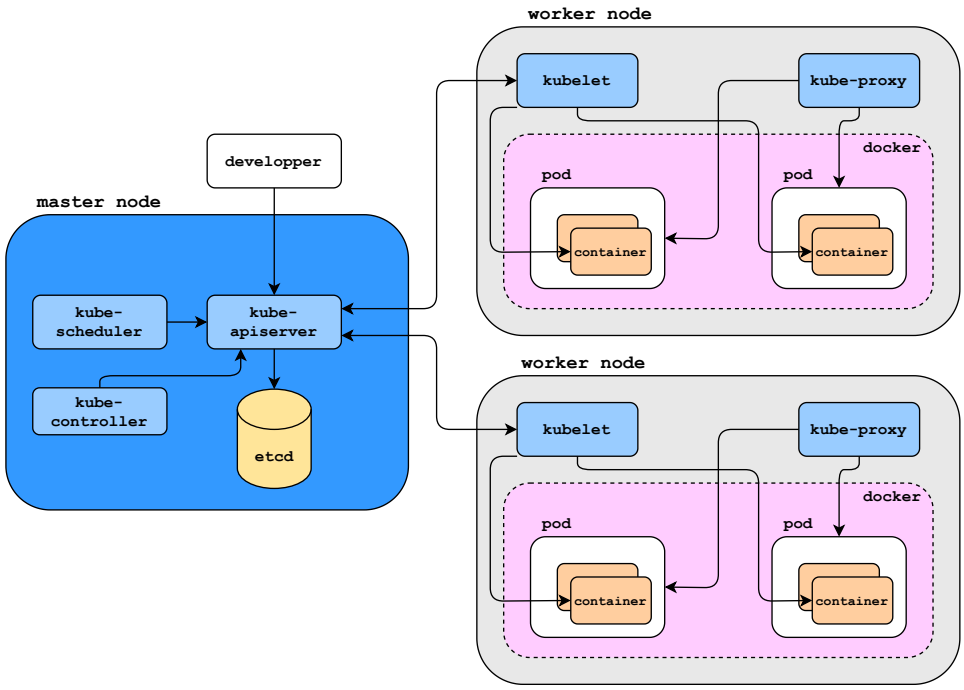

Figure 3.4: Kubernetes architecture

Master Node

The master node consists in an API server (*kube-apiserver*), a scheduler (*kube-scheduler*), a controller (*kube-controller*) and a key-value store database (*etcd*).

- The API server is the entry point for configuring the K8s cluster. It allows a user to configure the K8s cluster through a client (kubectl) using configuration files (K8s manifests), scripts, or automation software (Terraform, Ansible, etc.). Manifest Files are written in YAML, an example for deploying 3 replicas of an Nginx server with a manifest is provided in Listing 3.1[1]
- The scheduler assigns Pods to nodes based on CPU utilization or other metrics.
- The role of the controller is to continually monitor the cluster and make sure that the cluster runs is in the desired state the programmer has defined. It is actually composed of multiple controllers responsible for different types of K8s objects. So, the controller dictates the commands to the API server, which applies them.
- The database is used for storing configuration data and information about the state of the cluster.

```
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
```

Listing 3.1: Kubernetes Manifest File

Worker Node

A Kubernetes cluster typically includes multiple worker nodes for resiliency purposes. For example, a developer will deploy multiple instances of the front-end microservice of its application on different nodes so that his application remains accessible in the event of a node failure. It is composed of the following components:

- A compute node communicates with the control plane through an agent named *kubelet*. It is this agent that commands the operations within a node, such as deploying Pods.
- Each node also runs a container engine (we are using docker in the lab) to run the containers.
- A Pod is the smallest deployable unit of computation that can be managed in Kubernetes. It represents a single instance of an application (such as the front-end of a three-tier application) and consists of one or more containers. Pods are usually deployed using a Deployment, which is a Kubernetes object allowing a developer to roll out multiple Pods at the same time for one of its microservice.
- Pods can communicate with one another with the use of a Kubernetes object called a Service. By default, Services are implemented by *kube-proxy* through the manipulation of the IPtables rules of the worker node.

Closer View at Pods

From a networking point of view, a Pod is a single network endpoint, (although it may contain multiple containers) as each newly created Pod is assigned an IP address from a range assigned to the node. It means that if a developer deploys a new version of a microservice on Kubernetes, the old set of Pods for that microservice will be destroyed and the IP addresses of the newly created Pods will be different. Furthermore, the new Pods can even be deployed on different worker nodes.

A Pod is similar to a group of Docker containers with shared namespaces and file system volumes. As shown in Figure 3.5 A Pod consists of a Pause container, and the containers running the code of the application. The pause container is the first container that Kubernetes creates for every Pod, it is responsible for holding the network namespace and serves as the "parent container" for all the containers in the Pod[34] and acts as a gatekeeper. Its job is to keep Pod alive even if one of the application container fails, it avoids the creation of a new Pod and thus a new network namespace if one of the containers fails within the Pod.

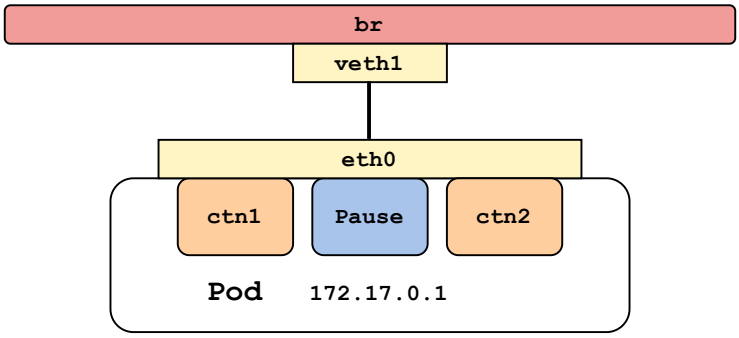

Figure 3.5: Anatomy of a Pod

By default, Kubernetes relies on traditional Linux bridges, and veth pairs for the connectivity.

Closer view at Services

A Service is a way to expose a microservice running on a set of Pods as a network service[3]. It solves the following problem: if a set of Pods (cart microservice of a webshop application) provides some functionalities to another set of Pods (payroll microservice), how can the payroll Pods find out and keep track of the IP addresses to connect to the cart Pods? And as we know, if a new version of the cart microservice is deployed, a new set of Pods with different IP addresses will replace the old ones so Services solves the dilemma of having to keep up with every transient IP address assigned to the Pods.

To put it simply, a Service defines a single IP/port combination that provides access to a set of Pods. The way a Service is tied to a set of Pods is through the use of labels and selectors; Pods are labeled and a Service references one or multiple labels. And, a Service has a domain name that can be resolved with kube-dns, an internal DNS server for the Kubernetes cluster. As a result, even if a Service is destroyed and redeployed, Pods can still use the Service without noticing the change in the IP/port combination of the newly created Service.

By default, K8s implements the Services by manipulating the NAT table of the IPtables rules of the worker nodes.

There are several types of services: ClusterIP, Nodeport and Loadbalancer services. Only the ClusterIP and Loadbalancer services will be briefly discussed in this section as they are the ones I use.

ClusterIP is a Service that can only be accessed by other Pods in the cluster, It is used for inter-pod communications. ClusterIP Services leverage source NAT and destination NAT for the forwarding of packets. Figure 3.6 shows how a payroll pod would communicate with a cart pod using a Service.

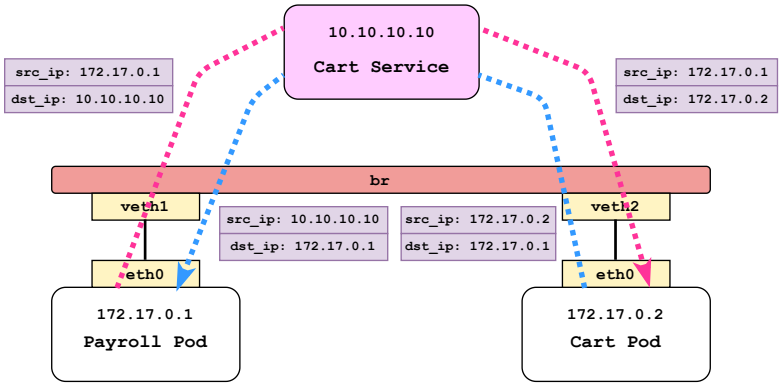

Figure 3.6: ClusterIP Service

A K8s Loadbalancer exposes an application running in Pods to be reachable from outside the cluster. For instance, the frontend service of an application would lie behind a Loadbalancer Service.

3.3 ACI and Kubernetes integration

3.3.1 ACI-CNI Plugin

Networking in Kubernetes is handled by network plugins (*kubenet* by default), which implements the Container Networking Interface (CNI) specifications[14]. This is why these plugins are called CNI plugins. CNI plugins are responsible for handling the networking-related tasks in K8s.

The ACI CNI plugin provides integration between the APIC and a Kubernetes cluster connected to an ACI fabric. This integration is implemented across two main functional areas:

Firstly, The Cisco ACI CNI plugin extends the ACI fabric capabilities to Kubernetes clusters in order to provide IPAM, networking, load balancing, and security policies to containerized workloads. In addition, all Pods are connected to the VXLAN overlay network provided by ACI thanks to the plugin.

Secondly, the entire K8s cluster is mapped to a VMM domain in ACI. This allows APIC to have access to all resources deployed in the cluster, including the number of nodes, services, Pods and their MAC and IP addresses, etc. The APIC, therefore, has full visibility of the K8s cluster, just as it has visibility of a vSphere environment.

When the Kubernetes cluster is provisioned by the Cisco Container platform along with ACI CNI, numerous ACI objects are automatically created for the integration: a tenant, bridge domains, a K8s application profile, EPGs, contracts and a logical load balancer associated with a service graph template with PBR capabilities.

The ACI CNI plugin is also transparent for the Kubernetes developers as all the networking policies are implemented in ACI. One of the main advantages of ACI CNI is that it allows applications running in Kubernetes to communicate with applications running in other form factors such as bare metal, virtual machines, or other K8s clusters, without any constraint or performance bottleneck, because a container is treated like any other endpoint in ACI[53].

When provisioning resources to Kubernetes, all the developer needs to do is add some annotation (metadata) for ACI, which may be different depending on the type of isolation desired. There are 3 types of isolation, from the laxest to the strictest: cluster isolation (a K8s cluster is mapped to an EPG); namespace isolation (a K8s namespace is mapped to an EPG); deployment isolation (a k8s deployment is mapped to an EPG). In the case of deployment isolation, which we use for the deployment of the microservice applications, we need to specify the tenant, the application profile and the EPG. Then, ACI network policies can be configured for the set of Pods belonging to the deployment. Note that once the deployments are annotated and mapped to EPGs, all Pods can no longer freely communicate with all other Pods belonging to different EPGs, unless a contract authorizes them to do so.

ACI CNI Plugin components

As seen in figure 3.7, ACI CNI consists in multiple components.

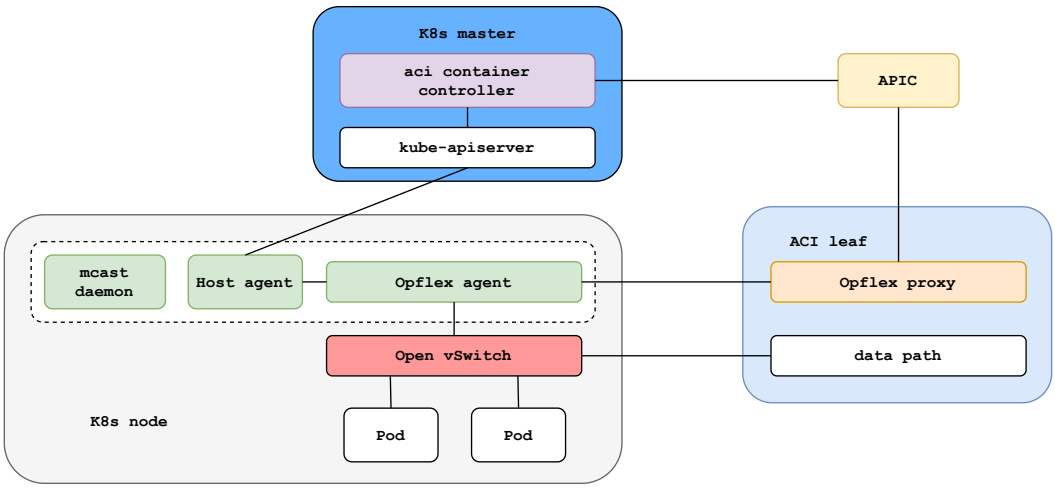

Figure 3.7: ACI CNI components

The Opflex proxy process runs in the ACI leaf and propagates the network policies configured in the APIC to an Opflex agent, which runs on the Kubernetes worker node.

On the K8s node, a pod consisting of 3 containers runs. It consists of the host agent, the Opflex agent and a multicast daemon (not shown on the diagram). The Opflex agent translates the network policies it receives from the Opflex proxy into Openflow rules and programs the Open vSwitch (OVS), which is a virtual multilayer switch. Therefore, OVS acts a bit like an ACI virtual leaf. The Host agent keeps track of the endpoint running in the nodes and receives updates from the API server of the master node. The Open vSwitch runs in a container that implements the data path so it takes care of configuring the routing and switching between the Pods, the K8s Services, and enforces network policies like ACI contracts.

The ACI container controller handles IPAM, delivers endpoint information and pushes configurations to the APIC when required.

Open vSwitch

The Opflex implementation of OVS is composed two different OVS bridges, each of them implements a different set of features: *br-access* and *br-int*. This is shown in figure 3.8

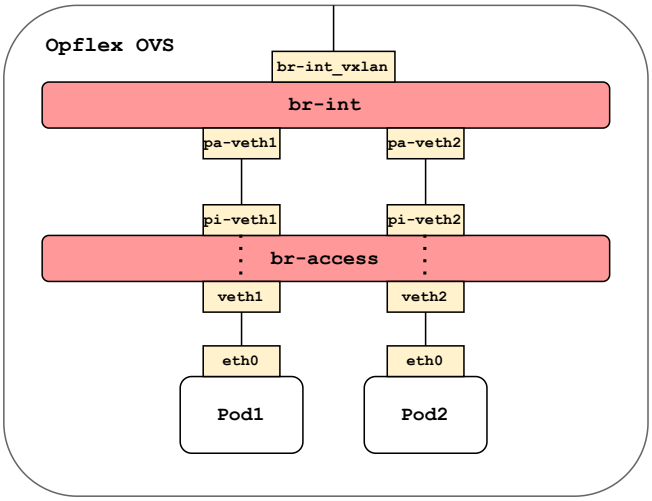

Figure 3.8: Opflex OVS

The bridge *br-access* is used to enforce Kubernetes Network Policies, which are similar to ACI contracts. These policies define how Pods are allowed to communicate with each other, where Pods are identified by labels and can be used in conjunction with ACI policies. For example, let's say that we have 3 Kubernetes deployments, labeled red, blue, and green. The Pods of the blue and red deployment belong to the same EPG, the Pods of the green deployment belong to another EPG, and these two EPGs are able to communicate with each other thanks to an ACI contract. With Kubernetes Network Policies, we can specify for instance that the red Pods can not communicate with the green Pods even if a contract authorizes them to do so.

Whereas *br-int* is in charge of performing routing, switching, enforcing ACI policies such as contracts, NAT, and load balancing leveraged by Kubernetes Services. As a result, ClusterIP Services are no longer implemented via the modifications of IPtables rules.

Since the nodes of the cluster are forming an L2 overlay network, packets need to be encapsulated in either VLAN or VXLAN frames when traffic is not destined to a Pod local to the node. Hence the usefulness of the *br-int_vxlan* interface.

Thankfully, OVS comes with utilities allowing to simulate network traffic between two endpoints in the K8s cluster and to visualize the Openflow rules traversed, which is quite insightful.

Traffic from Pod to another backend Pod running in two different nodes was simulated using the *ovs-appctl ofproto/trace* command (Listing 3.2). The following command was run on the K8s master node and instructs a Pod (frontend Pod) to send traffic to another Pod (backend Pod) over port 3000. Some information needed to be retrieved such as the source MAC and IP addresses of the Pods and the port of the OVS bridge the frontend Pod is connected to.

```
1 kubectl exec -n kube-system aci-containers-openvswitch-6bp4x -- ovs-appctl
    ofproto/trace br-access tcp,in_port=22,dl_src=06:0a:f4:52:f9:1d,dl_dst=f6
    :91:16:c2:e6:a0,nw_src=10.51.0.110,nw_dst=10.51.0.82,tp_src=56000,tp_dst
    =3000
```

Listing 3.2: command to simulate traffic between two Pods

Listing 3.3 shows the output.

```
bridge("br-access")
-------------------
 0. in_port=22,vlan_tci=0x0000/0x1fff, priority 100
    load:0x3->NXM_NX_REG6[]
    load:0x1->NXM_NX_REG0[]
    load:0x15->NXM_NX_REG7[]
    goto_table:2
 2. reg0=0x1, priority 8192
    goto_table:3
 3. metadata=0/0xff, priority 1
    output:NXM_NX_REG7[]
     -> output port is 21 #see pa-veth3cceaea

bridge("br-int")
----------------
 0. ip,in_port=15,dl_src=06:0a:f4:52:f9:1d,nw_src=10.51.0.110, priority 30
    goto_table:1
 1. in_port=15,dl_src=06:0a:f4:52:f9:1d, priority 140
    load:0x708002->NXM_NX_REG0[]
    load:0x1->NXM_NX_REG4[]
    load:0x4->NXM_NX_REG5[]
    load:0x1->NXM_NX_REG6[]
    goto_table:3
 3. priority 1
    goto_table:4
 4. reg4=0x1,dl_dst=f6:91:16:c2:e6:a0, priority 10
    load:0x720006->NXM_NX_REG2[]
    load:0xa02684c->NXM_NX_REG7[]
    write_metadata:0x7/0xff
    goto_table:11
11. tcp,reg0=0x708002,reg2=0x720006,tp_dst=3000, priority 8064, cookie 0xe
    goto_table:12
12. metadata=0x7/0xff, priority 15
    move:NXM_NX_REG2[]->NXM_NX_TUN_ID[0..31]
     -> NXM_NX_TUN_ID[0..31] is now 0x720006
    move:NXM_NX_REG7[]->NXM_NX_TUN_IPV4_DST[]
     -> NXM_NX_TUN_IPV4_DST[] is now 10.2.104.76
    output:5
     -> output to kernel tunnel #send traffic via VXLAN tunnel on port 5

Final flow: tcp,reg0=0x1,reg6=0x3,reg7=0x15,in_port=22,vlan_tci=0x0000,dl_src
    =06:0a:f4:52:f9:1d,dl_dst=f6:91:16:c2:e6:a0,nw_src=10.51.0.110,nw_dst
    =10.51.0.82,nw_tos=0,nw_ecn=0,nw_ttl=0,tp_src=56000,tp_dst=3000,tcp_flags
    =0
Megaflow: recirc_id=0,ct_state=-new-est-inv-trk,ct_mark=0,eth,tcp,in_port=22,
    vlan_tci=0x0000/0x1fff,dl_src=06:0a:f4:52:f9:1d,dl_dst=f6:91:16:c2:e6:a0,
    nw_src=10.51.0.110,nw_dst=10.51.0.82,nw_ecn=0,nw_frag=no,tp_src=0x8000/0
    x8000,tp_dst=3000
Datapath actions: set(tunnel(tun_id=0x720006,dst=10.2.104.76,ttl=64,tp_dst
    =8472,flags(df|key))),4
```

Listing 3.3: OVS Openflow rules traversed

As we can see on the trace, traffic is flowing through the two OVS bridges and is then encapsulated in a VXLAN frame (39) and sent to a leaf switch. The VXLAN tunnel is 0x720006 (7471110), which corresponds to what is shown on the APIC (see Figure 3.9).

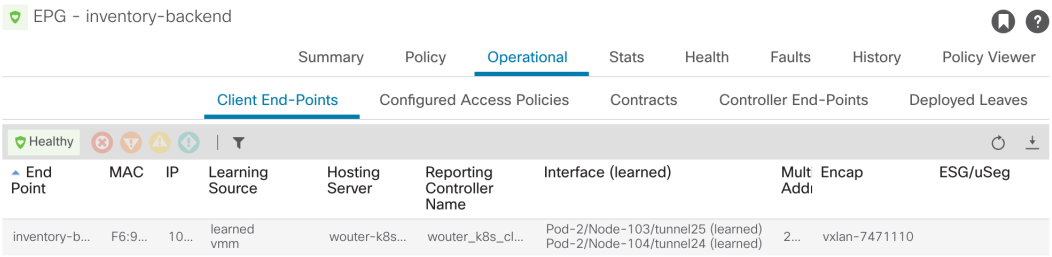

Figure 3.9: VXLAN Encapsulation Backend Pod

3.3.2 Accessing Pods From the Outside

When a client wants to access an application running in K8s from the outside, it is done through a Kubernetes LoadBalancer lying in front of the application, and an ACI PBR service graph. Upon the creation of a LoadBalancer Service in K8s, the ACI container controller automatically performs some configuration on the APIC.

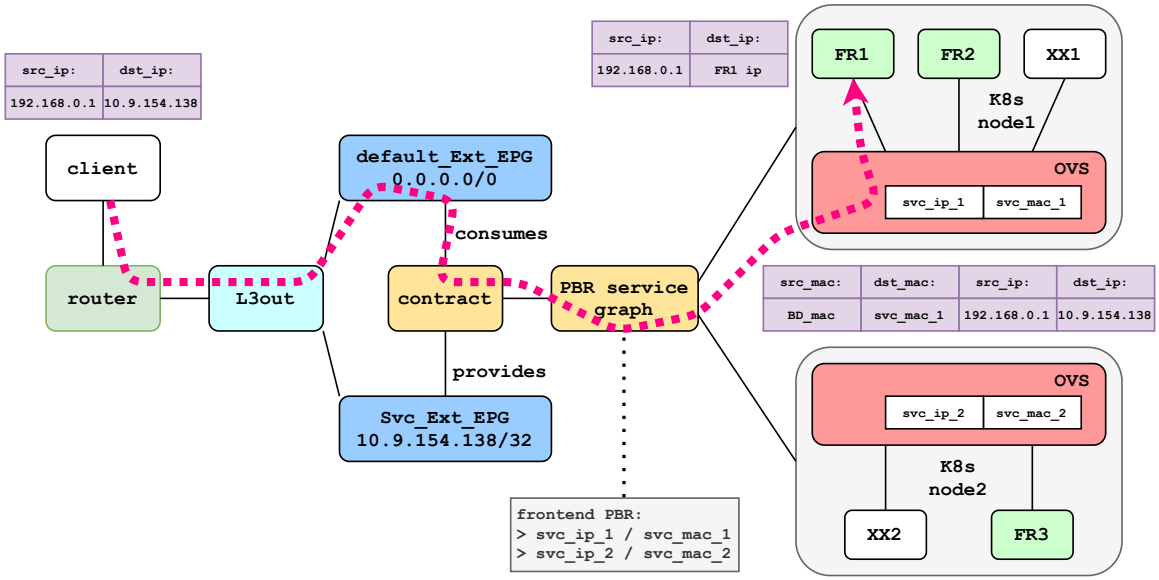

Figure 3.10: K8s Loadbalancer Packet Walk

In the example presented in Figure 3.10, the client wishes to access the frontend of an application running in the Kubernetes cluster. The frontend deployment has three Pods, FR1 and FR2 running on the first node, and FR3 running on the second node. And, a K8s LoadBalancer named FR_LB has been created to access the frontend application, which instantiated a PBR service graph in ACI attached to a contract, and an external EPG mapped to the IP address of the K8s Loadbalancer.

First, the request of the client reaches the ACI fabric, is going through the L3out and is classified into the default external EPG (default_Ext_EPG) by ACI. Then, ACI performs a lookup for the destination IP address of the packet (10.9.154.138), which is mapped to the external EPG of the Loadbalancer Service (Service external EPG) exposing the frontend application

Then, the PBR attached to the contract provided by the Service external EPG and consumed by the default external EPG is applied. Next, ACI load-balances the request to a K8s

node running the frontend application. Let us dig a little bit deeper to understand what is happening during this step.

When ACI manages the K8s network with ACI CNI, a virtual endpoint (also called a service endpoint) used solely for PBR purposes is assigned to each K8s node. These endpoints belong to the bd-kubernetes-service bridge domain whose MAC address is BD_mac. Therefore, what the PBR service graph does after selecting the K8s node (K8s node1) as a next hop, is replacing the source and destination MAC addresses by BD_mac and svc_mac_1. The packet is then ready to be forwarded to the K8s node over the corresponding VXLAN tunnel since the APIC is aware of those service endpoints.

The figure below (3.11) is a screen capture of the APIC showing the available service endpoints for an exposed microservice. As we can see, the Pods are running on three different nodes in this case.

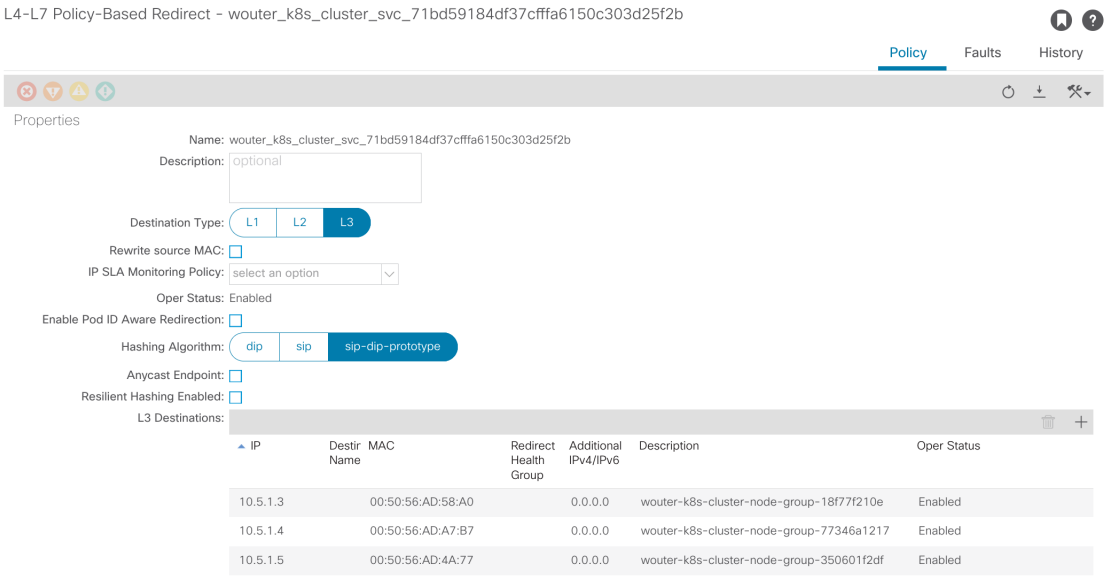

Figure 3.11: ACI Service Endpoints of Pods Exposed by a K8s Loadbalancer

Once the request reaches the Open vSwitch, it is load-balanced among the Pods exposed by the Service and DNAT is performed by the *br-int* bridge of the OVS. The final destination is Pod FR1 in this example.

Chapter 4

Project Objectives

4.1 Objectives and Deliverables Summary

This section summarizes the objectives pursued throughout the project. The objectives are twofold, making a demonstration of the ACI-SDA integration, and building an Infrastructure-as-Code demonstration around ACI, Kubernetes, and vSphere.

4.2 Story Telling

The IT infrastructure of the company is composed of two domains: the SDA domain for enterprise networking, and the ACI domain for running the applications on premise. Three applications are running in the data center:

- an inventory application accessed by the retail employees to add beers to the inventory.
- a webshop application accessed by customers to buy some beers.
- a finance application accessed by the financial department, and that displays the history of the purchases.

4.3 Deliverables

- Three applications developed with a microservice architecture (Section 6.1).
- Automated ACI configuration deployment with Terraform (Section 6.2).
- Automated deployment of the applications to Kubernetes using either Terraform, or Helm charts (Section 6.3).
- Automated deployment of the applications to virtual machines (Section 6.4).
- Ticketing system and CICD job notifications leveraging Gitlab issues, webhooks, and Webex Teams (Section 6.5).
- Network configuration for the use-case (Section 6.1 & 8.2)

4.4 Remarks

Initially, the ACI-SDA integration was supposed to be performed with the applications running in the Kubernetes cluster. However, the first phase of the integration is too limited and does not permit it. As a result, the applications were deployed on virtual machines with a VM per container strategy. Details about the deposited code can be found in Section 10.

Chapter 5

What is Infrastructure-as-Code?

The Cisco's Beer Factory's IT infrastructure is managed according to infrastructure-as-code (IaC) and DevOps principles, from network configuration and virtual machine provisioning to application deployment in Kubernetes.

In this chapter, IaC and DevOps principles are explained, followed by sections explaining the main tools that were used.

5.1 What is Infrastructure-as-Code?

As the name suggests, IaC is the process of managing and provisioning IT infrastructures (e.g. servers, network, storage, public clouds, etc.) through code and configuration files, which specify infrastructure elements and how they should be configured.[49].

When using IaC, the configuration files contain the specification of the infrastructure, which makes it easier to edit and distribute configurations. It can also help engineers to tear down, or reproduce the infrastructure easily. But more than anything, IaC provides speed and consistency when it comes to managing the environment.

Some IaC tools like Terraform also promote the immutable infrastructure approach, an approach whereby infrastructure elements should never be changed after they have been deployed, rather they should be replaced. This is where containers come in very handy as destroying and spin up containers is not resource expensive. By doing so, it helps to avoid "snowflakes", i.e. infrastructure elements that require additional configuration beyond that covered by the automation tools[7].

The concepts of declarativeness and imperativeness discussed in Section 2.2.1 and can also be applied to IaC. In a declarative approach, the desired end state of the infrastructure is defined and the automation software takes care of the rest, whereas in an imperative approach, the script supplies the infrastructure step by step.[12]. Both approaches have their use-cases and are complementary. For example, provisioning virtual machines in the public cloud declaratively with a tool like Terraform would be the best fit. And it would more suitable to configure these VMs in the next step of the provisioning process with a shell script or an Ansible playbook.

As we will see in the next section, IaC and DevOps are closely linked.

5.2 DevOps

This section presents the DevOps practices, introduces the concept of CI/CD pipelines on Gitlab, and finally explains how DevOps can be applied to IaC.

5.2.1 Defining DevOps

The term DevOps is a mashup of the words "development" (Dev) and IT "operations" (Ops). Defining DevOps is not an easy task, but broadly speaking, it is a set of ideas, practices, and tools that increases an organization's ability to deliver applications and services at high velocity[51]. It explains why microservices and Kubernetes go hand in hand with DevOps, as they allow for rapid deployment of an updated version of the application.

Since applications need IT resources, working under the DevOps model is also a kind of cultural shift and require the development and infrastructure teams to work closer together[48]. However, going as far as saying that "*these two teams are merged into a single team*"[51] is still an utopia in my honest opinion, especially in big organizations where employees are siloed in multiple teams: virtualization, network, security, development, Kubernetes, cloud, etc.

Continuous integration (CI) and either continuous deployment or either delivery (CD) are two important principles of DevOps.

Continuous integration is a practice where all working copies (i.e branches) of the developers are frequently merged into a central repository, after which automated build and tests are performed. All these automated tasks are run in what is called a "pipeline".

Continuous deployment is the process of automating the deployment of application releases into test, staging, and production environments. Continuous delivery is similar, except that the deployment of the release into production requires manual approval. Multiple tests (load tests, integration tests, etc.) should also occur between the deployment to staging and production.

5.2.2 Gitlab

Gitlab is a DevOps platform that is installed on-premise that we use for the project. Among other things, it provides source control with git, continuous integration, and deployment pipelines capabilities. The pipeline features of Gitlab are quickly explained in this subsection.

in Gitlab, a pipeline is a set of automated processes attached to a repository, it consists of stages, which contain jobs that can be executed in parallel. A pipeline can either be triggered when new code is pushed or even manually with an API call, and if a job of the pipeline fails, the pipeline is said to be broken and the subsequent stages are not run. Examples of pipelines will be provided in the following sections.

Anything can be executed within a job, one can for instance execute a script in Python, deploy configuration with an IaC tool, build a Docker image and push it to a registry, or even trigger the pipeline of another repository.

The jobs are picked up and run by an application called a runner, which must be associated with an executor that determines the environment each job runs in. In our case, we use the Docker executor, so each job runs in a container.

Therefore, if one wants to use Ansible in a job, he has to make sure that it is installed and configured in the supplied Docker image. This step can involve the creation of tailored images with Dockerfiles, which are nothing but text documents that contain all the commands a user could call on the terminal to assemble images.

A pipeline is defined in a YAML file named *.gitlab-ci.yml*. An example is provided below in listing 5.1 for illustration purpose. In this example, 3 stages are defined and there is one job

per stage except for the second stage, which contains 2 jobs. Some rules can also be defined, for instance, job-3 can only be executed manually.

```
stages:
    - stage-1
    - stage-2
    - stage-3

job-1:
    image: ubuntu:latest
    stage: stage-1
    script:
        - echo "do some stuff"

job-2-1:
    image: ubuntu:latest
    stage: stage-2
    script:
        - echo "do some stuff"

job-2-2:
    image: ubuntu:latest
    stage: stage-2
    script:
        - echo "do some stuff"

job-3:
    image: ubuntu:latest
    stage: stage-3
    script:
        - echo "do some stuff"
    rules:
        - when: "manual"
```

Listing 5.1: .gitlab-ci.yml example

A pipeline can be better visualized as such (see Figure 5.1).

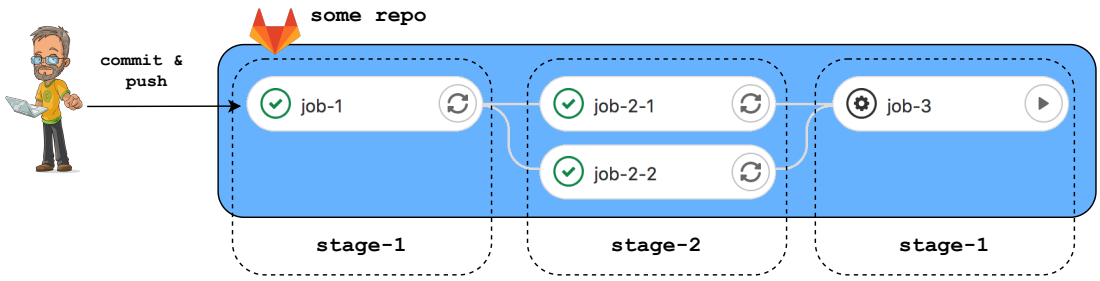

Figure 5.1: Pipeline Representation

There is of course much more to say about the DevOps capabilities of Gitlab, but the basics are covered for what comes in the next sections.

5.2.3 Relationships Between DevOps and IaC

DevOps and infrastructure-as-code are closely related, in fact, we could say that IaC is one of the key enablers, or is part of the DevOps movement.

Defining the whole infrastructure to code means that it should also be treated the same way that any other code would be. Therefore, DevOps best practices can also be applied to IaC.

Consequently, git repositories can be used as a single source of truth to deliver IaC. And, IT operations can be performed directly from the pipelines, which is the strategy adopted by the infrastructure team of Cisco's Beer Factory. Such a strategy is also called the GitOps framework[18].

5.3 IaC Tools

Several automation tools used throughout this project are presented in this section. Among them, Terraform, Helm charts, Packer, and Ansible.

5.3.1 Terraform

Terraform is an open-source IaC tool written in Go leveraging a declarative approach, it is developed by Hashicorp and is mostly used for creating remote resources. This process is referred to as provisioning.

The declarative approach is best suited to infrastructure provisioning as it allows the state of the infrastructure to be fully captured. Indeed, at a glance, one can see what is currently deployed and configured by looking at the code base, and running Terraform multiple times will not affect the end result. That is why Terraform is said to be idempotent as it keeps track of created resources.

This is not necessarily the case with procedural IaC tools as one would have to know about the full history of all the changes that have taken place.

Configuration files are written in a human-readable language called HCL (Hashicorp Configuration Language). An example of HCL for provisioning an ACI Tenant in Terraform using a resource is shown below in listing 5.2. The set of available resources that one can use depends on the Terraform Provider that he is using, which is responsible for understanding API interactions and exposing resources.[23] For instance, There is a dedicated Provider for ACI, Kubernetes, Amazon AWS, and vSphere, etc.

```
1 resource "aci_tenant" "tenant" {
2     name = "my_tenant"
3 }
```

Listing 5.2: Terraform HCL Example for Creating a Tenant in ACI

Terraform code is therefore no more than configuration files containing resources that can be bundled into modules. One can think of a module as a regular function in programming, which can be called, take in parameters, and then create some resources.

In my honest opinion, writing Terraform configuration files in itself is not hard as it is just a matter of reading the documentation of the Providers correctly, however, the difficulty lies in writing reusable and flexible modules because the expressive power in Terraform, as in most declarative languages is rather limited, so doing logic (conditions, loops, etc.) is quite tricky. And, keeping a Terraform codebase DRY, (Don't Repeat Yourself), maintainable and scalable.

As shown in Figure 5.2, Terraform uses a client-only architecture, which means that it does not require the installation of additional software on the platform to be provisioned. Under the hood, Terraform simply makes API calls based on the resources defined in the configuration files.

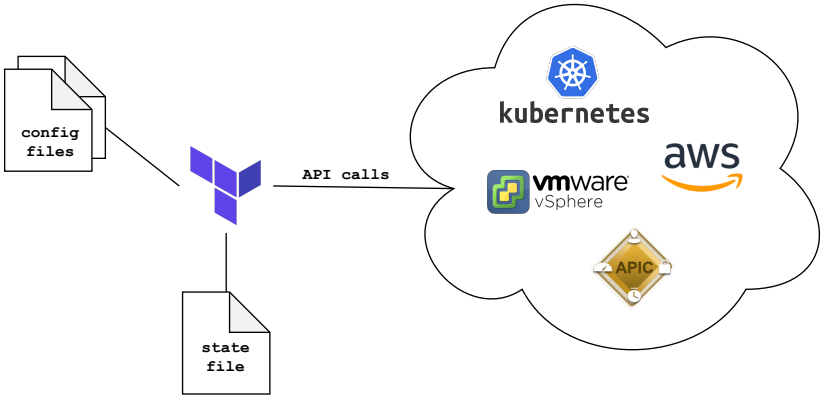

Figure 5.2: Terraform client-only Architecture

Terraform maintains the state of the infrastructure (i.e. what has been provisioned) in a state file. This JSON file contains mappings from the Terraform configuration to objects provisioned in the real world[6], it can be seen as a kind of internal database used by Terraform.

So, when Terraform creates a remote object, it will record the identity of that remote object against a particular resource instance in the Terraform configuration, and then potentially update or delete that object in response to future configuration changes.[25]

Gitlab integrates well with Terraform and provides a backend for storing state files. Storing the state file remotely allows it to be shared among all members of the infrastructure team, and also enables state locking, which prevents concurrent writes to file.

The Terraform workflow is quite straightforward, it consists of a refresh, plan, and apply phase.

Terraform starts with a refresh to update the state file, this phase is performed to reconcile real-world configuration drifts, which can occur, for example, if a resource previously created with Terraform has been deleted manually or by another configuration tool. In this case, it will delete the resource from the state file before running the planning phase.

During the plan phase, Terraform determines the actions to be carried out in order to achieve the desired state defined in the configuration files and outputs a plan. This phase is convenient for reviewing the configuration and check whether the changes match your expectations. The workflow of this plan step is illustrated in Figure 5.3.

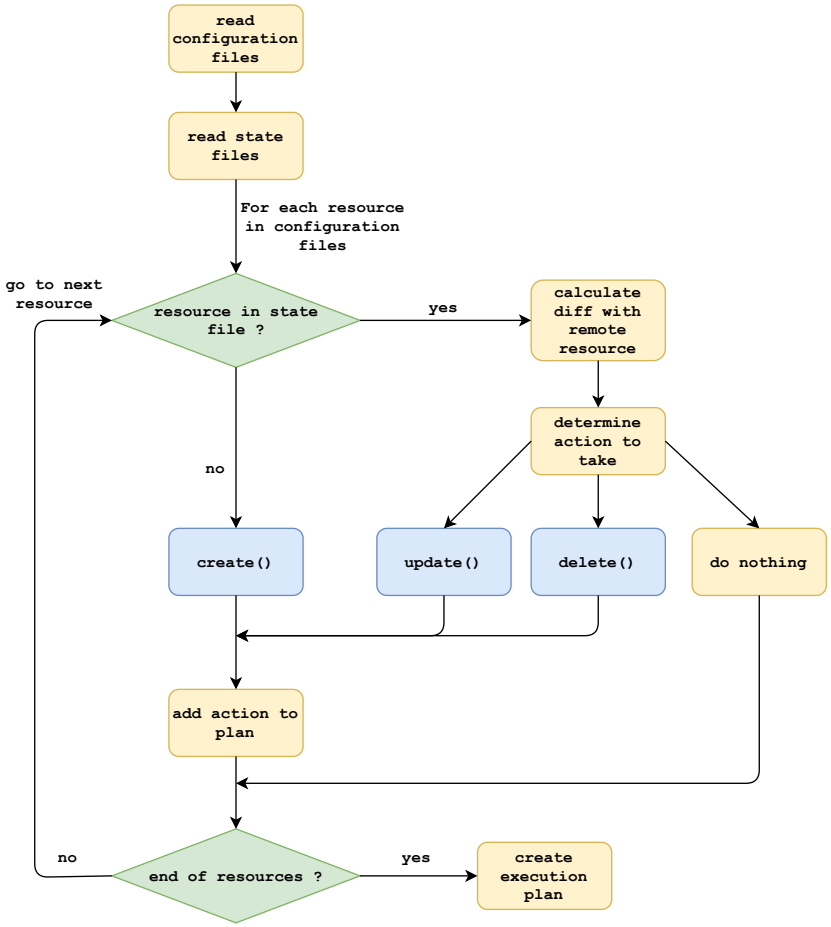

Figure 5.3: Terraform Plan Workflow

The diagram speaks for itself, however, a comment can be made on the last step when Terraform creates an execution plan. Internally, Terraform builds a directly acyclic graph (DAG) to determine the order of the actions it will perform, because some resources may depend on other ones. For example, an application profile has to be created before an EPG.

Once the changes are reviewed, the apply phase is run, which applies the changes to the infrastructure required to reach the desired state, and updates the state file accordingly.

5.3.2 Helm Charts

Helm is a package manager for K8s, it allows developers to bundle K8s applications in what are called "charts" and to share them with the community, following the same kind of philosophy as Docker. Basically, a Helm chart contains templates used to generate K8s manifest files based on possible inputs fed to the Helm template engine, which can then be deployed in the cluster with the Helm client.

The packaging capabilities of Helm really shine when the complete application is developed and needs to be deployed to multiple environments, or even published to a chart registry.

5.4 Image Building with Packer

5.4.1 Packer

Packer is another tool that automates the creation of machine images using Builders. Packer's Builders can be viewed in the same way as Terraform's Providers. They are able to

create an image for a single platform by reading a configuration file. Examples of Builders include Docker, VMware, Amazon EC2.

Using Packer to build an image is quite simple as all there is to do is write a JSON configuration file. In its simplest form, it be can separated into two sections; a building section used to build the image, and a provisioning section used to edit it using regular shell scripts, Ansible, Powershell, etc.

5.4.2 VM Templates

A VM template is a golden image of a virtual machine used for the purpose of VM cloning. It is particularly useful when multiple VMs with similar configuration needs to be deployed. Administrators can either convert a virtual machine to a template or clone a virtual machine from a template, leaving the template intact.

One important feature of a VM template is that it can not be powered on or edited once it has been created, which provides safety as administrators can not accidentally modify the virtual machine used as a template.

When Deploying a VM from a template, vSphere also provides a functionality allowing administrators to customize the guest operating system of the resulting VM during the cloning phase. Among other things, the network interfaces, default IP gateway, and DNS can be configured.

5.5 Ansible

Ansible is an open-source automation tool sponsored by Red Hat. As moving applications to virtual machines was a kind of last resort solution, I did not have time to explore all the good, the bad and the ugly things Ansible has to offer.

From an architectural point of view, Ansible works a bit like Terraform as it does not require any additional software to be installed on the devices to be configured. So, it is made up of a device running Ansible (i.e control node) and the devices managed by the control node (i.e managed nodes). There is no limit to what Ansible can configure as long as it can communicate with the device via an API or connect to it via SSH. For instance, Configuring ACI, deploying virtual machines, and customizing Linux servers can be performed by this tool. Unlike Terraform, Ansible does not keep track of the state of the infrastructure. It is therefore up to the administrator to know what has already been deployed. Automation work can also be implemented in modules (e.g. a module for creating an EPG in ACI), which can be idempotent depending on how they are developed. This means that it is up to the module code to check the state of the device and determine what changes need to be made.

Automation tasks are grouped into what is called a "playbook", a YAML file passed to the Ansible engine, usually accompanied by an inventory file listing the devices to be configured.

Using Ansible to configure virtual machines is illustrated below in Figure 5.4.

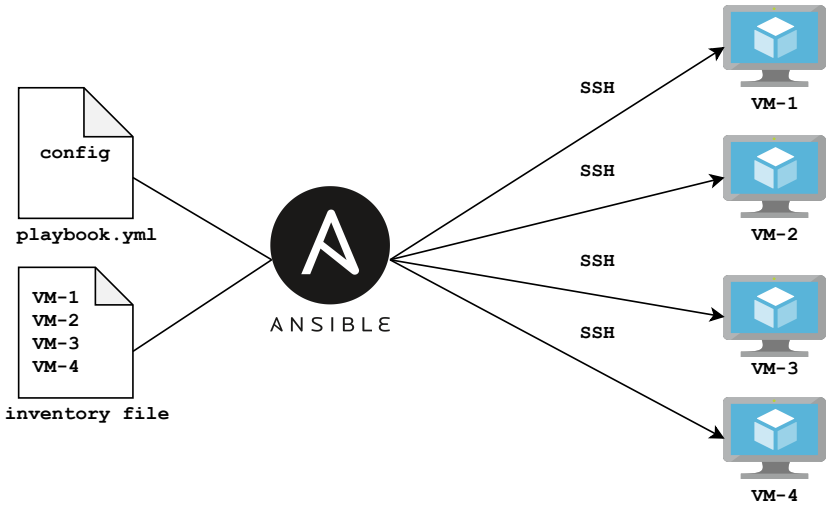

Figure 5.4: Ansible Workflow

Chapter 6

Infrastructure-as-Code Demonstration

6.1 The Applications

The inventory, webshop, and purchase history applications of the beer business are developed with the microservice pattern. These are deployed to Kubernetes, and to virtual machines. In both cases, communication between the different microservices and access to the applications are managed by ACI. Note that these applications are only for demonstration purposes and are in no way production-grade applications.

6.1.1 Architecture and Technologies

The microservices

Figure 6.1 shows the different microservices composing the applications, and how they are connected. each application is composed of a web frontend, and one or multiple backends.

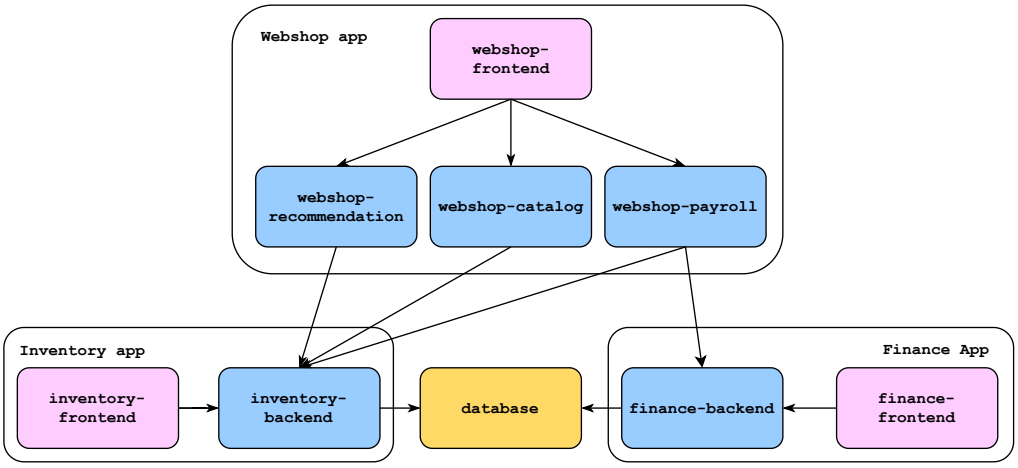

Figure 6.1: Cisco's Beer Factory Microservices

On Kubernetes, each microservice runs in its Pod in the cluster. Each Pod of the backend or database microservices is exposed by a ClusterIP Service since those are only during used for inter-pod communications. Whereas the Pods of the frontend microservices sit behind a LoadBalancer Service. On virtual machines, each microservice runs in a container on separate VMs.

Frontends

The frontends are single-page applications developed in Javascript using the Vue.js framework. A single-page application loads only a single web document and then updates the content of this document via AJAX requests. This, therefore, allows users to navigate on a website without loading new web pages from the server, which can result in a more dynamic experience.[40]. Vue.js was selected for its simplicity, user-friendliness, and documentation. The Bootstrap[5] framework was used for the CSS. In addition, Wouter styled the frontends by incorporating some CSS. The web documents are served by Nginx, which also provides an API gateway redirecting the client requests to the corresponding backends. The frontends of the web applications are shown below on Figure 6.2, 6.3, and 6.4.

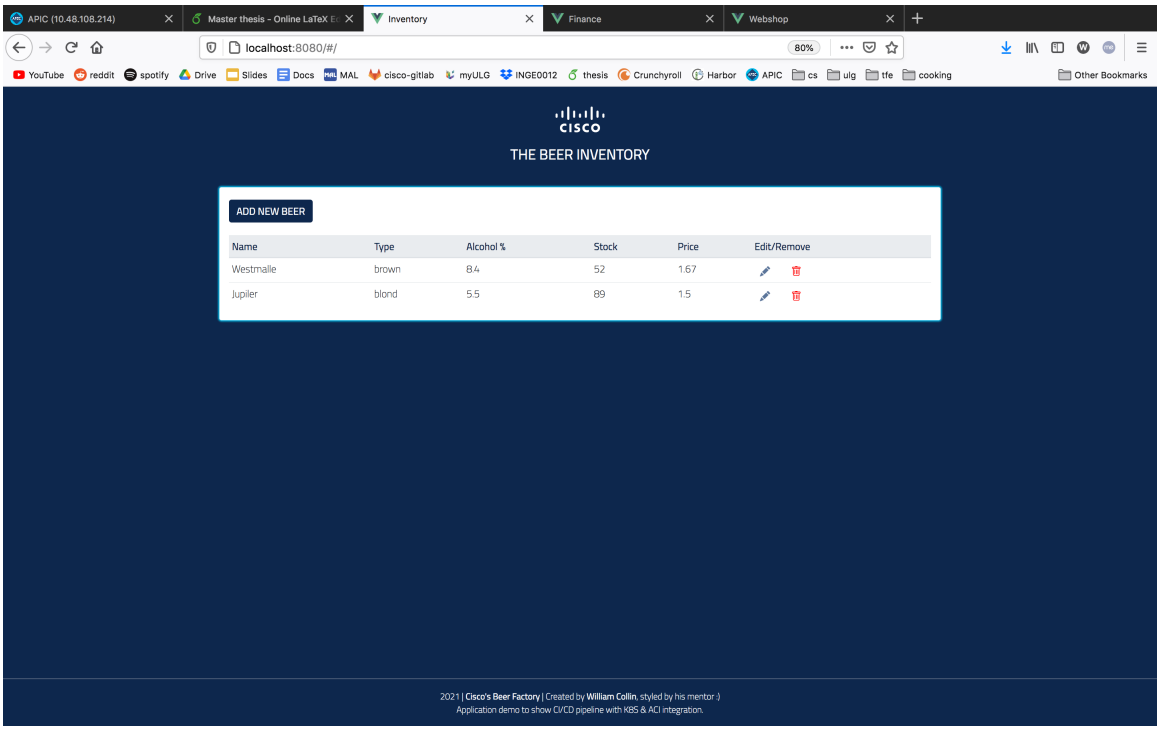

Figure 6.2: Frontend of the Inventory application

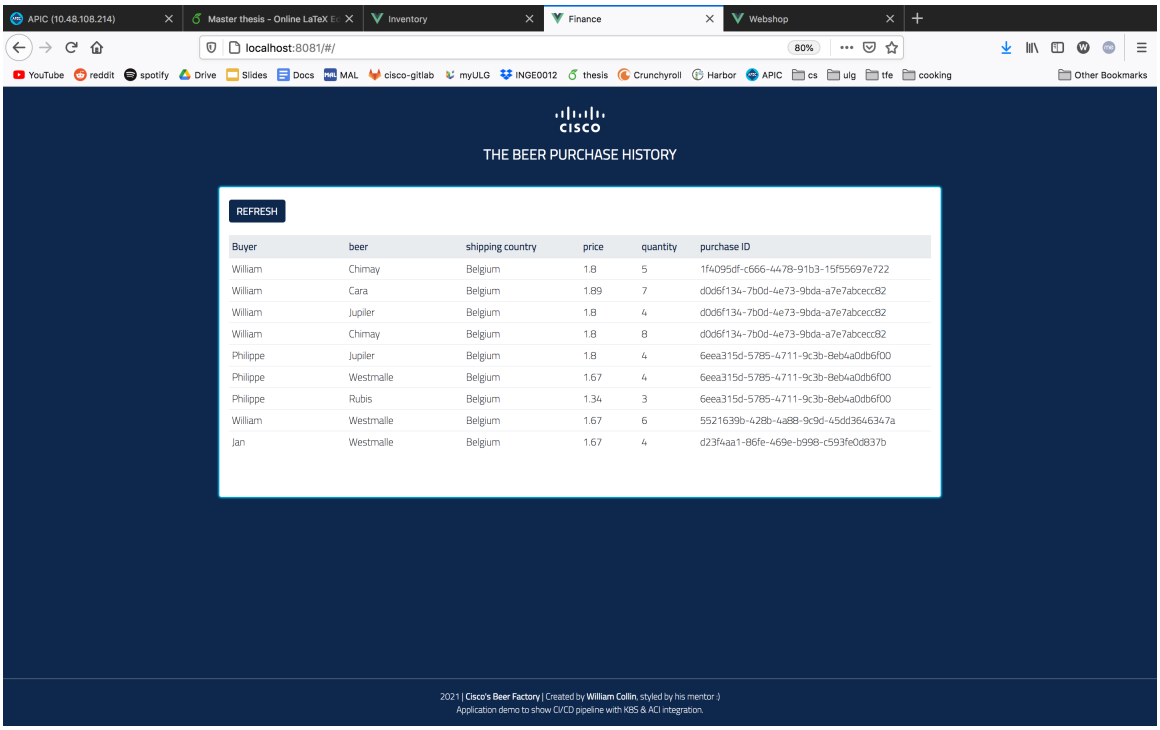

Figure 6.3: Frontend of the Finance application

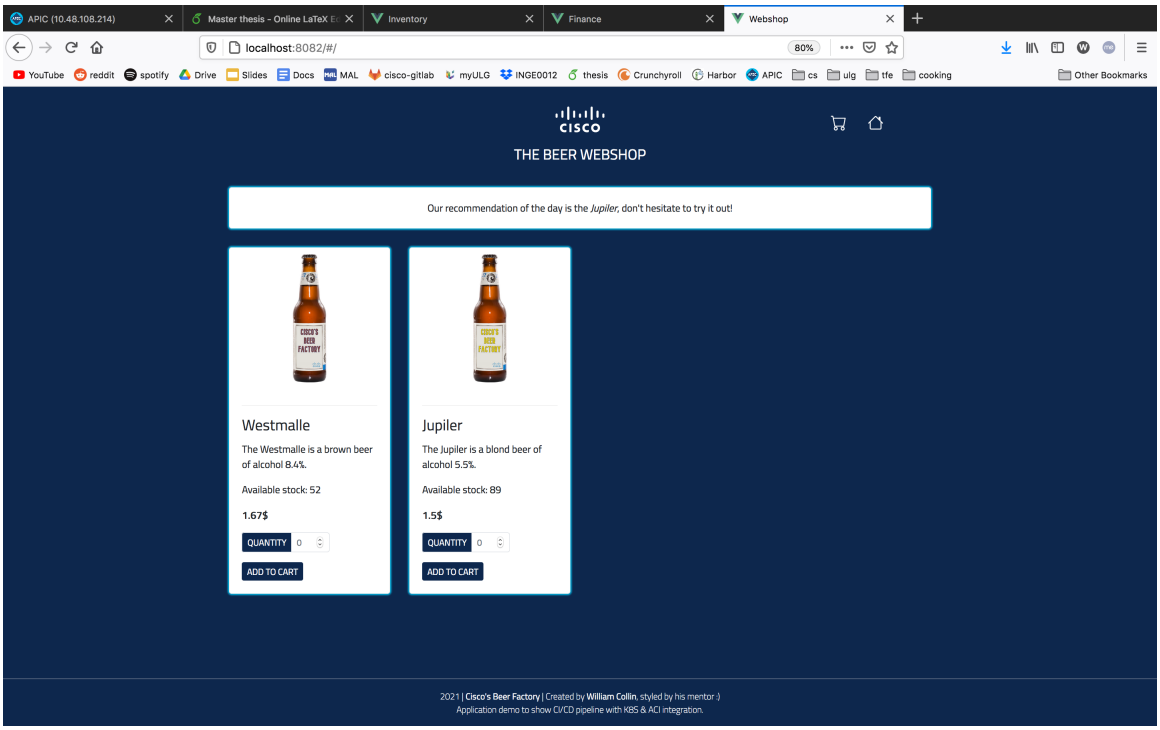

Figure 6.4: Frontend of the Webshop application

Backends & Database

Each backend microservice exposes a REST API and is built in Javascript with the Node.js Express framework. Database connection and manipulation of the data are performed thanks to Mongoose, an Object Data Modeling library for the document database MongoDB.

6.1.2 Storage

On Kubernetes

The database also runs as a Pod. Persistent storage is therefore required for, as the data would be lost if it fails.

Storage in Kubernetes is based on volumes, which are abstract storage unit that containers can use to store data. Allocating persistent storage for some Pods is done with Persistent Volumes (PV), and Persistent Volume Claims (PVC). PVs are pieces of storage allocated to the cluster that can be used by Pods, and PVCs are requests for storage. For example, the inventory-db Pod may request 1GB of storage from a PV that has a capacity of 10GB. A Statically provisioned *hostPath* Persistent Volume is used in the project. A hostPath PV uses a directory on the K8s node to simulate network-attached storage. Therefore, the database Pod must always run in the same node.

On Virtual Machine

since the database application runs as a container, persistent storage is performed with a Docker volume.

6.1.3 Network configuration

There are two application profiles: tbf-internal for the inventory and finance applications, and tbf-external for the webshop application.

On kubernetes, Deployment isolation is used so each application profile is composed of multiple EPGs, one per microservice. And, these EPGs are all linked to the Kubernetes VMM domain and placed in the bridge domain kube-pod-bd, which is the bridge domain used for all Pods created by the developers.

It is similar for the deployment on virtual machines, except that a bridge domain and a subnet had to be created for the applications.

ACI contracts are needed for allowing communication between the EPGs. All Express applications listen on port 3000, therefore, the same filter could be reused by the contracts.

Contracts to EPGs relationships for the communication between the microservices are shown in Figure 6.5, where contracts are represented by the circles and EPGs by the rectangles. A green line means "provides", whereas an orange line means "consumes".

Some contracts are also needed for the deployment on virtual machines or in Kubernetes, such as the contract allowing Ansible to communicate with the virtual machines, the HTTP contracts for accessing the web applications, or the DNS contract used by the Pods in Kubernetes since they use the domain name of the K8s Services for inter-pod communication. Those contracts are not shown so as not to overload the diagram.

Additional network constructs are also necessary for the ACI-SDA integration, but will be detailed in a further section.

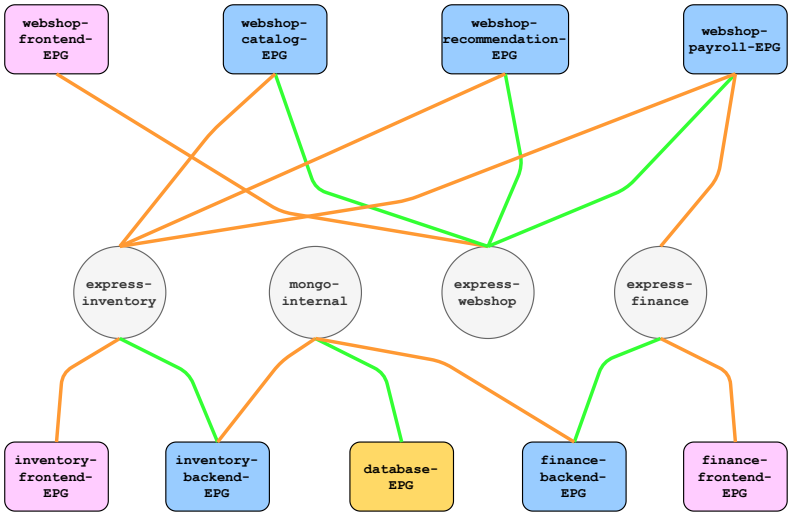

Figure 6.5: Contract Relationships of The Cisco's Beer Factory Microservices

Creating a single contract for all Express traffic would have worked, but communication between some Pods would have been allowed when they should not be able to communicate with each other. For instance, traffic from the finance-frontend Pod to the inventory-backend Pod would have been permitted with this approach.

Loadbalancing the Webshop Frontend

We decided to insert a loadbalancer for the Nginx server of the webshop application. As a result, three virtual machines implements the server of the webshop. The loadbalancer also runs as a VM and is implemented by HAproxy[22].

To steer the traffic to HAproxy upon a request from a client from the Internet to the webshop, a service graph is applied to the contract linking the clients to the webshop servers (http-lb-external). The clients no longer use the IP addresses of the VMs, but the Virtual IP of the loadbalancer to reach the application. As we can see on Figure 6.6, HAproxy performs SNAT and DNAT.

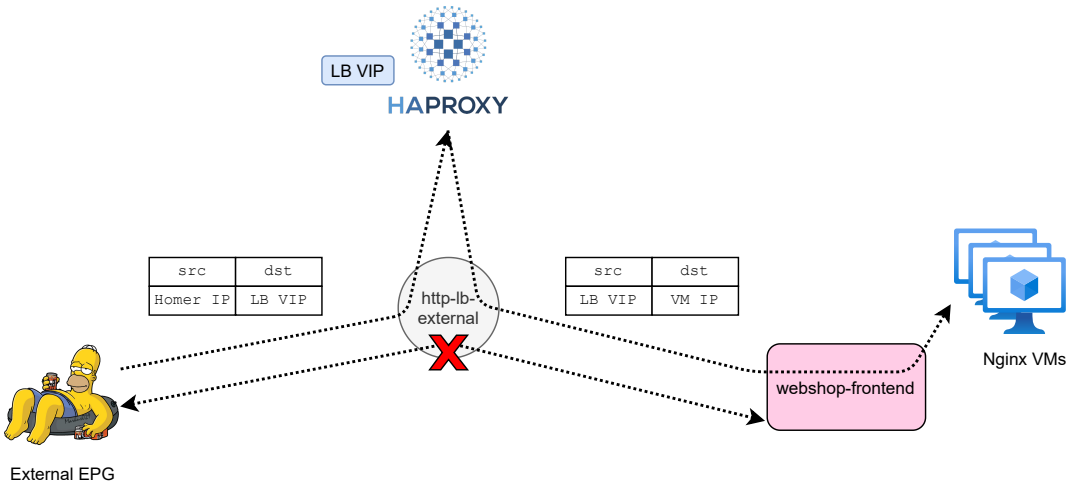

Figure 6.6: Loadbalancing the Webshop Frontend with HAproxy and PBR

6.1.4 Query Example

Here is, for illustration purposes, an example of the messages exchanged when a retail employee opens the inventory application. First, the web document is retrieved from the Nginx server (1). Second, an API call is performed upon the creation of the web document to retrieve the beers stored in the database (2). As we can see, the API call is received on the Nginx API gateway, which proxies the request to the corresponding backend microservice querying the database. Then, the beers stored in the JSON format are sent back to the client. This is shown on Figure 6.7

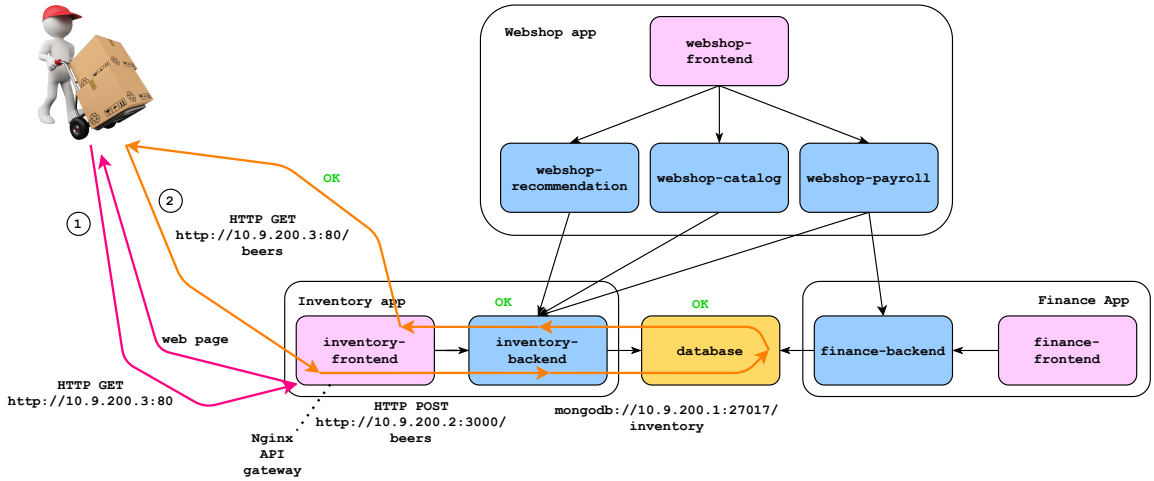

Figure 6.7: Opening the Inventory Application

6.2 Terraforming ACI

Terraform has been my tool of choice for deploying configuration. How it works and why people use it will be briefly explained, followed by a section describing how ACI is configured using this tool.

6.2.1 Automating Configuration deployment

The First Attempt

Now that the basics of Terraform are covered, it is a good time to discuss how I first envisioned IaC in this project.

Not going to lie, I was personally a bit brainwashed by the messaging of marketing people about how ACI brings developers and network engineers together, and the whole DevOps religion.

So my first idea was to follow an "application-centric" approach, where all requirements are defined by the application themselves. For example, an application deployed needs an EPG, resides in a bridge domain, can consume contracts, lies behind a K8s Service, etc. Therefore, ACI and Kubernetes resources were declared in a single Terraform module.

Although it is a nice idea on paper, such an approach is not going to work in the real world for multiple reasons. First, it would not scale well as it would lose flexibility as it grows. Second, thanks to Wouter's insights, I've learned that employees are working in silos and there is no chance that the network team would allow developers to even touch ACI, and that is quite fortunate when I think about it.

As a result, automation in the Kubernetes and the ACI domain is fully segregated. How it is performed in ACI is explained in the next section.

Project Structure

In this sub-section, the structure of the Terraform code base to configure ACI is described.

Terraform modules are needed to avoid duplication of code. When one looks at the panes shown in the APIC GUI when a tenant has been selected, he can see 5 configuration panes: application profiles, networking, contracts, policies, and services. Therefore, it makes sense to build one Terraform module per pane. For example, the application-profiles module can create applications-profiles, EPGs, link these EPGs to VMM domains, and make them consume or provide contracts.

The good news is that Terraform modules do not have to be in the same project folder as the configuration files that call them. They can be located for instance in a different git repository, which is interesting because multiple branches can be created for the development of new features. And, when calling a module residing in a remote git repository, the branch, or release can be specified.

So far, we have one git repository (ACI-modules) that will act as a sort of module library, and another git repository (ACI-live) containing Terraform files that will perform module calls for creating resources in the real world. As a matter of safety and making the code base manageable, it is convenient to have a Terraform state file per ACI tenant. Therefore, ACI-live is composed of several root folders, one per tenant. The pseudo-architecture of the Terraform code base is shown below in Figure 6.8 (not all files are displayed so as not to overload the diagram). In our case, there are two production tenants (the first one for the applications running on the virtual machines), renamed "TBF-virtual-machines" and "TBF-k8s-cluster" in the diagram for clarity purposes. The last one is a development tenant.

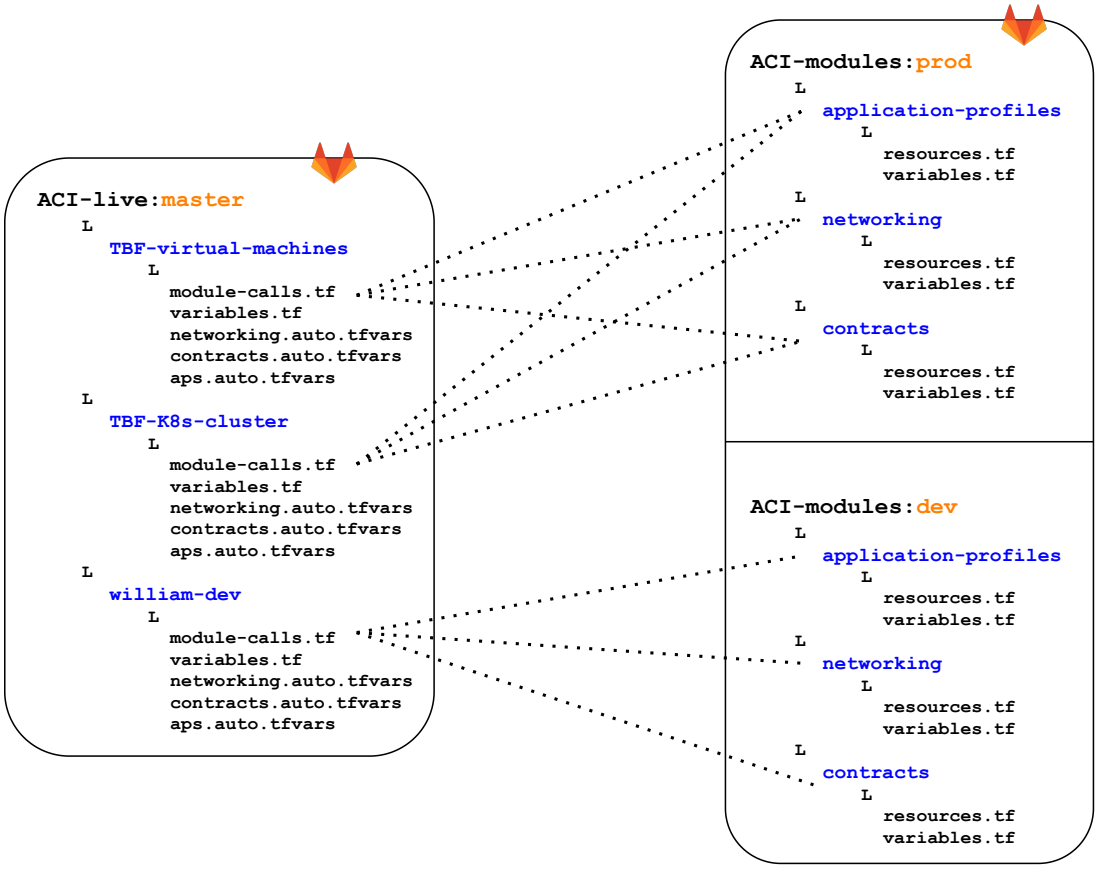

Figure 6.8: ACI-live and ACI-modules repositories

Working with such a project structure, the network team can develop new modules in the development branch and test them in a test tenant. Once the changes have been validated, they can perform a merge request from the test branch to the production branch of the ACI-modules repository.

As we can see in Figure 6.8, each tenant folder contains multiple variable files: variables.tf contains the variable declarations, the others (*.tfvars) contain the definitions. The variables in these files are used as module arguments. As such, the network team can easily reason about the state of the infrastructure by looking at these files.

In most cases, the variables are either lists or dictionaries as we want to be able to provision multiple resources of the same types (application profiles, EPGs, contracts, etc.) within the same tenant.

Flexibility Demonstration

The tricky part was building something easily usable, flexible and that integrates well with resources already created manually or by another system such as the Cisco Container Platform. For instance, the TBF-virtual-machines tenant in the diagram is actually Wouter's tenant and contains resources already created manually such as filters, a VRF, an L3out, EPGs (Gitlab) that I had to use in Terraform. Fortunately, it is feasible to fetch remote resources in Terraform and to integrate them in the code. An example is provided below to provide a glimpse of what it is possible to configure. In particular, this example allows the communication between Ansible running in Gitlab and a virtual machine running one of the application (inventory-backend).

This snippet (6.1) creates a bridge domain (the-beer-factory), places it in the Wouter's VRF, and associates it to an existing L3out. Then, a subnet is created for the bridge domain.

```
vrfs = {
    Wouter = {
        exists = "true"
        tenant = "Wouter"
    }
}

l3outs = {
    L3Out-Wouter = {
        tenant = "Wouter"
    }
}

bds = {
    the-beer-factory = {
        vrf = "Wouter"
        l3out = "L3Out-Wouter"
    }
}

subnets = {
    the-beer-factory = {
        gw = "10.9.200.254/24"
        bd = "the-beer-factory"
        scope = ["public"]
    }
}
```

Listing 6.1: networking.auto.tfvars example

Next (Listing 6.2), a contract (tbf-services) and a contract subject (tbf-service-subj) are created. The contract subject reuses two existing filters (icmp and ssh) and a newly created filter (http).

```
filters = {
    http = {}
    icmp = {
        exists = "true"
    }
    ssh = {
        exists = "true"
    }
}

filter-entries = {
    http = {
        filter = "http"
        ethertype = "ipv4"
        ip_protocol = "tcp"
        destination_port_from: 80
        destination_port_to: 80
    },
}

contracts = {
    tbf-services = {}
}

contract_subjects = {
    tbf-services-subj = {
        contract = "tbf-services"
        filters = ["http", "icmp", "ssh"]
    }
}
```

Listing 6.2: contracts.auto.tfvars example

Finally, an application profile (tbf-internal) is created within the existing tenant of Wouter, and an EPG named inventory-backend is provisioned within this application-profile. This EPG lies in a bridge domain (the-beer-factory) and is linked to an existing VMM domain (HX-VMM). Finally, contracts (tbf-services) are attached to the manually created EPG DevOps, in which Gitlab resides. both EPGs provide and consume this contract. It is shown in listing 6.3. Some arguments are optional, for instance, the user can create a bridge domain without specifying an L3out.

```
app_profiles = {
    Applications = {
        exists = "true"
    }
    tbf-internal = {}
}

vmms = {
    HX-VMM = "uni/vmmp-VMware"
}

epgs = {
    DevOps = {
        exists = "true"
        ap = "Applications"
    }
    inventory-backend = {
        ap = "tbf-internal"
        bd = "the-beer-factory"
        vmm = "HX-VMM"
    }
}

epg_to_contract = {
    DevOps-tbf-services-cs = {
        epg = "DevOps"
        contract = "tbf-services"
        type = "consumer"
    },
    DevOps-tbf-services-pv = {
        epg = "DevOps"
        contract = "tbf-services"
        type = "provider"
    },

    inventory-frontend-tbf-services-cs = {
        epg = "inventory-frontend"
        contract = "tbf-services"
        type = "consumer"
    },
    inventory-frontend-tbf-services-pv = {
        epg = "inventory-frontend"
        contract = "tbf-services"
        type = "provider"
    },
}
```

Listing 6.3: aps.auto.tfvars example

Continuous Delivery

As said earlier, all the IT operations are performed from Gitlab. As such, the pipeline of ACI-live is triggered upon when a new configuration is pushed and consists of 3 stages: A validation phase verifying that the configurations is valid, a plan phase that will output an execution plan used by the last apply phase.

I did not have to resort crafting my own Terraform Docker image as one already exists[17]. The pipeline is presented below in Figure 6.9

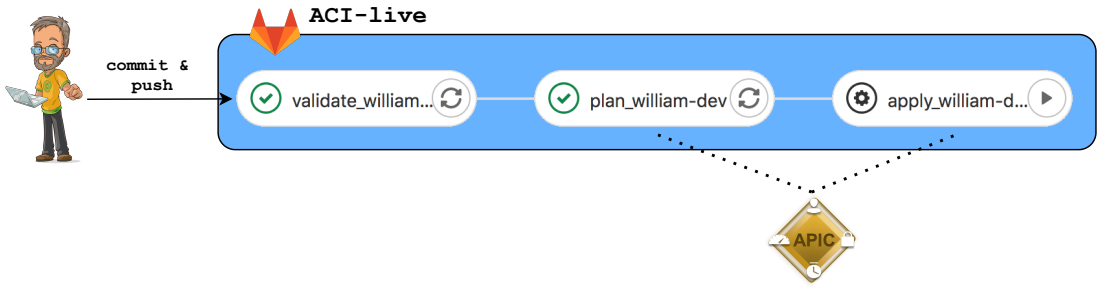

Figure 6.9: Pipeline of ACI-live

Assuming that there are 2 tenants defined in ACI-live, it means that each tenant should have its own Terraform pipeline since there is a state file per tenant. However, making a change in tenant "A" should not trigger the pipeline of tenant "B". Fortunately, Gitlab comes with rules specifying that a job is to be run only if there has been a change in a certain folder.

6.3 Deploying The Applications to Kubernetes

This section is about the deployment of microservices in Kubernetes following the DevOps ideas. Multiple ways have been tried out; the first one using Terraform and the second one using Helm charts. Prior to delving into these topics, a short subsection on how continuous integration is performed is given.

6.3.1 Continuous Integration

Each microservice is associated to a repository in Gitlab, and as you might expect, a pipeline is triggered every time new code is pushed.

The first step of the pipeline is to build a new Docker image automatically and to publish it to a registry, Harbor in our case. Building a Docker image is not hard, all that needs to be provided is a Dockerfile, a document containing commands to assemble an image. The image is tagged with the 8 first characters of the commit hash.

Building a Docker image inside a pipeline when the Docker executor is used requires to use "Docker in Docker" or more commonly called dind. Dind makes things really convenient for developers, however, it comes with security risks and should be used with caution in real environments[44][46].

Note that the first pipeline job of the frontend microservices is to build the frontend using Webpack, which outputs a folder containing the HTML and Javascript bundled together, and that can be sent (thanks to Gitlab artifacts) to the second job that will build an Nginx image using this bundle.

6.3.2 Deployment Using Terraform

The first to deploy the applications in Kubernetes was with Terraform and the Kubernetes Provider. How It was carried out is explained in this sub-section.

The Terraform project architecture is similar to what is described in Section 6.2.1. As a result, there are two git repositories: K8s-live, and K8s-modules, and the Terraform operations are also running inside a pipeline just as ACI-live.

Each root folder in K8s-live represents a namespace in the cluster, as such, there is a namespace in which all the microservices of Cisco's Beer Factory are deployed.

Concerning the modules, there is a module for deploying the applications, another one for the databases along with the persistent volumes, and the last one for the K8s Services. The project structure is presented in Figure 6.10.

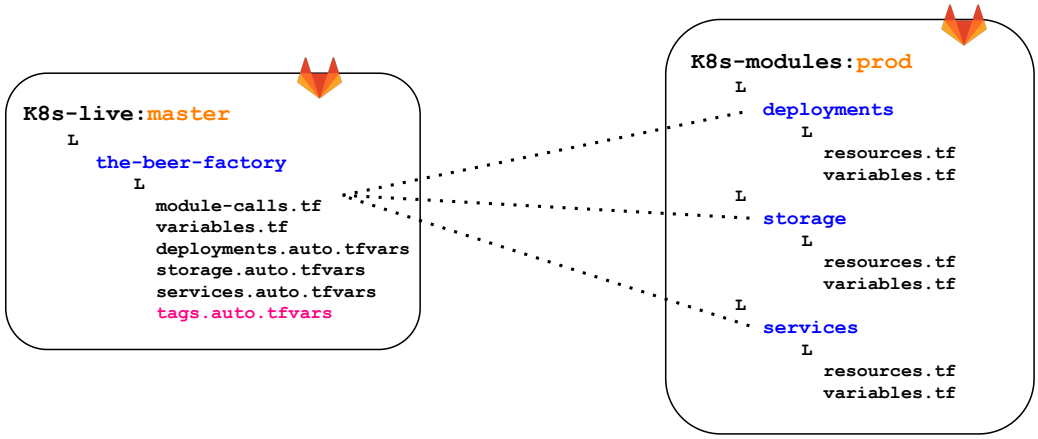

Figure 6.10: K8s-live and K8s-modules Repositories

You should notice a variable file named *tags.auto.tfvars* on the diagram above, his purpose is to map Docker image names, to tags. The existence of such a file is a consequence of how the Terraform plan phase is working and why a new image tag is used every time an image is built. Indeed, if a new image is pushed with exactly the same tag in the registry (latest for instance) and that a Deployment for this application is already running in the cluster, problems arise; the desired state is already reached since the configuration remains unchanged.

As such, the tag of the image in the Terraform configuration has to be changed every time a new image is built, which is not really convenient but works nonetheless.

The CICD workflow is illustrated in Figure 6.11.

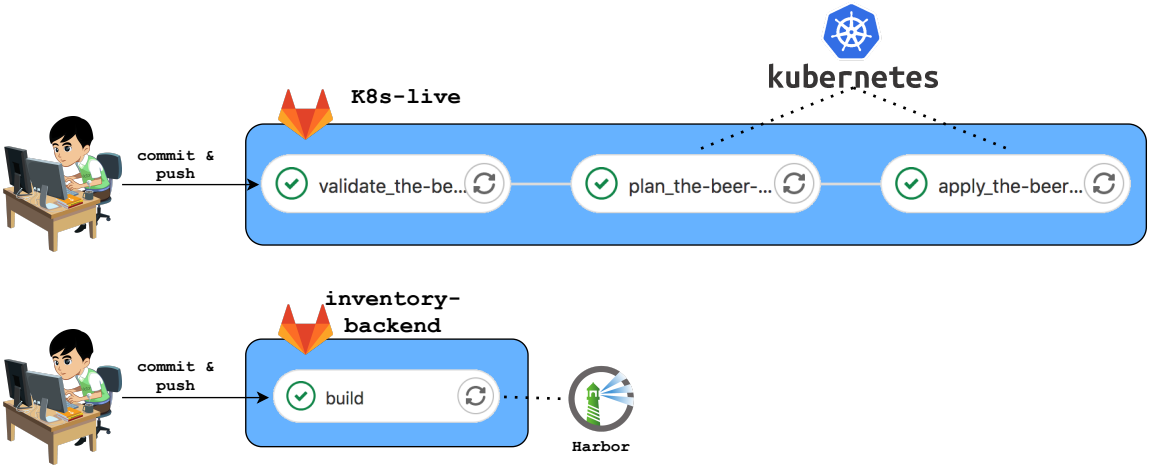

Figure 6.11: K8s CICD workflow K8s with Terraform

6.3.3 Deployment Using Helm Charts

Deploying applications from a centralized location, as is the case in the previous approach has its perks and downsides. One one hand, it provides visibility as the Kubernetes captain has a clear view of the whole system. On the other hand, it makes continuous deployment a bit trickier.

Each microservice is directly deployed in the cluster using a Helm client[26]. Of course, using such a client in a pipeline job requires some configurations. Fortunately, configurations can be baked into the Docker image used for the deployment, hence removing the hassle of reconfiguring it every time in the pipeline file. In the end, an IaC Docker image packing tools (Helm, kubectl, Python, Ansible, and Packer) preconfigured for automating the infrastructure in the lab has been created.

The CICD workflow is presented below in Figure 6.12. In this case, the repository contains a Helm chart used by the Helm client to deploy the app to K8s.

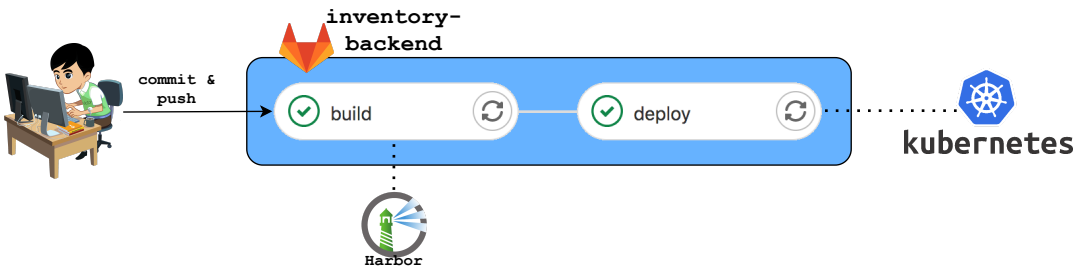

Figure 6.12: K8s CICD workflow with Helm Charts

6.4 Deploying The Applications to Virtual Machines

This section is dedicated to the template creation, provisioning, and configuration of virtual machines. The whole deployment process, from the creation of the VM template, provisioning of the VMs, to the configuration of the VMs is automated.

The pipeline steps are shown in Figure 6.13. In a few words, a virtual machine template to run our applications is firstly created with Packer. Then, the virtual machines are provisioned in vSphere using Terraform. Lastly, Ansible instructs each VM to run a containerized application.

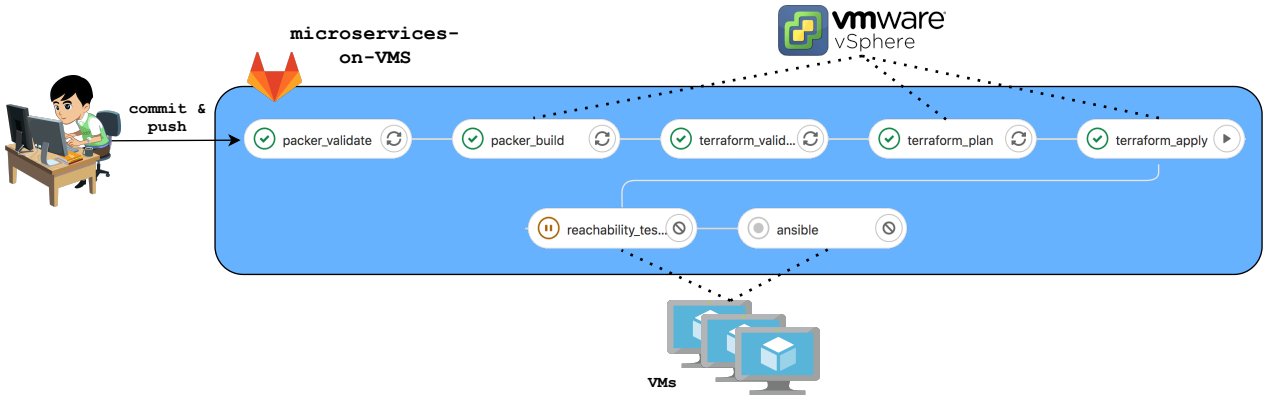

Figure 6.13: From Kubernetes to VMs pipeline

For illustration purposes, the structure of the repository is shown in Figure 6.14. The next subsections describe these steps in more details.

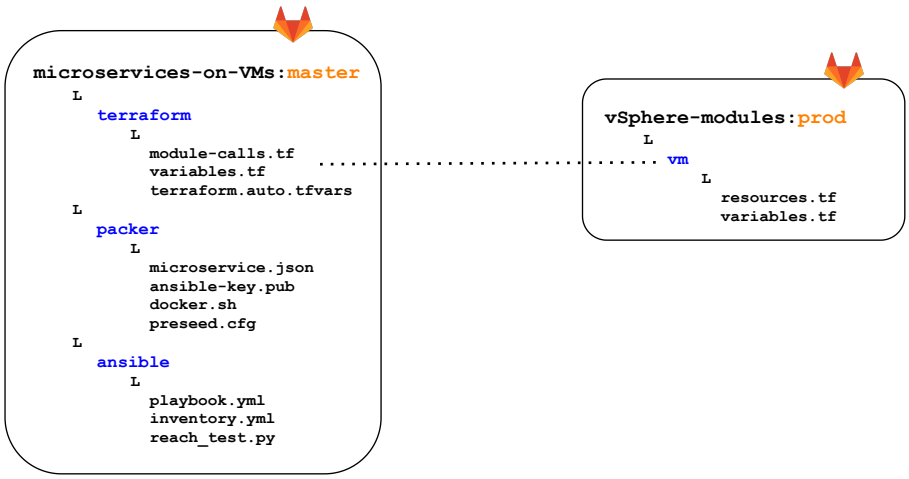

Figure 6.14: microservices-on-Vms Repository

Because more steps and tools are involved, Figure 6.15 gives more insights on the workflow, and will be used as a basis to describe the deployment process.

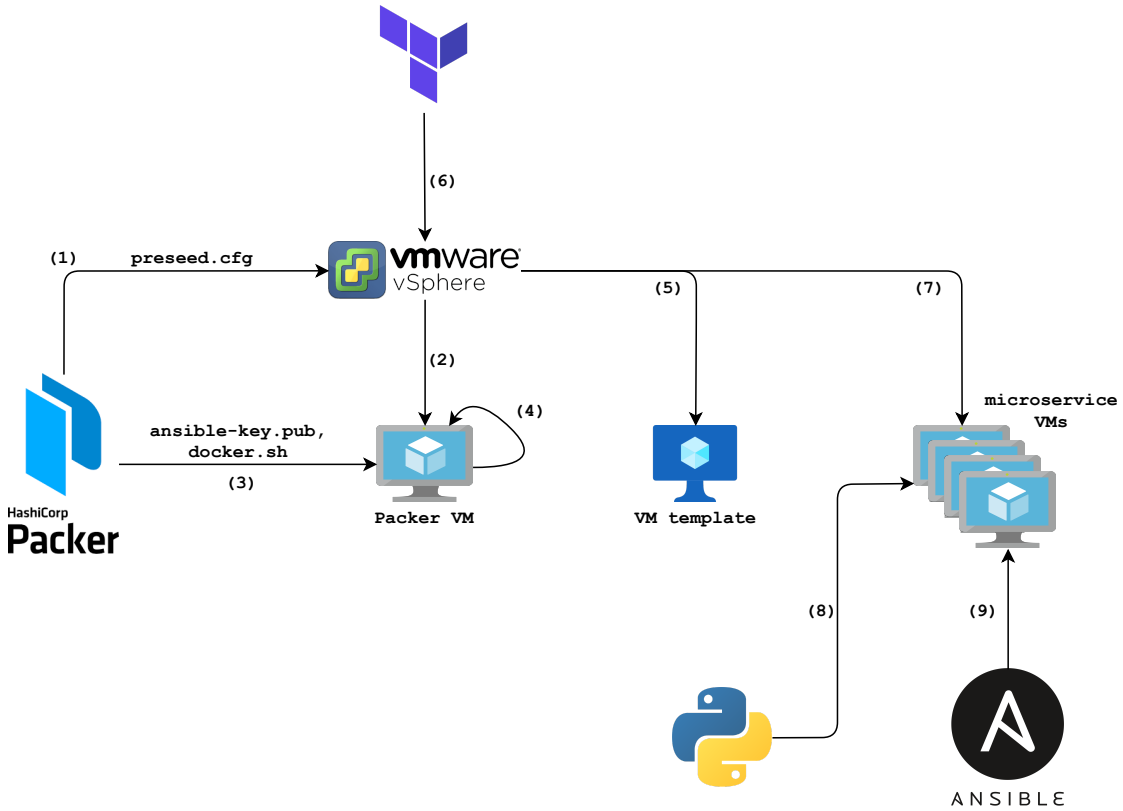

Figure 6.15: Automating The Provisioning and Configuration of Virtual Machines

6.4.1 Virtual Machine Templates Creation with Packer

Our goal is to create a master template to run our applications in the vSphere environment, the vSphere builder was employed, which allows to create a VM template from a source ISO file.

The Packer workflow has a validate phase, which lints makes sure that the configuration is valid, and a building phase depicted in Figure 6.15.

It starts by instructing vCenter (1) to provision a virtual machine with the specified ISO file used to install the operating system (2). Quite a lot of information needs to be provided in the building section of the configuration file, such as where the virtual machine is to be run, compute resources needed, the network.

A configuration file (preseed.cgf) is also provided to vCenter to install the guest OS without any manual intervention. Information includes the IP address of the virtual machine, layout keyboard, software packages needed, and so on.

Once the OS is installed, the guest VM is restarted and ready to be edited over SSH (3). The File Provisioner was used to upload a public key to the VM and the shell Provisioner to run a script on the virtual machine, which configures SSH for the configuration step done later with Ansible, and installs Docker (docker.sh) (4).

When the configuration of the guest OS is performed, the VM is shut down, converted to a template and, then deleted (5).

6.4.2 Provisioning

Once the golden image is created, Terraform applies the changes to the vSphere environment (6), resulting in a deployment of a dozen of VMs from the template created by Packer (7) (see Figure 6.15).

It is in this step that guest OS customization occurs, in particular, each virtual machine is placed in the corresponding network (distributed port-group) pushed by ACI when an EPG is created, and the network interfaces are configured accordingly. Other parameters can also be set in this step, such as the disk or RAM size.

6.4.3 Configuration Management with Ansible

The final step in this migration process is to run a container in each virtual machine. As the simultaneous configuration of multiple devices is a relevant use case for Ansible, this tool was used for this task. This subsection starts with a short explanation of how Ansible works, and is followed on how the task is performed.

Ansible setup and Configuration Deployment

To configure our Ubuntu servers, Ansible needs to be able to connect to them via SSH. Therefore, it is better to check that the VMs are reachable beforehand. To this end, the script reach_test.py pings all the hosts defined in the inventory file, and the job is successful when all the machines are reachable (8). This preliminary step also holds another purpose, it makes sure that ACI knows how to reach these devices prior to attempting SSH connections to them. Indeed, ARP gleaning has to be performed since those machines are silent hosts and are thus unknown to the ACI fabric after the provisioning process, thus it takes a few packets before ACI knows the exact location of these endpoints. Not performing this lookup step can result in some failed SSH connection attempt to the hosts the first time.

SSH clients can either authenticate to servers using a password or SSH keys. The second option was preferred because Ansible must be able to seamlessly connect to the virtual machines and for security reasons. So, a pair of keys consisting of a private and a public key was generated with Ed25519, an elliptic curve signature scheme[37]. The public key was uploaded to the VM run by Packer, which was then converted into a template, therefore, each virtual machine provisioned by Terraform possesses the public key. While the private key is baked into the container on which Ansible runs. This is relatively safe as long as the Docker image of the container is not shared. While putting the private key into the Docker image is the most convenient option, it is not the only one. I could have for instance written some instructions in the pipeline file to copy the private key to the container every time Ansible is to be run.

Finally, Ansible running in the IaC container starts the playbook, which configures Docker to use the Harbor registry, and then runs the corresponding container and also creates a Docker volume for each database container (9). Ansible can be instructed to run a different container in each VM thanks to variables defined in the inventory file.

There are probably existing Ansible modules to manage containers on hosts, but I decided to keep things simple and just write shell commands in the Ansible playbook.

6.5 Integration With Webex Teams

The Cisco's Beer Factory has two teams, a network team and a development team, and each team has its Webex Teams room. When developers need some network resources to run their applications, they open an issue on the ACI-live git repository. Then, the network team can comment on the ticket, make changes to the network if the request is reasonable, and then close it. All of these actions send notifications to the corresponding Webex rooms.

First, rooms are needed. Fortunately, Webex provides an API that can be used to create rooms, provided the developer has a token that can be retrieved from the Webex developers' website. Notification messages are sent by a bot that can be registered and added to rooms. Thus, the rooms were created and setup using a script in Python.

The next necessary component is Gitlab webhooks, which are API calls that are automatically triggered and sent to a server of our choice when a specific event occurs. On Gitlab, events are classified into different categories, pipeline and issue events, among others. Using webhooks is quite simple as all the developer needs to do is select the type of event they want to monitor and provide the destination of the API calls triggered by the events.

The final components are web servers that listen for API calls sent by Gitlab webhooks, and will post messages to the rooms using the Webex bot. All it needs are the room IDs and the bot token.

In our case, there are two web servers written in Python using the Flask framework, the first is in charge of ticket notifications, and the second posts messages related to pipeline events. Both web servers are deployed in Kubernetes with Helm charts.

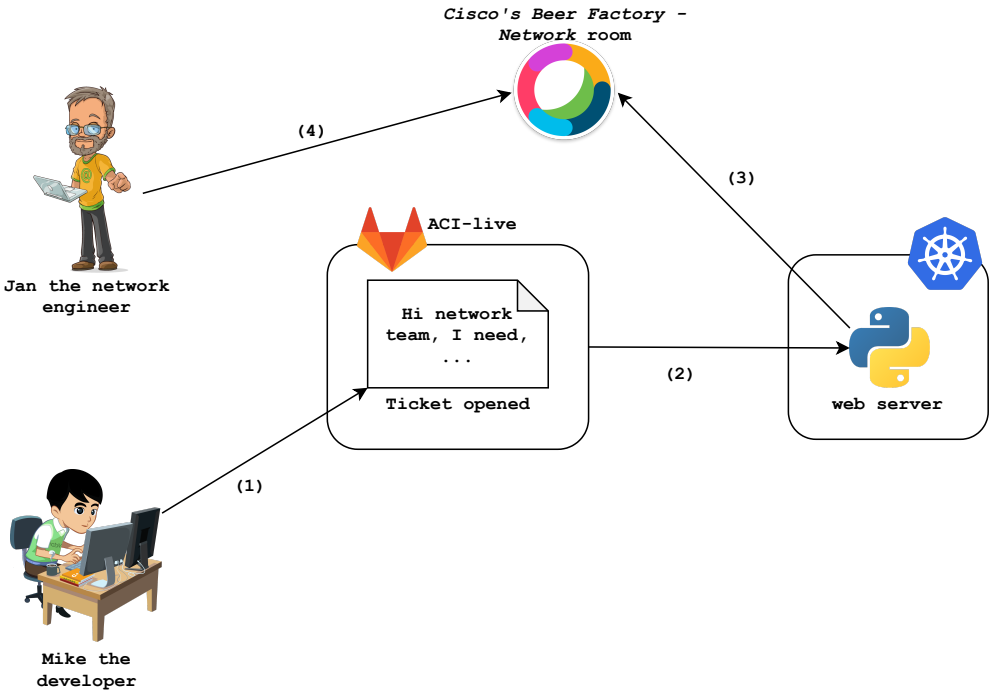

Figure 6.16: Gitlab Webhooks and Notifications to Webex Teams Rooms

The workflow is shown in Figure 6.16. First, Mike the developer opens a ticket on the ACI-live repository (1), it triggers a webhook to the web server running in Kubernetes, which instructs the bot to post a message on the room of the network engineers (3), and finally, Jan the network engineer receives and reads the notification in the room (4). Similarly, if Jan comments on the tickets, a notifications is sent to the developer rooms.

Sending pipeline notifications is based on the same principles. An example of pipeline notifications is shown below in Figure 6.17 for illustration purposes.

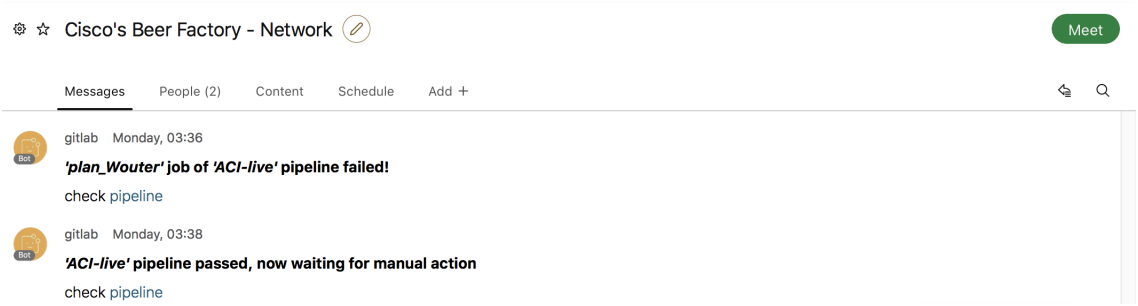

Figure 6.17: Pipeline Notifications

Chapter 7

Software-Defined Access

This chapter starts by introducing Cisco Software-Defined Access (SDA), describes its main components, explains how packets are routed and forwarded within SDA, and finally shows how users are authenticated and authorized in the network.

7.1 Software-Defined Access Overview

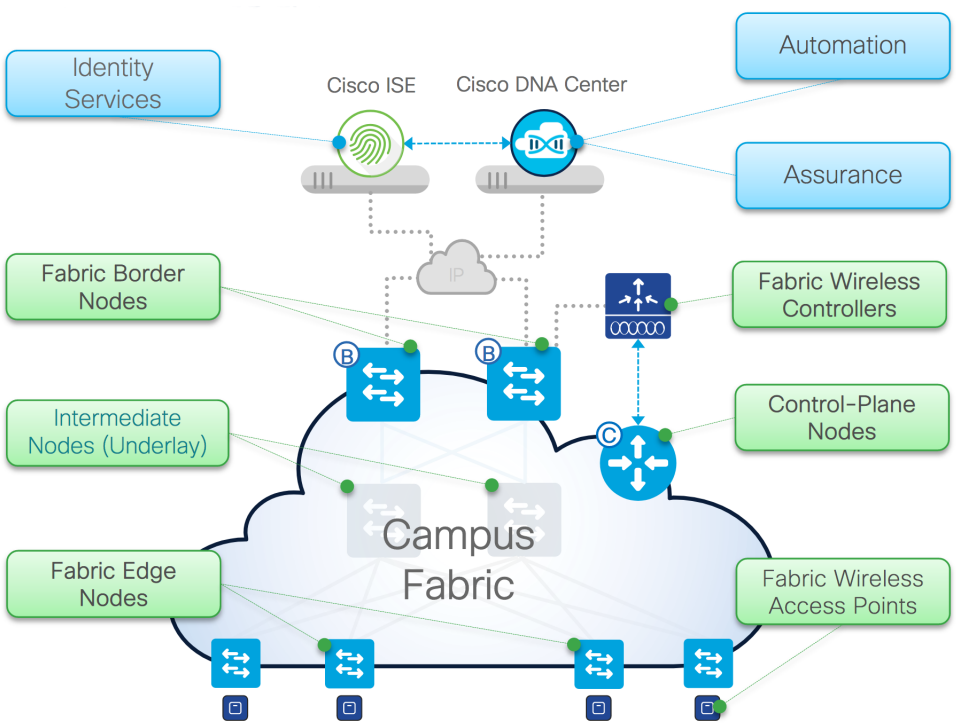

Figure 7.1: Cisco SDA overview, by Shawn Wargo in Cisco SD-Access - Solution Fundamentals - DGTL-BRKCRS-2810, p. 19, 2019[70]

Software-Defined Access is the solution to build campus networks using an SDN approach, where *access* refers to the endpoints (users, printers, cameras, etc.) connecting to the network through wired or wireless connections. In a few words, SDA automates the configuration and deployment of the enterprise network using policies, enables dynamic host mobility for wired and wireless endpoints, and performs identity-based network segmentation.

This solution uses a centralized controller named DNA center and includes a network fabric composed of an overlay and an underlay network leveraging the Locator/ID Separation

Protocol (LISP)[35] for the control plane, and VXLAN[73] for the data plane.

As we will see in further sections, using LISP in the corporate network has many perks such as improving the mobility of users within the network.

SDA also goes hand in hand with Cisco Identification Service Engine (ISE)[56], a security policy management platform providing network access control (NAC), dealing with AAA, which refers to the authentication, authorization, and accounting of users (i.e. who you are, what you can do and what you have done) using, for instance, 802.1X[50] and Cisco TrustSec[59]. ISE is not a new product, it already existed when SDA came out.

A high-level diagram of SD-Access is presented in Figure 7.1. The next sections explain in more detail the components and concepts of SDA.

7.2 SDA Main Components

7.2.1 DNA Center

The DNA Center controller provides lots of mechanisms to manage the enterprise network. It provides a single-pane-of-glass GUI that allows the administrators to configure the network using policies and have full visibility of the network by providing site layouts, topology maps (see Figure 7.2), or a device inventory.

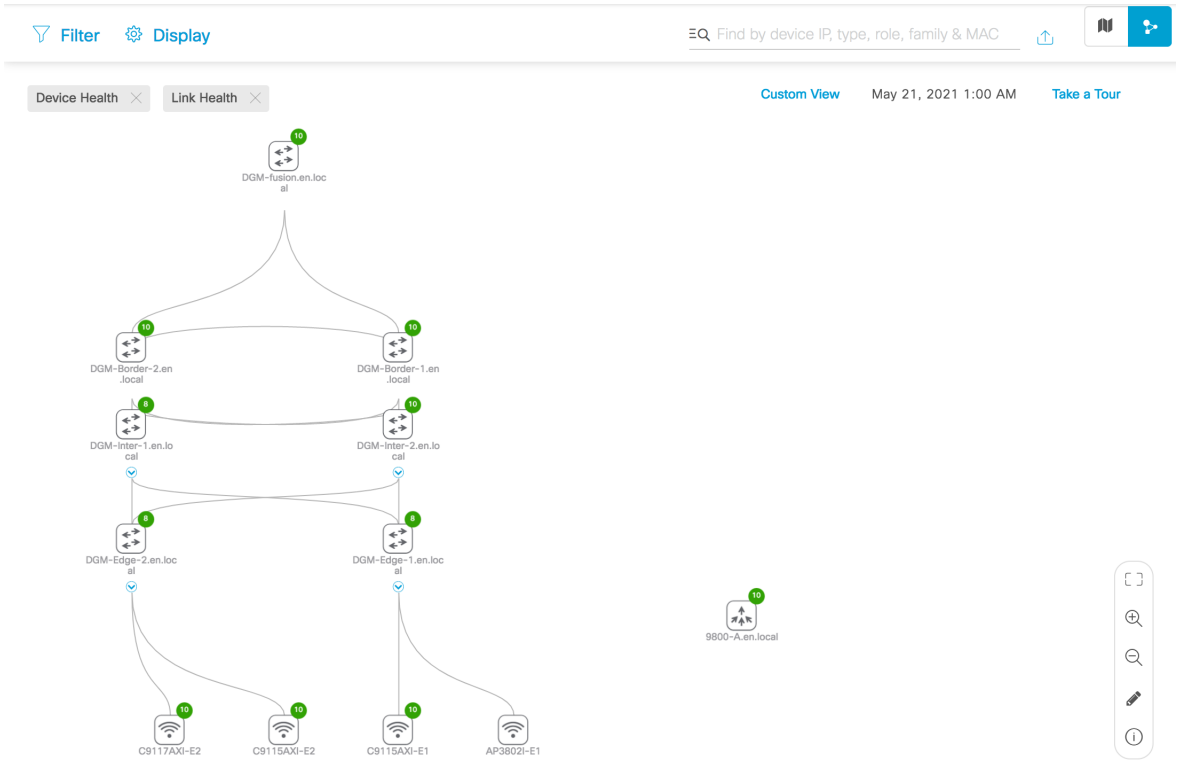

Figure 7.2: DNA Center Topology

DNA Center also speeds up the onboarding of new devices, and the deployment of configuration through templates called network profiles leveraging network policies.

As most of the controllers nowadays, it features a northbound REST API that engineers can use to deploy configurations on SDA. Just like ACI, SDA does not completely separate

the control plane from the data plane of the devices it manages, however, it is not using Opflex. Instead, DNA Center supports several southbound APIs to directly communicate with the devices it manages, it includes NETCONF[4] and RESTCONF[71]. The controller also allows you to connect via SSH or telnet into older network devices directly from its graphical user interface.

On top of that, it offers assurance mechanisms, and metrics to analyze the health of the network, which aims at automating the process of identifying and resolving network issues.

7.2.2 Identification Service Engine

ISE is the platform provisioning user access policies to the network devices and provides information over the endpoints connected to the network. For instance, it can be used to check whether endpoints connected to the network are compliant with the security policies of the company (e.g. devices without a given antivirus installed are not granted access to the network). ISE provides multiple ways for the users to connect to the campus; web-based authentication, 802.1X, or even MAC address bypass for IOT devices. In the case of 802.1X, ISE acts as a RADIUS server.

ISE is quite a large platform, we will mainly focus on a component of ISE named Cisco TrustSec (CTS) allowing network administrators to perform network segmentation using tags.

Note that some access policies can also be created directly on DNA Center. In that case, DNA Center will pushes those policies to ISE, so DNA Center just acts as a second "GUI over ISE".

Cisco TrustSec

TrustSec makes network segmentation easier by assigning tags to traffic, and allows the network devices to treat that traffic based on source and destination tags, as opposed to source and destination IP addresses. In the case of Trustsec, the tags are called Security/Scalable Group Tags (SGT).

In TrustSec, once endpoints are authenticated, they are classified into a group using an SGT, which is a 16-bit value assigned to the user or endpoint's session upon login. Assuming that we are managing the network of a company, we can imagine an SGT for the human resources employees, another one for the engineering team, etc. Then, group-based access control (GBAC) can be performed based on these SGTs using Secure Group ACLs (SGACLs), which are similar to ACI contracts. In fact, the SGTs can be thought as the EPGs of ACI. The different SGTs are defined within a given Virtual Network (VN).

In addition, different VRFs named Virtual Networks can also be provisioned in SDA, hence adding another layer of segmentation. It provides complete isolation between traffic and devices in one VN from that of other VNs. For instance, it can be useful to create a VN dedicated to the employees, and another one for the IOT devices (cameras, etc.).

In SDA, segmenting the network based on Virtual Networks (resp. SGTs) is called macro-segmentation (resp. micro-segmentation). This is illustrated in Figure 7.3

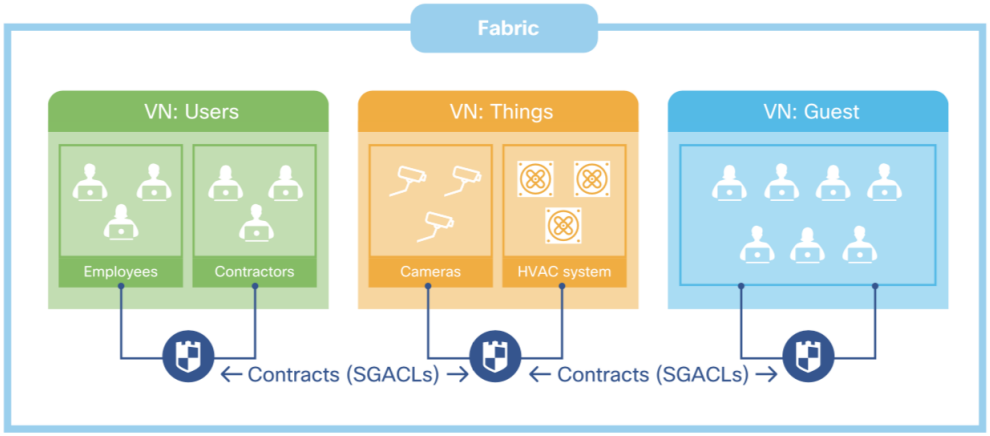

Figure 7.3: Contracts and Virtual Networks in TrustSec, Cisco Software-Defined Access Enabling intent-based networking 2nd edition, p. 121[58]

7.3 Routing and Forwarding in SDA

This section describes how routing and forwarding is performed in SDA. It starts by explaining how SDA is leveraging the LISP protocol for the control plane. After that, it explains how packets are forwarded and why it improves the mobility of the users.

7.3.1 LISP Control plane in SDA

SDA has a routed access layer and uses a combination of VXLAN and LISP (Locator/ID Separation Protocol). In traditional IP routing, an IP address has two functions: identifying the endpoint, and locate the endpoint in the network. LISP is a map and encapsulation protocol that aims to decouple the identity of an endpoint from its location. Its major role SDA is to provide a routing architecture that allows the network to track the point of attachment of every endpoint in the network.

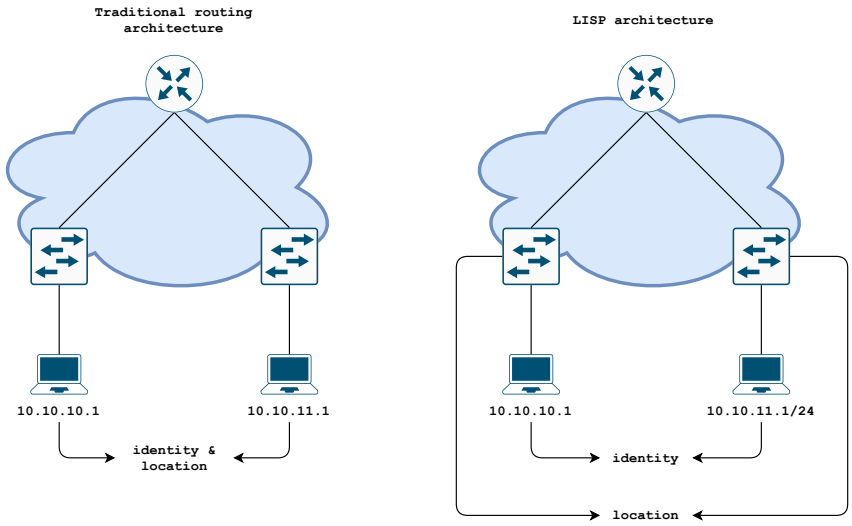

Figure 7.4: Separating the Identity from the Location

In LISP, the location is known as a Routing Locator (RLOC), and the identity refers to as an endpoint identity (EID). To track users, LISP uses a mapping database (LISP map

server) implemented by the control plane node that can be queried by other nodes. As shown in Figure 7.5 An entry in this database maps the identity of an endpoint to its location.

Figure 7.4 presents the different types of nodes in SDA. The LISP map server is implemented in the control plane node and is queried by either the fabric border nodes connecting the SDA fabric to the outside or the fabric edge nodes (FE) connecting the endpoints to the enterprise network.

In SDA, an RLOC is an advertised loopback IP address that is assigned to each fabric edge, and fabric border nodes. And, the EID consists of the IP address of the endpoint. This implies that there needs to be a registration mechanism. Indeed, upon discovering a new endpoint, the FE will register it (Map-Register message) to the control plane node, and will also cache this information.

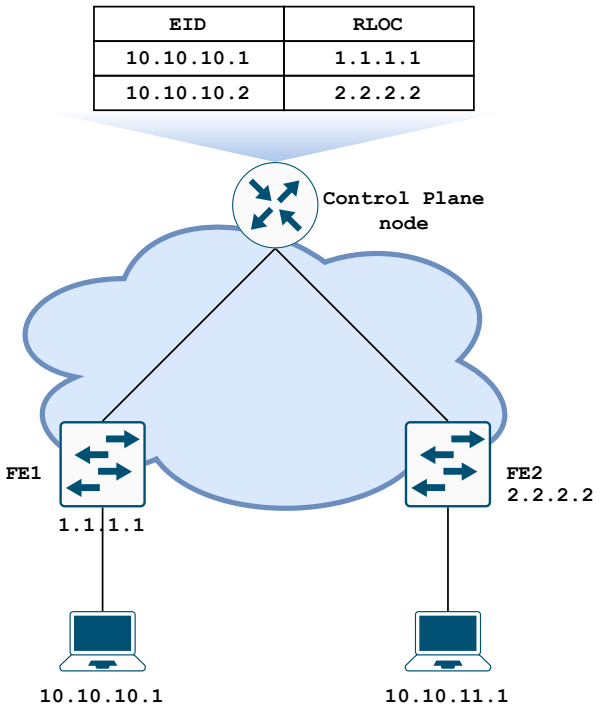

Figure 7.5: Control Plane Node Entries

Forwarding the packets within the SDA fabric is performed through a VXLAN overlay network built between the RLOCs. In fact, a FE is a VTEP and called ITR (resp. ETR) in the LISP terminology, which stands for Ingress Tunnel Router (resp. Egress Tunnel Router).

When an endpoint sends traffic to another endpoint not connected to the same FE in the SDA fabric, the packet is encapsulated in a VXLAN frame at the ingress FE (ITR). The source IP address of the VXLAN packet is the ingress RLOC, and the destination IP address is the destination RLOC. The ITR learns and caches the destination RLOC by querying (Map-Request message) the LISP map server, which either directly replies to the query to the ITR, or forwards the request to the ETR that will answer to the ITR; An ETR may also request during the registration of an EID, that a map server directly answers Map-Requests instead of forwarding them to the ETR by setting the "proxy Map-Reply" flag[13]. Both cases are illustrated in Figure 7.6. As we can see, the LISP mapping system is a bit analogous to a DNS lookup.

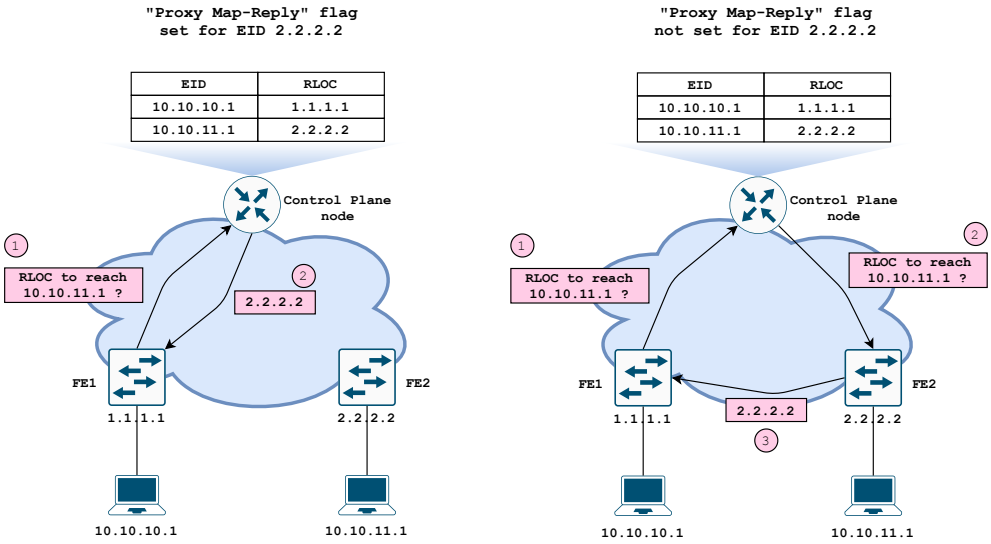

Figure 7.6: Map-Request messages in LISP

Once the VXLAN packet arrives at the destination RLOC, it is decapsulated, and the original frame is forwarded to the endpoint. The forwarding of a packet via the VXLAN overlay is shown in Figure 7.7.

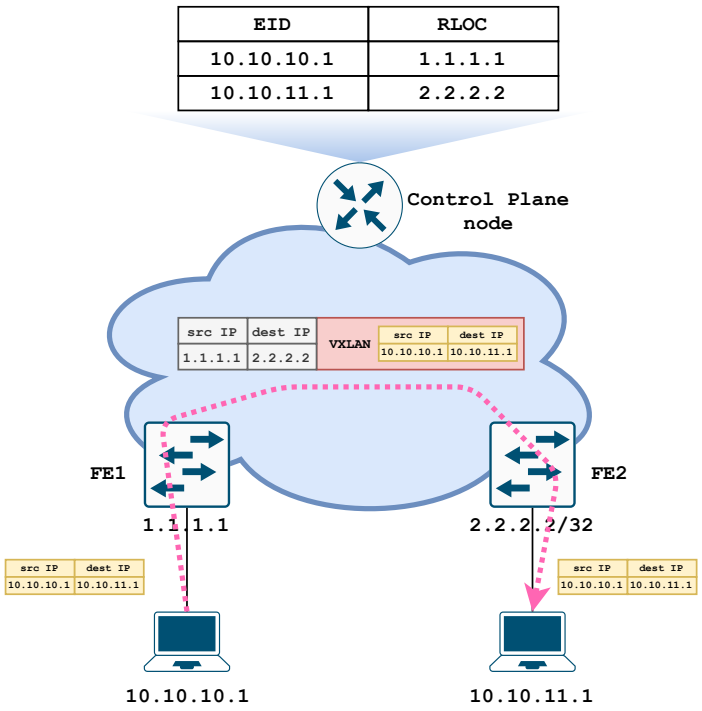

Figure 7.7: VXLAN Encapsulation and Decapsulation in the Fabric Edge Nodes in SDA

As we have seen, SDA allows the nodes of the fabric to only learn the topology information that is relevant to them by providing a LISP map server that is used by the FEs to query the location of the endpoints, whereas nodes in a traditional IP routing infrastructure have to learn and store the entire topology of the fabric at all times, and thus improve the scaling of the network.

Routing in the underlay network is, however, performed with IS-IS.

7.3.2 User Mobility

One of the major advantage of using a LISP control plane is that it improves the mobility of the endpoints within the network.

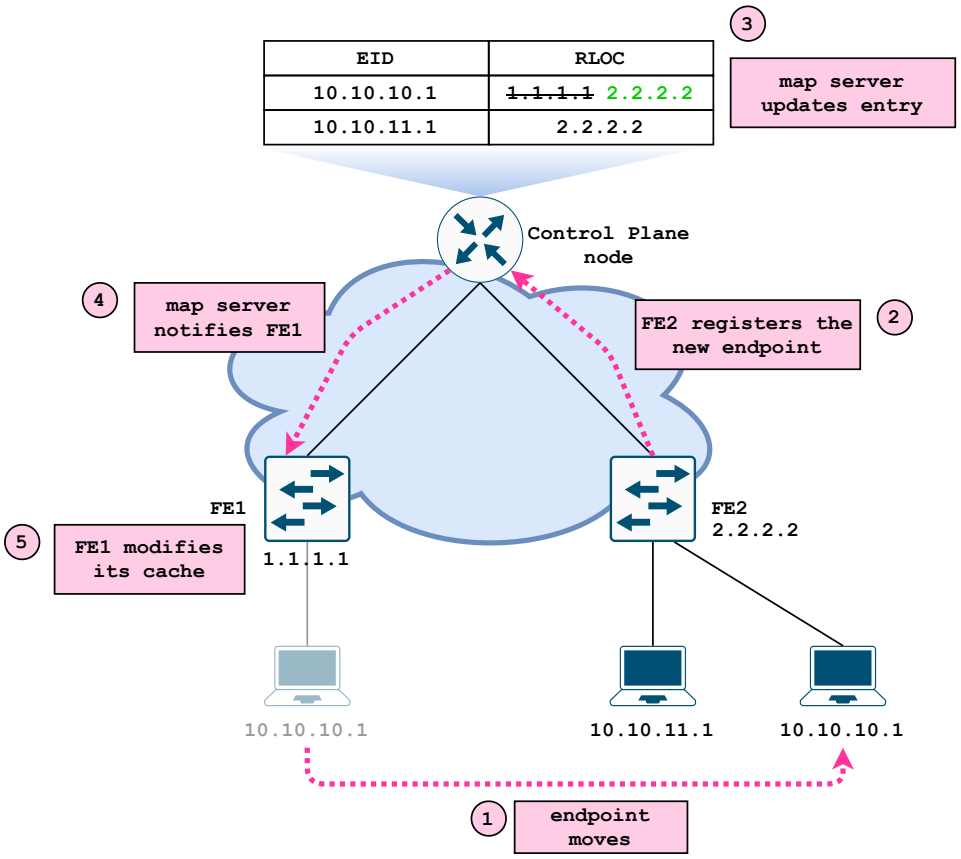

Figure 7.8: User Mobility in SDA

As we can see in Figure 7.8, if an endpoint moves to a new location (1), the fabric edge it is connected to, registers it with the map server (2). If the entry already exists, it is updated with the new RLOC (3). Then, the map server notifies the FE, to which, the endpoint was previously connected (4). Finally, the FE updates its cache with the new location (2.2.2.2) to reach the endpoint (5).

This mobility process raises an interesting question. How are the default gateways of the FEs configured? Indeed, if an endpoint changes location, he shouldn't have to reconfigure its default gateway. In SDA, all the FEs share the same virtual switch interface (SVI), which has the same IP and MAC addresses. An SVI represents a logical L3 interface on a switch and was introduced for inter-vlan routing. This shared SVI is used as a default gateway for the endpoints and is called an L3 anycast gateway in SDA.

7.4 Authentication and Authorization of Users

This section briefly explains the authentication and authorization of users in the enterprise network using 802.1X and ISE.

In a lot of companies, an external identity store is also employed to manage the credentials, user accounts, and so on, of the employees, Active Directory (AD) from Microsoft is one of

them and can be integrated with ISE. In particular, AD allows the administrators to also define groups for the different employees, and, ISE can map the groups defined in AD to SGTs.

Figure 7.9 gives the big picture of how a finance employee is authenticated and authorized in the network using 802.1X, ISE and AD.

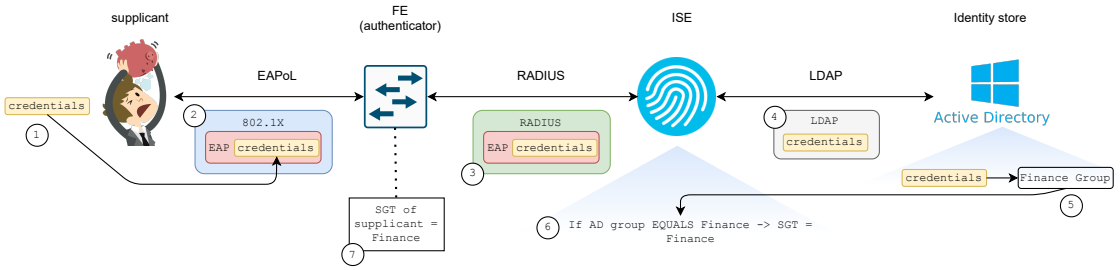

Figure 7.9: DNA Center Topology

First, the user named the supplicant in 802.1X logs in to his computer by entering his credentials (1). Then, the credentials are transported using the EAP protocol[33] over 802.1X to the access switch (2), this is also called EAP over LAN (EAPoL). In SDA, access switches are referred to as fabric edge nodes (see Figure 7.4). After that, the authenticator proxies the EAP data to ISE, which acts as a RADIUS server (3). Next, ISE extracts the credentials and performs an LDAP[41] query to the AD server to retrieve the group the user belongs to (4). Finally, the identity of the user is verified by AD (5). The group the user belongs to is returned to ISE. ISE contains authorization policies, which states that users belonging to the AD group Finance must be placed into the Finance SGT (6). Finally, ISE notifies the FE that the endpoint belongs to the Finance SGT (7).

7.4.1 Policy Enforcement

In SDA, policy enforcement is performed with the SGTs and the Secure Group ACLs defined in TrustSec.

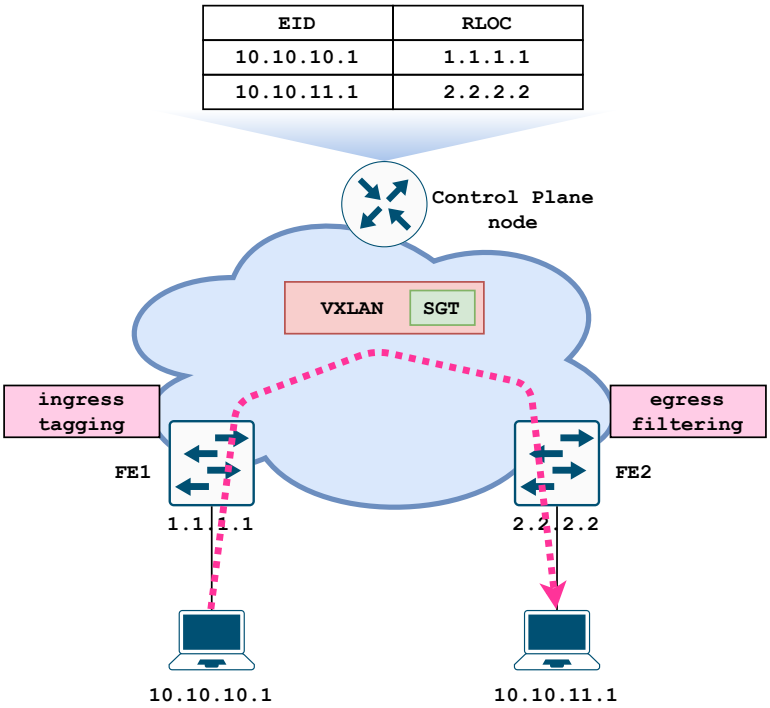

Figure 7.10: SDA Ingress Tagging and Egress Filtering

TrustSec uses ingress tagging and egress filtering to enforce the access policies[60]; the SGACLs are always applied at the egress node. Such a technique improves the scalability of SDA because the FEs do not need to know the SGTs of all endpoints, and download all the SGACLs in the system. Ingress tagging is performed at the ITR, which tags the source SGT in the VXLAN header of the packet. When the packet arrives at the ETR, the source SGT is retrieved and the filtering is performed using the SGACLs. This is illustrated in Figure 7.10.

Chapter 8

ACI-SDA Integration

This last chapter is about the integration of Software-Defined Access and Application-Centric Infrastructure, which are part of the Multidomain strategy pushed by Cisco these last few years[57]. As we will see, one of the assets of this multidomain integration is the ability to defined cross-domain policies to simplify the management of network policies, which can span across multiple domains.[62]. In particular, this integration gives the ability to filter the traffic between the two domains using group information exchanged by the controllers, Scalable-Group Tags in SDA, and Endpoint Groups in ACI.

It begins by presenting the integration of ACI and SDA in a theoretical way and then explains how the integration was achieved for the use case.

8.1 Theory

8.1.1 Phase 1 of the ACI-SDA integration

As of May 2021, there are two phases of the ACI-SDA integration. Due to hardware constraints in the lab, we had to limit ourselves to use the phase 1 of this integration, which bears some limitations as it only works between one VRF of an ACI tenant and one VN of SDA. Plain ethernet and regular IP routing are used for the communication between the domains.

8.1.2 Exchange of Identity/Group Information

The Integration of ACI and SDA enables the exchange of identity information between the two domains. As we have seen, ACI has some similarities with SDA when it comes to performing segmentation within the network: both use contracts and groups (EPGs in ACI, and SGTs in SDA). The integration allows administrators to apply policies within the ACI domain leveraging group information learned from SDA, and vice versa, enabling end-to-end segmentation from user to application to be performed seamlessly. For instance, they can define policies allowing a group of users in the campus network to access a particular set of applications running in the data center.

In the first phase of the integration, group information of only one ACI tenant (single VRF) can be sent to SDA, in particular, the APIC sends the EPGs and their IP mappings to ISE, And in reverse, ISE sends the SGTs and their IP mappings to the APIC. The ISE administrator can choose which SGTs to propagate to ACI. Communication between ISE and the APIC is handled by REST APIs, and the exchange is illustrated in Figure 8.1

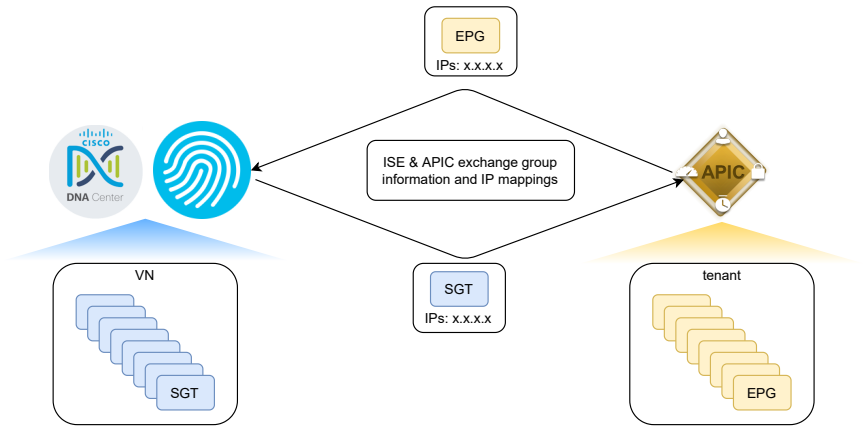

Figure 8.1: APIC and ISE exchange Group Information and IP Mappings

In ACI, the SGTs propagated are translated into external EPGs, and the EPGs are translated into SGTs in ISE. It is shown on Figure 8.2. In this example, an SGT named FINANCE is propagated to ACI, along with the IP information regarding the endpoints (10.10.100.102) belonging to this group. Note that on ISE, an IP to an SGT mapping is called an SXP mapping because SXP (Scalable-Group Tag eXchange Protocol)[31] is the protocol used by ISE to push the mappings to the network devices such as the fabric edge nodes.

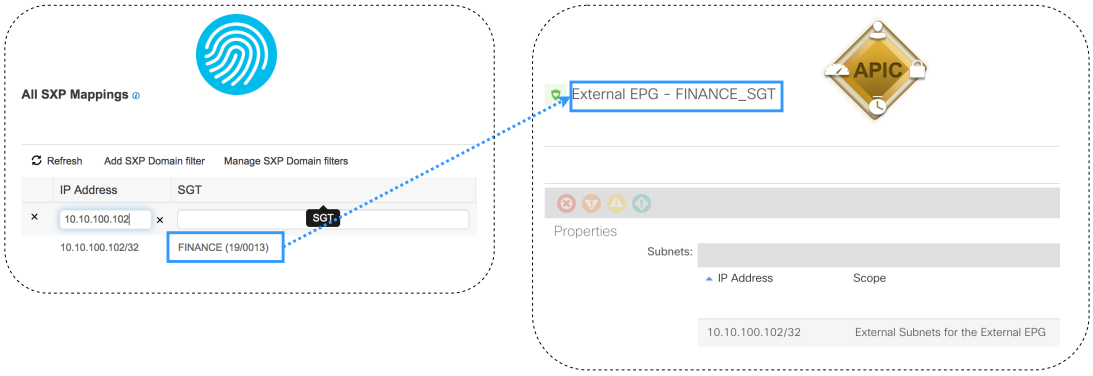

Figure 8.2: Finance SGT Translated Into an External EPG in ACI

Since normal IP routing and forwarding between the domains, ACI has to classify the traffic from SDA into the correct external EPG, this explains why IP addresses have to be sent along with the group information. The same process also applies to SDA.

8.1.3 Policy Enforcement

Given that the two domains can communicate with each other and that the SGTs (resp. EPGs) are propagated to ACI (resp. SDA), the administrators can use them to enforce policies with contracts. Contracts have to be created in both domains. To illustrate, imagine that a finance employee of the FINANCE SGT needs to access a web server running in ACI belonging to the APP EPG. To allow the communication between these two endpoints the administrator has to create two contracts. A contract in SDA allowing the FINANCE SGT to talk to the imported APP SGT, and a contract in ACI enabling the imported FINANCE External EPG to communicate with the APP EPG. This is shown below on Figure 8.3

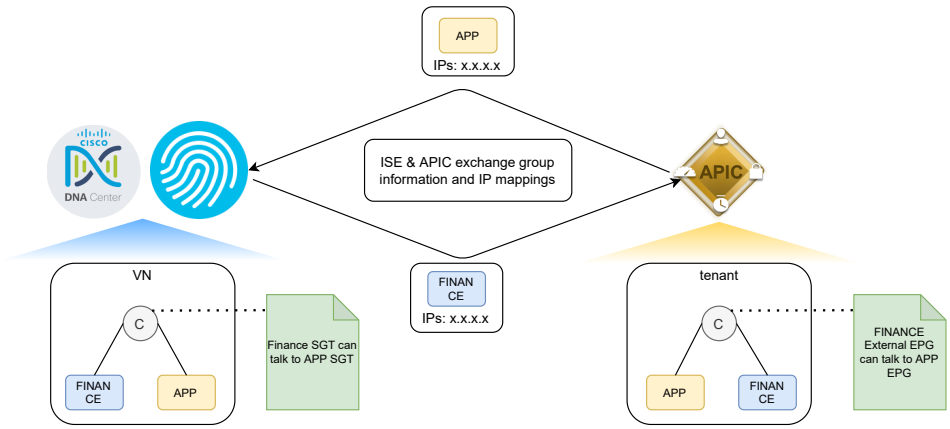

Figure 8.3: Multidomain Policy Enforcement

The main advantage of this integration is that it removes the need of maintaining large and hard-to-maintain access lists using IP addresses to filter the traffic between the two domains.

8.2 Use-Case Implementation

This section explains how the integration was carried out for the Cisco's Beer Factory use-case.

8.2.1 The ACI side

Once the integration is set up, what is left to do on the ACI side is to connect each application (inventory, finance, and webshop) to its corresponding external EPG via a contract. In that case, we have two external EPGs propagated by SDA (Finance and Retail) consuming an HTTP contract provided by the finance-frontend and inventory-frontend EPGs. Note that access policies are already enforced in the SDA domain, so there was no point in setting up two separate contracts. Customers are accessing the webshop application from the outside, so a default external EPG named RoW (Rest of the World) was created for this purpose. It consumes the contract http-lb-external, to which the service graph for the load balancing is attached (see Section 6.1.3). The contract relationships are shown on Figure 8.4

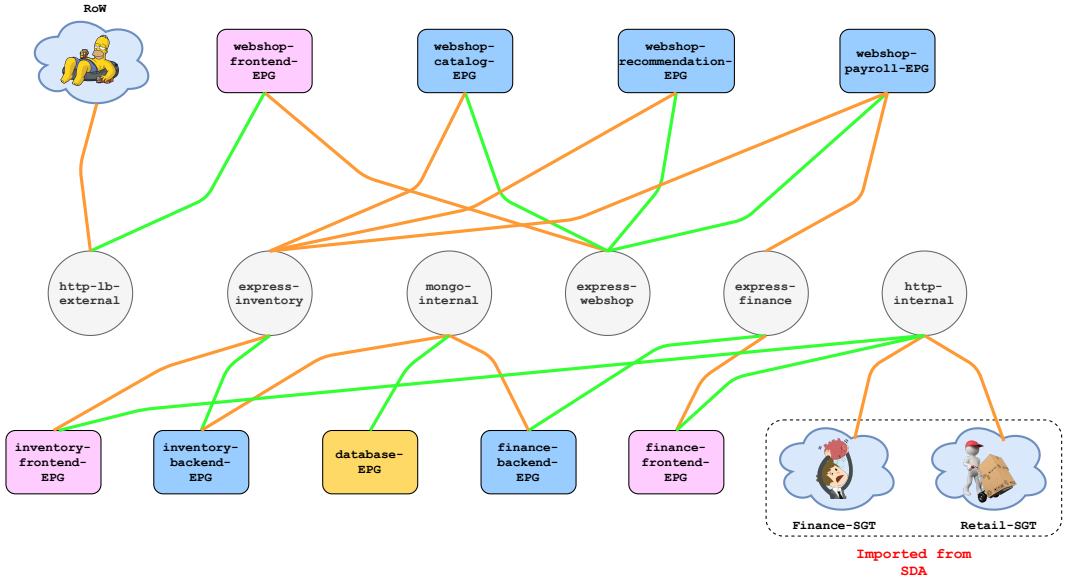

Figure 8.4: Contract Relationships for the Multidomain Integration

8.2.2 The SDA side

On the SDA side, there are two virtual machines, one for the finance employee, and the other one for the retail employee. Both of them authenticate to the network using 802.1X with an account managed in Active Directory as explained in Section 7.4. When logged in, they either belong to the Finance SGT or the Retail SGT. Then, just like in ACI, there is a contract allowing the Finance (resp. Retail) SGT to connect to the imported finance-frontend (resp. inventory-frontend) EPG over HTTP.

What was Already Setup

In the SDA lab, most of the setup was already configured as many demonstrations are given to customers with ISE and DNA Center. In particular, the virtual machines and the authentication of the users using 802.1X were already configured before the start of the internship. Consequently, the only lacking piece was to propagate the SGT of interest to ACI and setting up the contracts on DNA Center. In addition, the underlay for the connectivity between the ACI and SDA domains was also already set up.

Chapter 9

Conclusion

Software-Defined Networking has been a hot topic in the industry for years. Cisco has developed multiple SDN solutions, each of them targeting a particular domain of the IT infrastructure. Consequently, there is one solution for campus and enterprise networks named Software-Defined Access, and another one for the data center environment called Application-Centric Infrastructure. Those solutions are part of their Multidomain strategy aiming to integrate networking domains (the campus, data center, and WAN) with each other. As of now, ACI and SDA are quite complete products that are already running in the production environment of many customers, and the ability to create cross-domain policies between ACI and SDA is interesting as customers using Cisco technology on several domains can simplify their network management.

The objective pursued was to build a use case for this integration so that it can easily be explained and shown to Cisco customers. The use case takes part in the beer business and simulates the infrastructure of a small company named Cisco's Beer Factory running ACI and SDA. Two types of employees are sitting in the campus network, the finance and retail employees, and each of them has access to an application running in the data center, either a finance application or an inventory application. The company also runs a webshop application that can be accessed from the outside world by customers.

The DevOps and Infrastructure-as-Code movement is a second area of the IT industry that is gaining more and more ground. A second objective was to build an IaC demonstration for ACI to show its programmability features. This idea has been extended in the sense that all IT operations are performed using IaC.

So far, the point of the ACI-SDA integration is the ability to filter the traffic between the two domains using group information exchanged by the controllers, Scalable-Group Tags in SDA, and Endpoint Groups in ACI. This removes the need to maintain large access lists using IP addresses to filter the traffic. This feature, also called end-to-end segmentation was demonstrated in the use case by authorizing the finance (resp. retail) employees to only have access to the finance (resp. inventory) application.

There are multiple ways to pilot ACI. One can for instance deploy all the configurations using the graphical user interface of the controller, which seems to be a popular option as network engineers are not initially trained to write software. The other option is to rely on automation tools such as Terraform or Ansible, or by directly scripting API calls. Terraform was selected for this task. As a result, using Terraform and a GitOps approach to automate ACI was also achieved. The difficult part was to build a flexible code that remains manageable when the infrastructure grows.

In addition, the integration of other systems were also shown throughout this project. From the integration of ACI and, VMware Vsphere and Kubernetes, to the integration between Gitlab and Webex Teams. Then, performing all IT operations with IaC for these

systems was quite challenging as it involves learning a large number of tools (e.g. Terraform, Packer, Ansible, Docker, etc.), concepts, and the systems themselves.

The first phase of the integration between ACI and SDA is quite limited at the moment, however, it is a first step in this journey of integrating networking domains and will keep developing in the next years.

Regarding Infrastructure-as-Code, it makes no doubt that this practice of developing the infrastructure will keep evolving as it enables companies to bring their products to the market faster by making the infrastructure less of a bottleneck, and the majority of tech vendors are developing controllers or management platforms with open interfaces alongside the products that they are selling. Although IaC was heavily used during this internship, the task was greatly eased by having a greenfield environment. For large enterprises, adopting IaC and DevOps practices for managing the infrastructure is a journey whose destination is not reached overnight, as it is a complete culture change, involves training, and starting with a manually configured infrastructure already in place makes the task harder.

Bibliography

[1] The Kubernetes Authors. *Deployments*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/.

[2] The Kubernetes Authors. *Production-Grade Container Orchestration*. URL: https://kubernetes.io/.

[3] The Kubernetes Authors. *Service*. URL: https://kubernetes.io/docs/concepts/services-networking/service/.

[4] R. Enns; M. Bjorklund; J. Schoenwaelder; A. Bierman. *Network Configuration Protocol (NETCONF)*. 2011. URL: https://datatracker.ietf.org/doc/html/rfc6241.

[5] *Bootstrap*. URL: https://getbootstrap.com/.

[6] Yevgeniy Brikman. *Terraform Up and Running*. O'Reilly Media, Inc, 2016, p. 74.

[7] Michael Churchman. *Snowflake Configurations and DevOps Automation*. 2016. URL: https://www.sumologic.com/blog/snowflake-configurations-and-devops-automation/.

[8] inc. Cisco Systems. *Cisco Application Centric Infrastructure Fundamentals, Releases 2.x and 3.x*. 2014, p. 18.

[9] inc. Cisco Systems. *Cisco Application Centric Infrastructure Fundamentals, Releases 2.x and 3.x*. 2014, p. 30.

[10] inc. Cisco Systems. *Cisco Application Centric Infrastructure Fundamentals, Releases 2.x and 3.x*. 2014, p. 100.

[11] Jerome Durant. *Le SDN Pour Les Nuls*. 2015. URL: https://alln-extcloud-storage.cisco.com/gblogs/sites/53/2015/12/JRES2015-SDN-pour-les-nuls-11.pdf.

[12] IBM Cloud Education. *Infrastructure as Code*. 2019. URL: https://www.ibm.com/cloud/learn/infrastructure-as-code.

[13] V. Fuller; D. Farinacci. *Locator/ID Separation Protocol (LISP) Map-Server Interface*. 2013. URL: https://datatracker.ietf.org/doc/html/rfc6833.

[14] Cloud Native Computing Foundation. *CNI - the Container Network Interface*. URL: https://github.com/containernetworking/cni.

[15] Open Networking Foundation. *Software-Defined Networking (SDN) Definition*. URL: https://opennetworking.org/sdn-definition/.

[16] Andrew Froehlich. *Why Networks Are Evolving Toward Leaf-Spine Architectures*. 2018. URL: https://www.networkcomputing.com/networking/why-networks-are-evolving-toward-leaf-spine-architectures.

[17] Gitlab. *Terraform Gitlab Docker Image*. URL: https://gitlab.com/gitlab-org/terraform-images.

[18] Gitlab. *What is GitOps?* URL: https://about.gitlab.com/topics/gitops/.

[19] Adam Gluck. *Introducing Domain-Oriented Microservice Architecture*. 2020. URL: https://eng.uber.com/microservice-architecture/.

[20] Google. *Containers at Google*. URL: https://cloud.google.com/containers.

[21] Gursharan Singh; Pooja Gupta. *A review on migration techniques and challenges in live virtual machine migration*. 2016. URL: https://ieeexplore.ieee.org/document/7785015.

[22] *HAproxy*. URL: https://www.haproxy.com/.

[23] Hashicorp. *Providers*. URL: https://registry.terraform.io/browse/providers.

[24] Hashicorp. *Service mesh and microservices networking*. 2018. URL: https://www.hashicorp.com/resources/service-mesh-microservices-networking.

[25] Hashicorp. *State*. URL: https://www.terraform.io/docs/language/state/index.html.

[26] *Helm*. URL: https://helm.sh/.

[27] Docker Inc. *Docker overview*. URL: https://docs.docker.com/get-started/overview/.

[28] Red Hat Inc. *Containers vs VMs*. URL: https://www.redhat.com/en/topics/containers/containers-vs-vms.

[29] Docker inc. *Docker overview*. URL: https://docs.docker.com/get-started/overview/.

[30] Docker inc. *What is a Container?* URL: https://www.docker.com/resources/what-container.

[31] R. Kandula. *Scalable-Group Tag eXchange Protocol (SXP)*. 2019. URL: https://tools.ietf.org/id/draft-smith-kandula-sxp-07.html.

[32] Michael Kerrisk. *namespaces - overview of Linux namespaces*. URL: https://man7.org/linux/man-pages/man7/namespaces.7.html.

[33] B. Aboba; L. Blunk; J. Vollbrecht; J. Carlson; H. Levkowetz. *Extensible Authentication Protocol (EAP)*. 2004. URL: https://datatracker.ietf.org/doc/html/rfc3748.

[34] Ian Lewis. *The Almighty Pause Container*. 2017. URL: https://www.ianlewis.org/en/almighty-pause-container.

[35] D. Farinacci; V. Fuller; D. Meyer; D. Lewis; *The Locator/ID Separation Protocol (LISP)*. 2013. URL: https://datatracker.ietf.org/doc/html/rfc6830.

[36] Yunpeng Liu. *Only Slightly Bent: Uber's Kubernetes Migration Journey for Microservices*. 2019. URL: https://www.youtube.com/watch?v=91c3iUI2K7M.

[37] S. Josefsson; I. Liusvaara. *Edwards-Curve Digital Signature Algorithm (EdDSA)*. 2017. URL: https://tools.ietf.org/html/rfc8032.

[38] Microsoft. *Docker containers, images, and registries*. 2021. URL: https://docs.microsoft.com/fr-fr/dotnet/architecture/microservices/container-docker-introduction/docker-containers-images-registries.

[39] Microsoft. *What is a container?* URL: https://azure.microsoft.com/en-us/overview/what-is-a-container/.

[40] Mozilla and individual contributors. *SPA (Single-page application)*. URL: https://developer.mozilla.org/en-US/docs/Glossary/SPA.

[41] J. Sermersheim; Novell. *Lightweight Directory Access Protocol (LDAP): The Protocol*. 2006. URL: https://datatracker.ietf.org/doc/html/rfc4511.

[42] Wendell Odom. *CCNA 200-301 Volume 2*. Cisco Press, 2020, p. 364.

[43] Wendell Odom. *CCNA 200-301 Volume 2*. Cisco Press, 2020, p. 361.

[44] David Fiser; Alfredo Oliveira. *Why A Privileged Container in Docker Is a Bad Idea*. 2019. URL: https://www.trendmicro.com/en_us/research/19/l/why-running-a-privileged-container-in-docker-is-a-bad-idea.html.

[45] Matt Oswalt. *[SDN Protocols] Part 4 - OpFlex and Declarative Networking*. 2014. URL: https://oswalt.dev/2014/09/sdn-protocols-part-4-opflex-and-declarative-networking/.

[46] Jérôme Petazzoni. *Using Docker-in-Docker for your CI or testing environment? Think twice*. 2015. URL: https://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci/.

[47] Adam Raffe. *Learning ACI - Part 1: Overview*. 2014. URL: https://adamraffe.com/aci/nexus%5C%209000/2014/12/03/learning-aci-part-1-overview/.

[48] Inc. Red Hat. *Understanding DevOps*. URL: https://www.redhat.com/en/topics/devops.

[49] Inc. Red Hat. *What is Infrastructure as Code (IaC)?* URL: https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac.

[50] P. Congdon; B. Aboba; A. Smith; G. Zorn; J. Roese. *IEEE 802.1X Remote Authentication Dial In User Service (RADIUS) Usage Guidelines*. 2003. URL: https://datatracker.ietf.org/doc/html/rfc3580.

[51] Amazon Web Services. *What is DevOps?* URL: https://aws.amazon.com/devops/what-is-devops/.

[52] Cisco Systems. *ACI Fabric Endpoint Learning White Paper*. 2020. URL: https://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/application-centric-infrastructure/white-paper-c11-739989.html.

[53] Cisco Systems. *ACI Plugin for Red Hat OpenShift Container Architecture and Design Guide*. 2019, p. 9.

[54] Cisco Systems. *Application Centric Infrastructure Overview: Implement a Robust Transport Network for Dynamic Workloads*. 2013. URL: https://www.leadkeeper.net/files/downloads/ImplementARobustTransportNetworkForDynamicWorkloads.pdf.

[55] Cisco Systems. *Cisco Global Cloud Index 2015-2020*. 2015. URL: https://www.cisco.com/c/dam/m/en_us/service-provider/ciscoknowledgenetwork/files/622_11_15-16-Cisco_GCI_CKN_2015-2020_AMER_EMEAR_NOV2016.pdf.

[56] Cisco Systems. *Cisco Identity Services Engine*. URL: https://www.cisco.com/c/en_be/products/security/identity-services-engine/index.html.

[57] Cisco Systems. *Cisco Multidomain Integrations for Intent-Based Networking At-a-Glance*. 2019. URL: https://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/application-centric-infrastructure/at-a-glance-c45-741877.html.

[58] Cisco Systems. *Cisco Software-Defined Access Enabling intent-based networking 2nd edition*, p. 111.

[59] Cisco Systems. *Cisco TrustSec*. URL: https://www.cisco.com/c/en/us/solutions/enterprise-networks/trustsec/index.html.

[60] Cisco Systems. *Cisco TrustSec Switch Configuration Guide*. 2015. URL: https://www.cisco.com/c/en/us/td/docs/switches/lan/trustsec/configuration/guide/trustsec/arch_over.html.

[61] Cisco Systems. *Deploy a VXLAN Network with an MP-BGP EVPN Control Plane White Paper*. 2015. URL: https://www.cisco.com/c/en/us/products/collateral/switches/nexus-7000-series-switches/white-paper-c11-735015.html.

[62] Cisco Systems. *Intent-Based Networking's Next Evolution: Policy Integrations Between Multiple Domains*. 2019. URL: https://www.cisco.com/c/en/us/solutions/collateral/enterprise-networks/nb-06-multidomain-wp-cte-en.html.

[63] Cisco Systems. *OpFlex: An Open Policy Protocol White Paper*. 2014. URL: https://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/application-centric-infrastructure/white-paper-c11-731302.html.

[64] Samar Raza Talpu. "Network Traffic Observations in Data Centers andForecasting Techniques for Resource Utilization". In: *Future Technologies Conference (FTC) 2017* (2017), p. 990. DOI: https://saiconference.com/Downloads/FTC2017/Proceedings/137_Paper_21-Network_Traffic_Observations_in_Data_Centers.pdf.

[65] Gideon Tam. *Happy Birthday, Cisco Application Centric Infrastructure*. 2014. URL: https://blogs.cisco.com/perspectives/happy-birthday-cisco-application-centric-infrastructure.

[66] Cyxtera Technologies. *Why Slow Provisioning IT Infrastructure Derails Digital Transformation*. 2018. URL: https://www.cyxtera.com/blog/data-centers/why-slow-provisioning-it-infrastructure-derails-digital-transformation.

[67] Huawei Technologies. *Huawei DCN Design Guide*. 2018. URL: https://support.huawei.com/enterprise/en/doc/EDOC1100023542?section=j00z&topicName=spine-leaf-network-architecture.

[68] VMware. *VLAN configuration on virtual switches, physical switches, and virtual machines*. 2020. URL: https://kb.vmware.com/s/article/1003806.

[69] VMware. *vSphere Distributed Switch*. URL: https://www.vmware.com/be/products/vsphere/distributed-switch.html.

[70] Shawn Wargo. *Cisco SD-Access - Solution Fundamentals - DGTL-BRKCRS-2810*. 2019. URL: https://www.ciscolive.com/global/on-demand-library.html?search=sda%5C%20fundamentals#/session/1573153543008001J8XI.

[71] M. Bjorklund; A. Bierman; K. Watsen. *RESTCONF Protocol*. 2017. URL: https://datatracker.ietf.org/doc/html/rfc8040.

[72] M. Smith; R. Adams; M. Divorkin; Y. Laribi; V. Pandey; P. Garg; N. Weidenbacher. *OpFlex Control Protocol draft-smith-opflex-03*. 2016. URL: https://tools.ietf.org/html/draft-smith-opflex-03.

[73] M. Mahalingam; D. Dutt; K. Duda; P. Agarwal; L. Kreeger; T. Sridhar; M. Bursell; C. Wright. *Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks*. 2014. URL: https://tools.ietf.org/html/rfc7348.

[74] Takuya Yishida. *Mastering ACI forwarding*. 2020. URL: https://www.ciscolive.com/c/dam/r/ciscolive/emea/docs/2020/pdf/BRKACI-3545.pdf.

Chapter 10

Appendix

10.1 Code Deposited

An invite to the group 'wcollin-TFE' has been sent on the Gitlab of Montefiore to the members of the jury. It contains three subgroups:

- microservices. It contains the code of the applications (inventory-frontend, inventory-backend, finance-frontend, etc.). Each repository has three branches of interest: 'master' for the deployment on K8s using Terraform, 'helm' for the deployment on k8s using Helm charts, and 'vm' for version of the code deployed on virtual machines.
- IaC. Contains the code related to the automation of ACI (ACI-live & ACI-modules) using Terraform, the deployment of the applications to K8s with Terraform (K8s-live & K8s-modules), the code for the migration of the applications to virtual machines (microservices-on-vms & vSphere-modules), the load-balancer config (HAproxy), and a swiss knife IaC docker image.
- webex. Related to the creation of Webex Rooms (webex-rooms), the tickets and pipeline notifications system deployed on K8s with Helm charts (webex-tickets & webex-pipelines)