Datacenter Environment and Automation Tasks

MARCO MARTINS CORREIA

Licenciando em Engenharia Informática, Redes e Telecomunicações pelo Instituto Superior de Engenharia de Lisboa

Projeto Final de Licenciatura

Orientador: Nuno Miguel Figueiredo Garcia, Prof. Assistente Convidado (ISEL)

Arguente: Professor Doutor Nuno Miguel Machado Cruz (ISEL)

julho de 2025

Datacenter Environment and Automation Tasks

MARCO MARTINS CORREIA

Licenciando em Engenharia Informática, Redes e Telecomunicações pelo Instituto Superior de Engenharia de Lisboa

Projeto Final de Licenciatura

Orientador: Nuno Miguel Figueiredo Garcia, Prof. Assistente Convidado (ISEL)

Arguente: Professor Doutor Nuno Miguel Machado Cruz (ISEL)

julho de 2025

Datacenter Environment and automation tasks

Abstract

This final course project focuses on automating network configuration and management tasks in datacenters, with the aim of eliminating human errors, ensuring consistency and increasing operational efficiency.

The project is based on the integration of two main technologies: NetBox, used as a central inventory source (source of truth), and Ansible, responsible for automating the configuration of network devices. The infrastructure was designed based on the leaf-spine architecture, supporting underlay and overlay networks. EVPN-VXLAN technology was used in the overlay layer, allowing logical segmentation of the network and the creation of multitenant environments.

The topology was implemented and tested in Containerlab, simulating a virtual environment with Nokia SR Linux devices. The inventory was created and managed in NetBox, ensuring a clear and centralized view of the network, while Ansible, through Jinja2 templates, automated the configuration of interfaces, IP addresses, BGP sessions and EVPN-VXLAN settings.

Using NetBox as the source of truth and Ansible as the automation engine resulted in a solution that reduces implementation time and facilitates datacenter scalability, ensuring reliability and resilience. The project shows how a structured approach to automation can simplify the management of complex networks and adapt to the increasing demands of these types of environments.

Acronyms

AFI - Address Family Identifier

API - Application Programming Interface

ARP - Address Resolution Protocol

AS - Autonomous System

ASN - Autonomous System Number

BD - Broadcast Domain

BGP - Border Gateway Protocol

BUM - Broadcast, Unknown Unicast, and Multicast

CLI - Command-Line Interface

CSV - Comma-Separated Values

DCIM - Data Center Infrastructure Management

ECMP - Equal-Cost Multi-Path

eBGP - External Border Gateway Protocol

EVI - Ethernet VPN Instance

EVPN - Ethernet Virtual Private Network

FCS - Frame Check Sequence

FIB - Forwarding Information Base

IaC - Infrastructure as a Code

iBGP - Internal Border Gateway Protocol

ICMPv6 - Internet Control Message Protocol version 6

IMET - Inclusive Multicast Ethernet Tag

IP - Internet Protocol

IPAM - IP Address Management

IP-VRF - IP Virtual Routing and Forwarding

IRB - Integrated Routing and Bridging

JSON - JavaScript Object Notation

JSON-RPC - JSON Remote Procedure Call

L2VPN - Layer 2 Virtual Private Network

L3VPN - Layer 3 Virtual Private Network

MAC - Media Access Control

MAC-VRF - MAC Virtual Routing and Forwarding

MP-BGP - Multi-Protocol Border Gateway Protocol

NLRI - Network Layer Reachability Information

OSPF - Open Shortest Path First

RA - Router Advertisement

RD - Route Distinguisher

REST - Representational State Transfer

RIB - Routing Information Base

RR - Route Reflector

RT - Route Target

SAFI - Subsequent Address Family Identifier

SDN - Software-Defined Networking

SoT - Source of Truth

SR - Service Router

STP - Spanning Tree Protocol

TCP - Transmission Control Protocol

ToR - Top of Rack

UDP - User Datagram Protocol

VLAN - Virtual Local Area Network

VNI - VXLAN Network Identifier

VTEP - VXLAN Tunnel End Point

VXLAN - Virtual Extensible Local Area Network

YAML - YAML Ain't Markup Language

Index

List of Figures

CHAPTER ONE

1 Introduction

1.1 Motivation

The management of modern datacenters faces unprecedented challenges. The growing complexity of infrastructures, driven by virtualization, containerization, and the need for massive scalability, has rendered traditional manual configuration approaches unsustainable. Manual configuration performed device-by-device is intrinsically slow, susceptible to human error, and results in inconsistencies known as “configuration drift”. These problems not only increase costs and mean time to resolution but also limit the agility required for a business to respond quickly to new demands.

In this context, network automation emerges as a fundamental necessity, not merely an optimization. The adoption of Infrastructure as a Code (IaC) practice, where network infrastructure is managed programmatically, is essential to ensure the consistency, repeatability, and reliability that modern environments demand. The core motivation for this project is to demonstrate, in a practical and replicable manner, the implementation of such an automation framework, moving away from error-prone manual processes towards a more resilient, efficient, and agile operational model.

1.2 The Problem and the Proposed Solution

The main technical challenge this project addresses is the complexity involved in deploying and managing a modern datacenter fabric. A network built on a leaf-spine architecture using technologies such as EVPN-VXLAN requires detailed and interdependent configuration of the underlay routing, the overlay control plane, and tenant-specific services. Manually setting up this multi-layered environment often relies on complex spreadsheets to track IP addresses and configurations, a method that is highly susceptible to human error and severely constrains the network's scalability.

The proposed solution, illustrated in Figure 1, is an automated, data-driven framework that systematically solves this problem, by adhering to the network automation lifecycle paradigm of **Day 0, Day 1, and Day 2** operations.

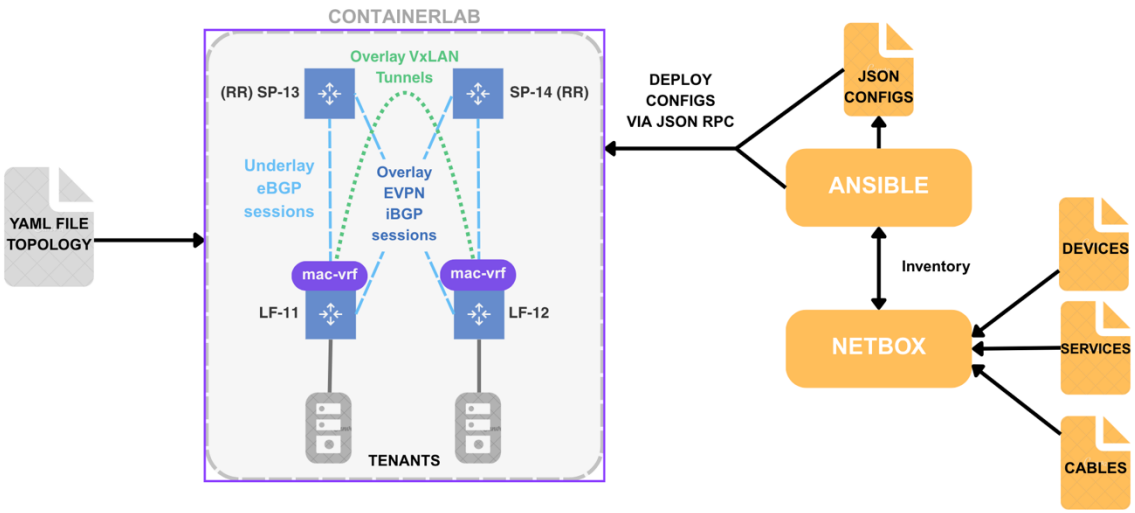


Figure 1 - Proposed Solution

- **Day 0: Design and Modeling in the Source of Truth.** This initial phase focuses on defining the network's intended state. Instead of using static spreadsheets, NetBox is used as a centralized and structured Source of Truth (SoT). All data including the physical inventory, device roles, IP addressing, BGP ASNs, and the full definition of tenant services is modeled in NetBox. This creates a complete, programmatically accessible blueprint of the entire network design before any device is configured.

- **Day 1: Initial Automated Deployment.** This phase involves translating the blueprint from Day 0 into a running configuration on devices. Ansible was chosen as the automation engine to orchestrate this process. It programmatically queries the NetBox API to build a dynamic inventory at runtime. Using this data, it generates device-specific configurations via Jinja2 templates and deploys them to the network devices via JSON-RPC. This ensures the initial state of the network precisely matches the intended state defined in the SoT.

- **Day 2: Ongoing Operations.** This phase involves all subsequent actions in the network's lifecycle, such as provisioning new services, scaling the fabric by adding new devices, or modifying existing configurations. The framework is designed so that all Day 2 operations are initiated by returning to the Day 0 phase: an operator simply updates the data model in NetBox to reflect the new desired state. The same Day 1 automation is then re-run. Due to its idempotent nature,

Ansible will apply only the necessary changes to bring the network into alignment with the updated SoT.

1.3 Report structure

This report has five chapters. The contributions of each chapter are summarized as follows:

Chapter One: Provides the motivation, the problem and the proposed solution.

Chapter Two: Provides the necessary theoretical background for the technologies employed in this project.

Chapter Three: Details the practical execution of the project.

Chapter Four: Presents the validation of the implemented Framework.

Chapter Five: Summarizes the key findings, achievements, and possible improvements of the project.

CHAPTER TWO

2 State of the art

In recent years, the increasing complexity of data centers and the demand for greater operational agility have driven the adoption of automation solutions. Traditional approaches based on manual configurations have become inadequate to keep pace with the scale and speed required by modern environments, especially with the rise of cloud computing, software-defined networking (SDN), and infrastructure as code (IaC).

Tools such as Ansible, NetBox, Terraform, and SaltStack have emerged as key players in the current landscape, offering effective mechanisms for automating configuration, management, and monitoring of network infrastructures. Among these, Ansible stands out due to its simplicity, modular structure, and compatibility with a wide range of platforms and devices. NetBox, on the other hand, has gained popularity as a central source of truth for network inventory, providing seamless integration with automation tools like Ansible via RESTful APIs.

The leaf-spine architecture, combined with technologies like EVPN-VXLAN, has become a standard design in modern data centers. It enables horizontal scalability, traffic segmentation, and support for multitenant environments.

In this context, the integration between NetBox and Ansible, explored in this project, reflects a current trend in network engineering. It promotes greater consistency, reduces human error, and supports full automation of the network infrastructure lifecycle.

2.1 Datacenter Overview and Architecture

According to Dinesh G. Dutt in *Cloud Native Data Center Networking* (2020), Modern data center designs often utilize the Clos topology, named after Charles Clos, one of its original designers, due to its ability to scale efficiently.

This design is frequently visualized as a two-tiered system made up of spine and leaf switches, which is why it's commonly referred to as a leaf-spine topology. Each leaf switch connects to every spine switch, establishing a mesh that enables high network throughput. Leaf switches serve as the connection point for servers, while spine switches interconnect the leaves. Typically, servers and leaf switches are housed within the same physical rack, with the switch positioned at the top, earning the nickname "Top of Rack" (ToR) switch.

One of the main benefits of this topology is its ability to support multiple simultaneous paths between any two servers, which significantly boosts bandwidth. Additional spine switches can be introduced to further enhance inter-leaf capacity.

Unlike traditional aggregation layers in legacy architectures, spine switches in a Clos topology serve a singular role: linking leaf switches. They do not host compute resources or perform services beyond basic forwarding. This architecture emphasizes decentralization by pushing network intelligence and services to the edges, namely, the leaf layer and the servers, rather than centralizing it at the core.

The leaf-spine architecture provides a scalable and resilient network fabric, as illustrated in Figure 2. In this design, all leaf switches are connected to all spine switches, ensuring high bandwidth and fault tolerance.

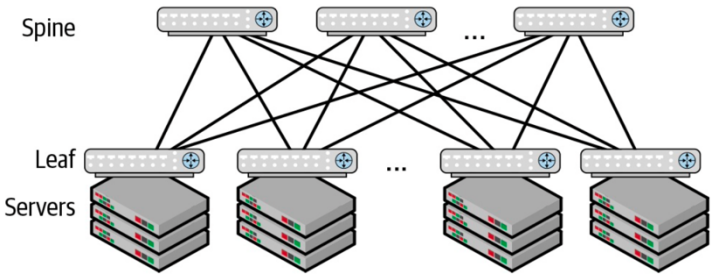


Figure 2 - Illustration of a leaf-spine Clos topology

Note. From Cloud Native Data Center Networking: Architecture, Protocols, and Tools (p.38), by D. G. Dutt, 2020, O'Reilly Media

Benefits for Network Automation and Datacenters

The repetitive and predictable nature of the leaf-spine topology makes it an ideal candidate for network automation. Because all leaves have a similar function and all spines have a similar function, configurations can be easily templated and deployed using tools like Ansible.

By using a routed fabric with ECMP all links between the leaf and spine layers are active. If a spine switch fails or a link goes down, traffic is automatically and seamlessly redirected over the remaining paths.

2.2 Underlay Networks

The underlay network is the physical infrastructure network that provides IP connectivity between the datacenter servers and routers. Routers in the underlay typically use eBGP to exchange routes to the datacenter workloads and routers due to its simplicity, scalability, and ease of multi-vender interoperability as per RFC 7938.

Benefits for Network Automation

The simplicity and repetitive nature of a Layer 3 underlay make it ideal for automation. With standardized configurations across all leaf and spine switches, operators can use tools like Ansible to deploy, manage, and validate the entire fabric from a central point. Innovations like BGP unnumbered further simplify configuration by removing the need to manage thousands of point-to-point IP addresses on inter-switch links, drastically reducing the potential for human error.

2.2.1 BGP Unnumbered

In traditional BGP-based fabrics, each point-to-point link between switches requires a dedicated /30 or /31 IP subnet. Managing this large number of IP addresses is endless, error-prone, and adds significant complexity to the source of truth (NetBox) and the automation templates (Ansible). BGP unnumbered solves this problem by allowing BGP sessions to be established without configuring IPv4 or IPv6 addresses on the interconnecting links.

The next Figure 3 exemplifies how a BGP unnumbered session is established.

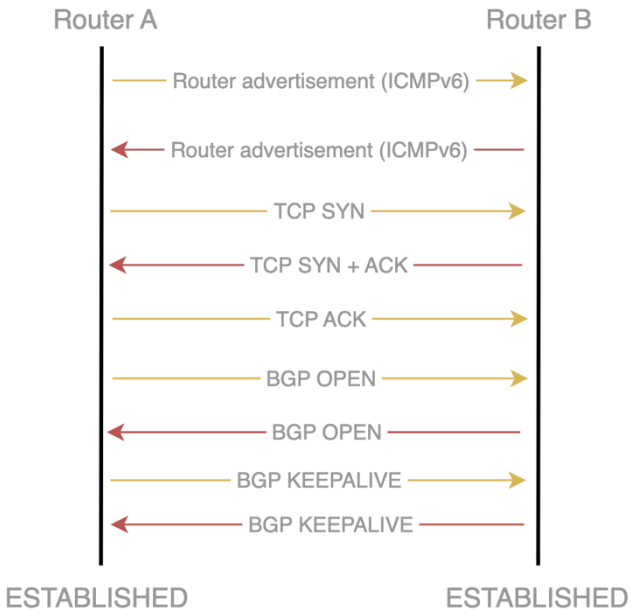


Figure 3 - Establishing a BGP unnumbered session

The sessions are established using IPv6 link-local addresses, which are automatically generated on every IPv6-enabled interface. Instead of static neighbor configuration, switches dynamically discover their peers on a link by exchanging ICMPv6 Router Advertisement (RA) messages. Once a peer's link-local address is discovered, a BGP session is automatically established over these addresses. The peering is authenticated and authorized based on the Autonomous System Number (ASN) of the neighbor, which must be within a pre-configured range.

The adoption of BGP unnumbered has a very positive impact on the project's automation workflow:

- **Simplified Source of Truth (NetBox):** The data model in NetBox becomes significantly cleaner. There is no longer a need to create, assign, and track dozens of IP prefixes for the fabric's point-to-point links. The "source of truth" is simplified to only essential data, reducing the potential for data entry errors.
- **Simplified Automation (Ansible):** Ansible automation becomes more robust and less complex. The Jinja2 templates for configuring the underlay do not need to contain logic to render link-specific IP addresses. The configuration is reduced to simply enabling BGP on the relevant interfaces, which drastically reduces the potential for configuration errors and makes the entire fabric deployment faster and more scalable.

2.3 Overlay Networks

An overlay network, illustrated in Figure 4, is established using tunneling techniques to carry traffic over the underlay network. This makes an overlay network logically separate and independent from the addressing and protocols used in the underlay network. It also keeps the overlay networks logically separate from each other. Workloads that are connected to the same overlay network can send Ethernet or IP packets to each other but not to workloads in other overlay networks. iBGP is used to distribute reachability information for the workload endpoints.

Benefits for Datacenters

This approach enables traffic isolation, custom network designs, and secure connectivity between departments, making overlay networks especially valuable in complex datacenter environments.

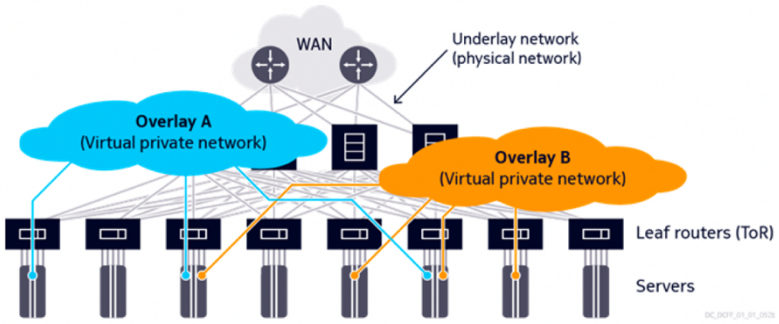


Figure 4 - Overlay Network

Note. From Nokia (2023). "Nokia Datacenter Fabric Fundamentals 3.1".

2.3.1 VXLAN

VXLAN is the overlay tunnel technology most often used in datacenters. VXLAN enables the interconnection of Layer 2 segments over a datacenter IP fabric (IP-based underlay). A VXLAN tunnel is established between two endpoints, known as VXLAN tunnel endpoints, or VTEPs. The VTEP is typically a leaf-router or a datacenter gateway.

The local VTEP encapsulates packets received from a local endpoint and sends the encapsulated packets over the IP underlay network. The remote VTEP decapsulates the tunneled packets and forwards them to the remote endpoint. Remote endpoint addresses can be learned through flooding in VXLAN, like L2 MAC learning in switches or through the exchange of EVPN routes using Multi-Protocol BGP as per RFC 7348.

The following Figure 5 is illustrating the format of a VXLAN packet:

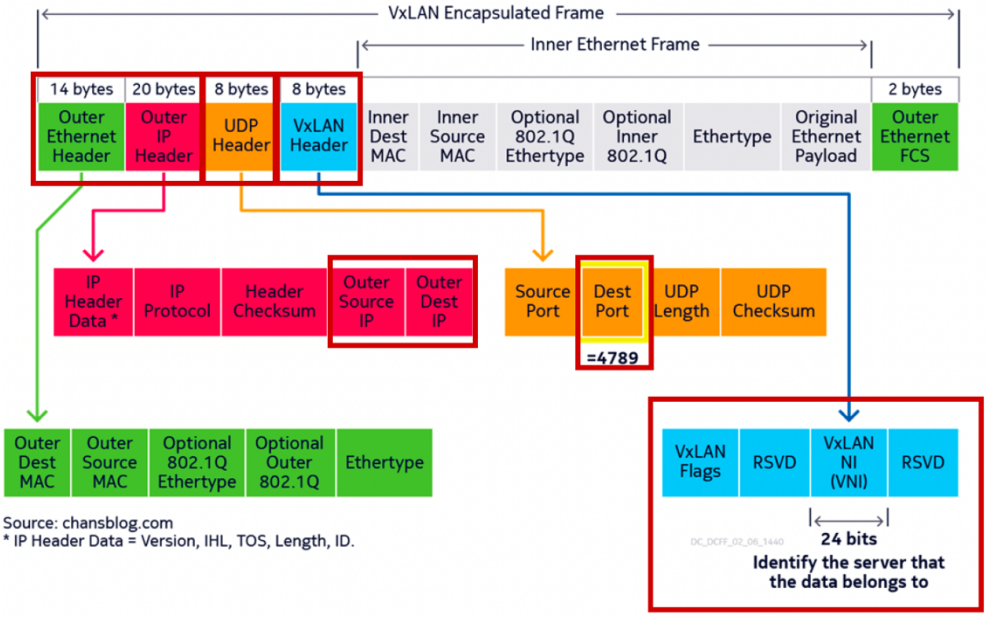


Figure 5 – VxLAN packet format

Note. From Nokia (2023). "Nokia Datacenter Fabric Fundamentals 3.1".

Overall Frame Structure

The entire frame starts with an Outer Ethernet Header (14 bytes), followed by an Outer IP Header (20 bytes), a UDP Header (8 bytes), and a VXLAN Header (8 bytes). These outer headers encapsulate the Inner Ethernet Frame, which is the original data frame. The frame ends with an Outer Ethernet FCS (Frame Check Sequence, 2 bytes).

Header Details

- The inner Ethernet frame is the data originating from the host.
- A VXLAN header (blue) is first added. It includes the 24-bit VNI (which allows for up to 16 million unique network segments (compared to the 4094 limit of VLANs).
- A UDP header (orange) in then added. The UDP destination port is set to 4789 to indicate a VXLAN-encapsulated data.
- The UDP header is encapsulated by an outer IP header (red). The outer source UP identifies the ingress VTEP, the VTEP that performs VXLAN encapsulation. The outer destination IP identifies the remote egress VTEP, to which the encapsulated data is destined.
- The Outer Ethernet header (green) is changed at each IP hop within the datacenter fabric.

2.3.2 EVPN (Ethernet Virtual Private Network)

EVPN is an address family defined for MP-BGP. EVPN can be used as the control plane with VXLAN data plane encapsulation to support bridged, routed and hybrid overlay services. MP-BGP sessions are established between VTEPs with MAC/IP information exchanged between the VTEPs as BGP-EVPN routes. There are several different EVPN route types which will be described later.

In most datacenter network designs, spines are often used as BGP route reflectors since they have direct connections to every device in the IP fabric and their control plane workload is less than the routers with VTEPs.

The EVPN standards define different types of EVPN routes to support the various EVPN use cases:

- **EVPN Type 1**, or Ethernet Auto-Discovery, routes are used in multi-homing scenarios to support aliasing and MAC mass withdraw for fast convergence.
- **EVPN Type 2**, or MAC/IP Advertisement, routes are used to advertise reachability information. Each route advertises the MAC address of a host, or optionally the MAC and IP addresses of the host.
- **EVPN Type 3**, or Inclusive Multicast Ethernet Tag (IMET), routes are used to discover remote VTEPs participating in the same VPN service, and to setup the tree required to deliver broadcast, unknown and multicast (BUM) traffic across EVPN networks.
- **EVPN Type 4**, or Ethernet Segment (ES), routes are used in multi-homing scenarios to discover VTEPs attached to the same Ethernet segment. They are also used to support the election of the designated forwarder.
- **EVPN Type 5**, or IP-Prefix, routes are used to advertise IP prefixes to support inter-subnet connectivity in L3VPN services.

By using these specialized BGP routes, EVPN eliminates the need for Spanning Tree Protocol (STP) and dramatically reduces broadcast traffic within the fabric. This approach optimizes the handling of BUM traffic, ensuring that only relevant endpoints receive this type of traffic.

A broadcast domain (BD) refers to an instantiation of an EVPN Layer 2 service on a given VTEP. A BD can span across multiple VTEPs connected to the same IP fabric, allowing communication between hosts attached to VTEPs in the same BD as if they were connected to the same Layer 2 switch, as illustrated in the Figure 6.

An EVPN instance (EVI) uniquely identifies an EVPN service that can span multiple VTEPs. The EVI refers to the group of broadcast domains that are part of the same EVPN service, providing the framework for multi-tenant isolation and workload mobility.

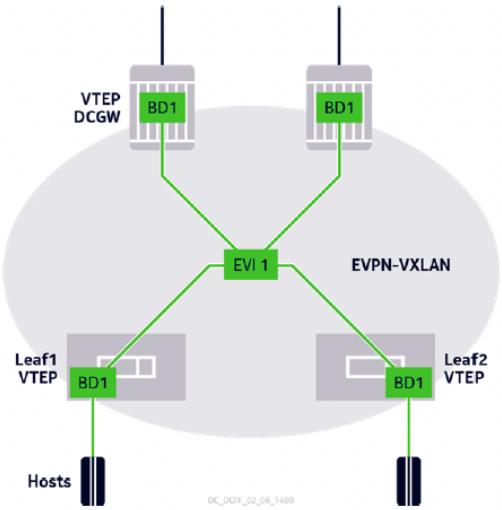


Figure 6 - EVPN Instance (EVI) Architecture

Note. From Nokia (2023). "Nokia Datacenter Fabric Fundamentals 3.1".

In EVPN, to isolate multi-tenant traffic, Route Distinguishers (RDs) and Route Targets (RTs) are used:

- **Route Distinguisher (RD):** Is an identifier attached to each route to ensure that overlapping IP or MAC addresses across different tenants are treated as distinct routes in the BGP control plane.
- **Route Target (RT):** Is an extended BGP community attribute that is used to control how routes are imported and exported between different VRFs (Virtual Routing and Forwarding, discussed in **section 2.4 and 2.5**). This mechanism ensures that only routes with the correspondent RTs are distributed across specific VRFs, ensuring traffic isolation between tenants. Note:
    - Each EVPN Instance (EVI) requires a unique route distinguisher (RD) per MAC-VRF and one or more globally unique route targets (RTs).

2.3.2.1 Advertised Route Types in Layer 2 Services

In the case of single-homed hosts, VTEPs participating in an EVPN-VXLAN service advertise two types of routes (Figure 7):

- **Type 2 MAC/IP Advertisement** routes are used to advertise the MAC or MAC/IP addresses of local hosts. VTEPs use type 2 routes received from remote VTEPs to populate their bridge tables in the MAC-VRF instances. The MAC-VRF is a Layer 2 network instance. An EVPN-enabled MAC-VRF use VXLAN tunnel to

connect to other MAC-VRFs of the same broadcast domain. A local VTEP uses its bridge table to send known unicast traffic to the proper remote VTEP.

- **Type 3 Inclusive Multicast Ethernet Tag (IMET)** routes are used to discover VTEPs participating in the same network instance. VTEPs use type 3 routes received from remote VTEPs to populate their flooding lists. A VTEP uses its flooding list to send BUM traffic to remote VTEPs.

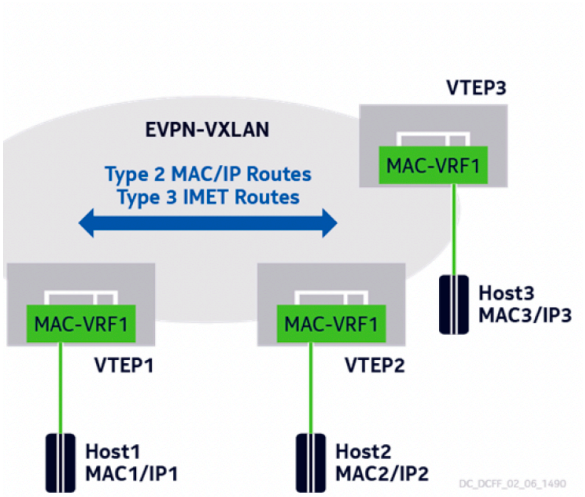


Figure 7 - Advertised Route Types for Layer 2 Services

Note. From Nokia (2025). "Nokia Datacenter Fabric Fundamentals 4.0".

2.3.2.2 MAC Learning in Layer 2 Services

VTEPs maintain and manage a MAC-VRF table for each provisioned EVPN service. Host MAC addresses learned on local bridge sub-interfaces are stored in the MAC-VRF of the associated EVPN service.

In Figure 8, Host 1 sends an ARP request message to learn the MAC address associated with IP2. The ARP request message has MAC1 as a source MAC address. VTEP1 receives the message over its bridged sub-interface, `eth-1/11.1`, learns the source MAC address MAC1, and stores the locally learned address in the MAC-VRF instance.

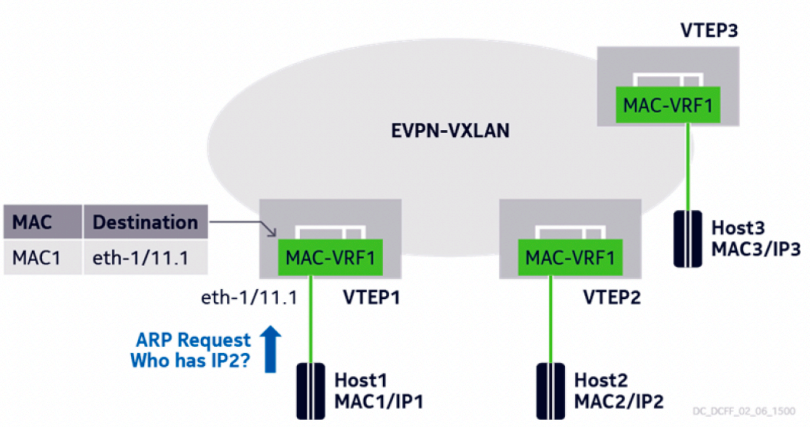


Figure 8 - MAC Learning is Layer-2 Services

Note. From Nokia (2025). "Nokia Datacenter Fabric Fundamentals 4.0".

2.3.2.3 MAC Advertisement in Layer 2 Services

Each EVPN instance requires a route distinguisher (RD) that is unique per MAC-VRF and one or more globally unique route targets:

- Route Distinguisher (RD): is used by the VTEP to distinguish routes between EVIs. It can be manually configured or automatically derived as `system-ip:evi`.
- Route Target (RT): is used by receiving VTEP to determine which routes are to be installed in the local network instance. Only routes with matching RT are installed. Like RD, RT can be manually configured or automatically derived as `autonomous-system:evi`.

When a VTEP learns a local MAC address it advertises the MAC/IP address pair to remote VTEPs using a MAC/IP Advertisement route (Type 2 MAC/IP Advertisement) over MP-BGP. The MP-BGP update includes the MAC/IP address pair, a route distinguisher, and a label, as well as other route parameters.

- The route distinguisher distinguishes routes between different EVPN instances.
- The label is used by remote VTEPs to encapsulate frames in the data plane. In the case of VXLAN encapsulation, the label refers to a VNI.

As shown in the Figure 9, VTEP1 learns MAC1 and advertises the address in a type 2 EVPN route to remote VTEPs. VTEP 1 also includes the route distinguisher and VNI label provisioned for the EVPN instance. In addition to the EVPN route, the MP-BGP update includes route targets. VTEPs use the RTs to determine which local MAC-VRF should import the received VRF route.

VTEP2 and VTEP3 receive the MP-BGP update, check the RT (RT1), and install MAC1 in the MAC-VRF OF EVPN instance. VTEP2 AND VTEP3 can now forward any unicast frame destined to MAC1 by encapsulating the frame with the associated VNI label, VN1, and sending it over the tunnel towards the next-hop VTEP1.

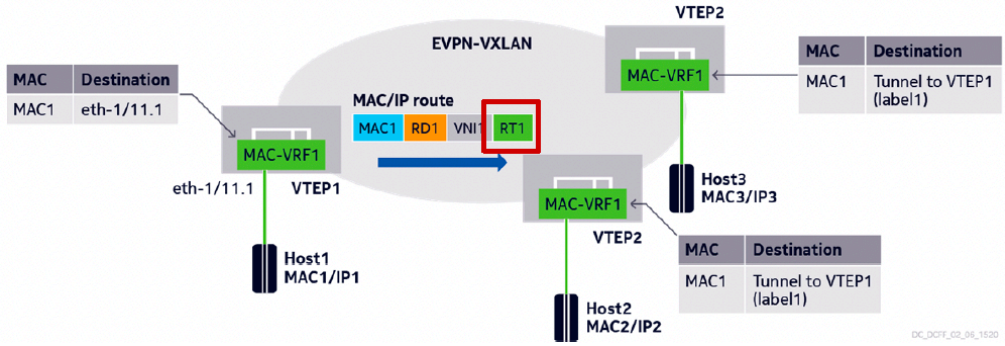


Figure 9 - MAC Advertisement is Layer-2 Services

Note. From Nokia (2025). "Nokia Datacenter Fabric Fundamentals 4.0".

2.4 Layer 2 Services (MAC-VRF)

In modern datacenter networks that leverage EVPN-VXLAN, providing isolated Layer 2 services for different tenants or applications is a primary requirement. This is accomplished using a MAC-VRF (MAC Virtual Routing and Forwarding).

A MAC-VRF creates its own independent bridging table, defining a separate Layer 2 broadcast domain within a physical switch. This mechanism is fundamental to enabling multi-tenancy in an EVPN fabric. Each MAC-VRF acts as a distinct logical network, ensuring that MAC addresses and broadcast traffic from one tenant remain fully isolated from those of others, even though they all share the same physical infrastructure.

By using MAC-VRFs, data center operators can create and manage thousands of isolated, scalable, and flexible Layer 2 networks on a shared physical underlay. This model is fundamental to the automation goals of this project, as the creation and management of these services can be defined as data in a Source of Truth and deployed programmatically.

2.5 Layer 3 Services (IP-VRF)

An IP-VRF is essentially a virtual router instance running on a physical network device. Its primary function is to create multiple, independent routing domains on a single piece of hardware.

Each IP-VRF maintains its own separate and isolated set of resources, including:

- **Routing Information Base (RIB):** The IP-VRF has its own routing table, ensuring that routes learned within one VRF are not visible or accessible to any other VRF.
- **Forwarding Information Base (FIB):** The corresponding forwarding table is also separate, dictating how packets belonging to that VRF are forwarded.
- **A dedicated set of interfaces:** Logical or physical interfaces are assigned to a specific VRF, binding them to that virtual routing domain.

The main benefit of this approach is complete traffic isolation at Layer 3. This allows for the creation of logically separate networks for different customers, departments, or applications on a shared physical infrastructure.

CHAPTER THREE

3 Implementation Overview

The project implementation was based on automating the data center infrastructure using a scalable approach. Network configurations were automated using Ansible and Jinja2 templates to ensure consistency and reduce manual errors. NetBox was used as the source of truth for managing the network inventory. JSON-RPC enabled the configuration of Nokia SR Linux devices, allowing the exploration of data center technologies through EVPN-VXLAN for the creation of overlay services and a multitenant environment. The underlay network leveraged BGP unnumbered, simplifying IP address management and enabling dynamic peering between directly connected devices without requiring manually assigned IP addresses.

3.1 Project Planning and Timeline

The implementation of this project followed a structured timeline over a five-month period, from February to July 2025. The work was organized into distinct, overlapping phases to ensure a logical progression from initial research to final validation. The Gantt chart, presented in Figure 10, provides a visual representation of this execution plan.

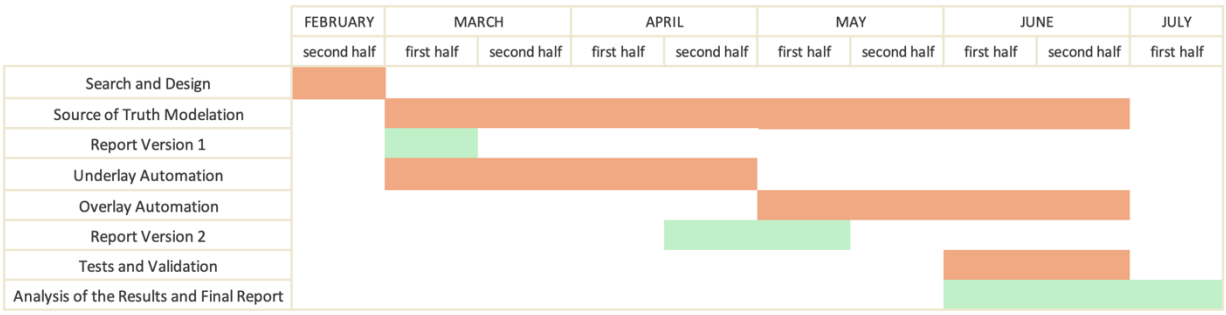


Figure 10 - Project Execution Gantt Chart

- **Search and Design:** The project began with a research phase focused on the state-of-the-art technologies for datacenter automation. This concluded with the final selection of the architectural approach (Leaf-Spine with EVPN-VXLAN) and the core tools (NetBox and Ansible).

- **Source of Truth Modeling:** This phase was dedicated to implementing the "Day 0" design. It involved modeling the entire network fabric, devices, interfaces, cabling, and the logical service constructs, within NetBox, establishing it as the single Source of Truth.

- **Underlay Automation:** Focused on creating the Ansible playbooks and Jinja2 templates required to deploy the BGP Unnumbered transport network.

- **Overlay Automation:** Built upon the functional underlay, this stream involved developing the more complex logic to dynamically provision L2VPN services based on the data model.

- **Testing, Validation, and Documentation:** This final phase was iterative and ran in parallel with the later stages of development.
    - The Tests and Validation stage involved executing the use cases detailed in Chapter 4 to verify the functional correctness of the entire framework.
    - The project concluded with the Analysis of the Results and Final Report where the benefits of the framework were analyzed, and all findings were compiled into this document.

3.2 Day 0: Design and Modeling in the Source of Truth

The objective of Day 0 is not to deploy a live network, but to build a complete, precise, and programmatically accessible model of the desired state. This data-driven model serves as the single source of truth for all future automation tasks. In this project, NetBox acts as the SoT, and the Day 0 phase involves finalizing the network design, thoroughly modelling it in NetBox, and preparing the automation environment to consume this data.

3.2.1 Finalizing the Network Design

- **Topology:** The foundation of this project's network infrastructure is a leaf-spine topology, a modern architectural standard for datacenters, as discussed in **Section 2.1**. The specific topology implemented for this project consists of a two-tier fabric with two spine switches and two leaf switches, as illustrated in Figure 11 (Spine switches `SP-13` and `SP-14`, Leaf switches `LF-11` and `LF-12`). This entire topology was deployed virtually using Containerlab, with each node emulating a Nokia SR Linux device. This virtualized physical layout serves as the underlay network, which will be configured in the subsequent steps to support the routing and overlay services.

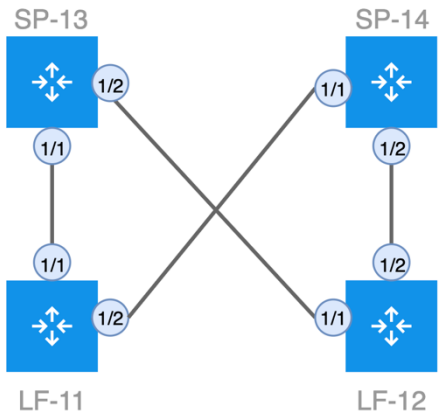


Figure 11 - Proposed network topology

- **Underlay Protocol:** For the underlay network, BGP unnumbered was chosen as the routing protocol to establish connectivity between the leaf and spine switches. This choice was made to simplify the Day 0 model, as it eliminates the need to plan, document, and manage thousands of point-to-point IP addresses for inter-switch links. Connectivity is established automatically using IPv6 link-local addresses.
- **Overlay Protocol:** EVPN-VXLAN was selected for the overlay to provide scalable and multi-tenant Layer 2 and Layer 3 services. The control plane for EVPN is iBGP, with all devices operating within a single Autonomous System (AS 65535). To avoid a full mesh of BGP sessions between leaf switches, the spine switches are configured as Route Reflectors (RRs), and the leaf switches act as their clients.

3.2.2 Modeling the Datacenter in NetBox: Establishing the Source of Truth

3.2.2.1 The Source of Truth Concept and the Role of NetBox

In traditional network management, configuration data and operational state are often fragmented across disparate sources such as static spreadsheets and institutional knowledge held by network engineers. This decentralized approach is naturally vulnerable to inconsistencies, and human mistakes, which makes it an unreliable basis for modern automation projects. As networks scale in complexity, maintaining an accurate and up-to-date representation of the infrastructure becomes nearly impossible, leading to “configuration drift” and prolonged troubleshooting cycles.

The concept of a Source of Truth (SoT) was introduced to solve these problems. A Source of Truth is a single, trusted location that contains all the data needed to define the desired state of the network. A SoT does not necessarily reflect the operational or running state of the network at any given moment but rather the desired, formally declared state that all automation aims to achieve.

For this project, NetBox was selected to serve as the SoT, providing a structured database and a robust API, replacing complex spreadsheets, becoming the programmatic foundation for the entire automation lifecycle. By using NetBox as the SoT, this project directly addresses the challenges described above and gains several important benefits:

- **Documentation**: The NetBox data model serves as the single source of truth for the network’s intended design, always up to date. Any change to the desired state is made in NetBox first, ensuring the documentation stays accurate and never becomes outdated.

- **Data Integrity**: NetBox applies a strict data model to make sure IP addresses are unique, interface connections are valid, and all required details (like device roles or BGP ASNs) are defined before being used in automation, helping the prevention of data entry mistakes right at the source.

- **Consistency**: Ensures that all automation tools, in this case, Ansible, to pull data from one trusted source. This removes the risk of “configuration drift” caused by manual changes or conflicting data repositories.

3.2.2.2 The NetBox Data Model for this Project

3.2.2.2.1 Modeling the Underlay Fabric

. This model was kept intentionally simple:

- **“Physical” Infrastructure (DCIM):** DCIM objects were used to represent the “physical” world: Sites (ISEL), Device Roles (leaf, spine), Manufacturers (Nokia), Device Types (7220 IXR-D2L), and the Devices themselves. Physical Interfaces and Cables were modeled to define the leaf-spine topology.

- **IP Addressing (IPAM):** Only the essential IP Addresses for the `system0` loopback interfaces were created and assigned. No IP addresses were defined for the point-to-point links.

- **BGP Peering Data:** To facilitate BGP configuration, a custom field, `DEVICE_ASN`, was added to the Device model to store each router's Autonomous System Number. Another custom field, `PEER_AS`, was created on the Interface model to identify which links are part of the BGP fabric and what their neighbor's ASN is expected to be.

3.2.2.2.2 Modeling the Overlay Fabric

The goal was to create a flexible and scalable model to define multi-tenant Layer 2 services (EVPN-VXLAN) that could be dynamically deployed by Ansible. This required establishing a clear chain of relationships between different NetBox objects (Figure 12):

- **L2VPNs:** This object serves as the representation of a tenant service. Each L2VPN object defines the service's unique EVI (Ethernet VPN Instance) via the Identifier field, and its BGP control plane policies through the association with Route Targets.

- **VNI Association to L2VPN:** A critical piece of information for a VXLAN service is its VXLAN Network Identifier (VNI). As NetBox lacks a native object for VNIs,

a workaround was implemented. A custom field named `vni_vlan` was added to the L2VPN model. This field links the L2VPN to a specific VLAN object. This approach provides a structured way to associate a unique VNI with each L2VPN service.

- **Access VLANs:** VLAN objects are created to represent the customer-facing VLAN tag for each service (e.g., VLAN 1 for SERVICE1). These VLANs are then assigned to the Tagged VLANs list on the appropriate physical interface. This natively models a trunk port carrying multiple tagged VLANs.

- **VLAN to L2VPN Link:** The most crucial link in the data model is connecting an access VLAN to the specific L2VPN service it carries. To achieve this, a custom field named `bound_l2vpn` was added to the VLAN model. This field creates an explicit, one-to-one relationship, allowing an administrator to declare, for instance, that "VLAN 1" is the access VLAN for "SERVICE1".

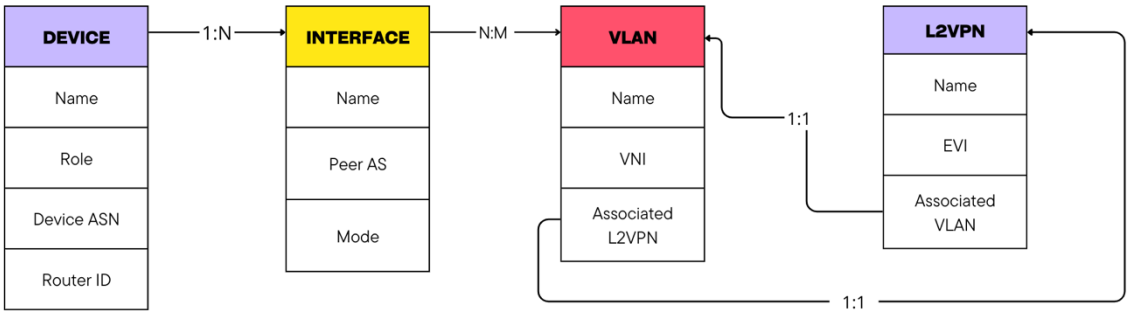


Figure 12 - Relations between objects and fields

3.2.2.3 Populating NetBox

This process involved cataloging all network devices, including routers and switches, and mapping their physical and logical interconnections to ensure an accurate topological representation.
Devices and connection data were populated into NetBox efficiently through its bulk import functionality, utilizing CSV-formatted files.

The initial device population was performed by importing a CSV file (Figure 13). This file defined key attributes for each device, including its name, operational status, location, manufacturer, functional role (e.g., leaf, spine), and model type. The successful importation resulted in the device list shown in Figure 14.

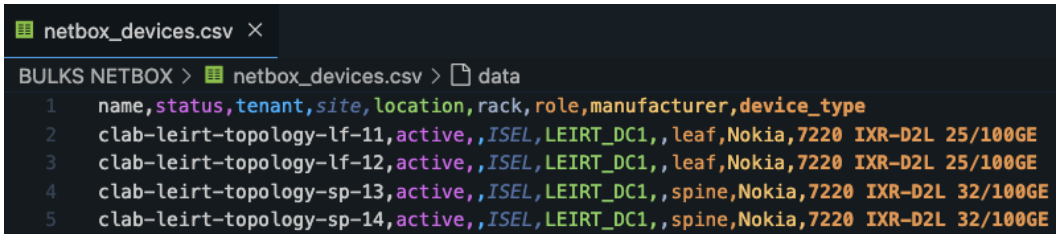


Figure 13 - CSV Bulk File with Network Devices

NOME	STATUS	SITE	LOCAL	FUNÇÃO	FABRICANTE	TIPO
clab-leirt-topology-lf-11	Ativo	ISEL	LEIRT_DC1	leaf	Nokia	7220 IXR-D2L 25*100GE
clab-leirt-topology-lf-12	Ativo	ISEL	LEIRT_DC1	leaf	Nokia	7220 IXR-D2L 25*100GE
clab-leirt-topology-sp-13	Ativo	ISEL	LEIRT_DC1	spine	Nokia	7220 IXR-D3L 32*100GE
clab-leirt-topology-sp-14	Ativo	ISEL	LEIRT_DC1	spine	Nokia	7220 IXR-D3L 32*100GE

Figure 14 - NetBox Devices added with the Bulk File

Following the device import, physical connectivity was established by importing a second CSV file (Figure 15). This file defined the inter-device cabling, specifying the termination points for each connection:

- **side_a_device / side_b_device:** The devices at each end of the cable.
- **side_a_type / side_b_type:** The object type, specified as a network interface (dcim.interface).
- **side_a_name / side_b_name:** The specific interface name on each device.

Figure 15 – CSV Bulk file with cables

The result of this import was a complete and accurate representation of the network's physical connections within NetBox, as illustrated in Figure 16.

ID	ETIQUETAS	STATUS	TERMINAÇÃO A	DISPOSITIVO A	TERMINAÇÃO B	DISPOSITIVO B
7	LEIRT	Conectado	ethernet-1/1	clab-leirt-topology-lf-11	ethernet-1/1	clab-leirt-topology-sp-13
10	LEIRT	Conectado	ethernet-1/1	clab-leirt-topology-lf-12	ethernet-1/2	clab-leirt-topology-sp-13
13	LEIRT	Conectado	ethernet-1/2	clab-leirt-topology-lf-11	ethernet-1/1	clab-leirt-topology-sp-14
14	LEIRT	Conectado	ethernet-1/2	clab-leirt-topology-lf-12	ethernet-1/2	clab-leirt-topology-sp-14

Figure 16 – NetBox cables added with the bulk file

3.2.3 API token generation for automation

Automated inventory extraction required programmatic integration between Ansible and NetBox. To facilitate this, an API access token was generated within the NetBox web interface. The generation process, showed in Figure 17, involved creating a new token with appropriate permissions and an expiration date.

This token provided the necessary credentials for Ansible to authenticate its API requests and securely retrieve inventory data, forming the foundation for dynamic inventory management.

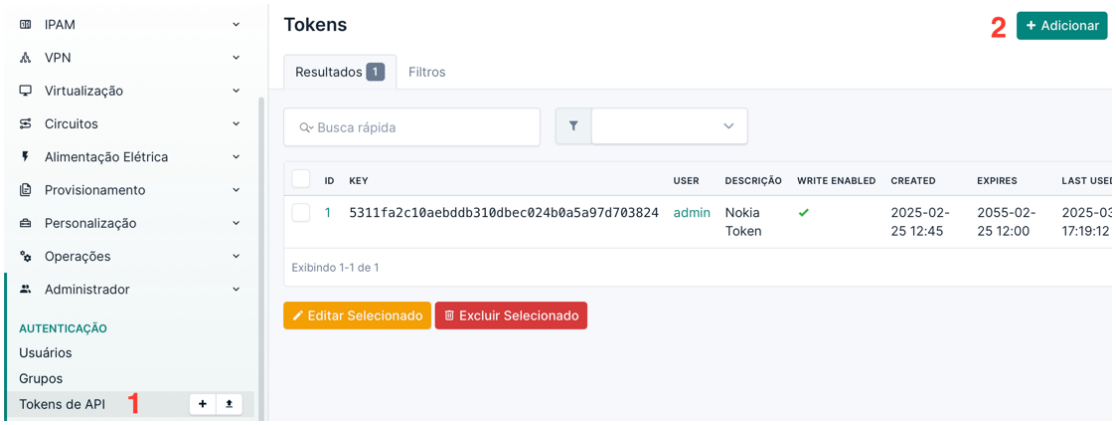


Figure 17 – Steps to generate a NetBox API token

3.2.4 Ansible dynamic inventory configuration

Ansible was chosen for this project precisely because of its ability to leverage a dynamic inventory plugin to interact directly with NetBox.

The configuration was defined in the **netbox_inventory.yml** file as shown below. This file specified key parameters:

- The NetBox API endpoint and the authentication token.
- Query filters to select devices tagged with LEIRT.
- Grouping logic to organize the resulting hosts by their manufacturer and device_roles.

```
plugin: netbox.netbox.nb_inventory
api_endpoint: http://127.0.0.1:8001
token: 630abadce8cdd1b35325814c2b18744fc9774d62
validate_certs: false
config_context: false
query_filters:
  - tag: LEIRT
group_by:
  - manufacturers
  - device_roles
```

The successful configuration of the dynamic inventory was verified by executing the `ansible-inventory --list -i netbox_inventory.yml` command. This confirmed that Ansible could dynamically source its inventory from NetBox, ensuring that all subsequent automation playbooks would operate on data aligned with the designated source of truth (Figure 18).

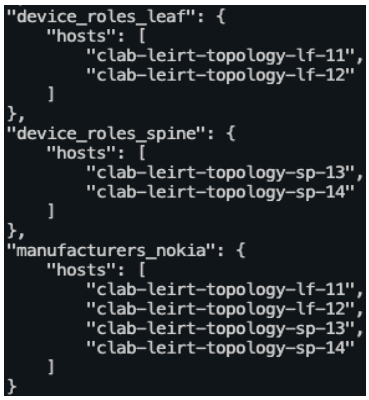


Figure 18 - Inventory obtained dynamically

3.3 Day 1: Initial Automated Deployment

3.3.1 Automation workflow

The Day 1 deployment process is driven by a carefully orchestrated workflow that translates the structured data from the Source of Truth (NetBox) into running configurations on the network devices. This workflow, orchestrated by Ansible, is designed to be repeatable, consistent, and idempotent. It ensures that every time the automation is run against a set of unconfigured devices, the result is a fully functional network fabric that precisely matches the intended state defined in Day 0. The entire process can be visualized as a data flow, as illustrated in Figure 19. This workflow forms the core of the Day 1 automated deployment.

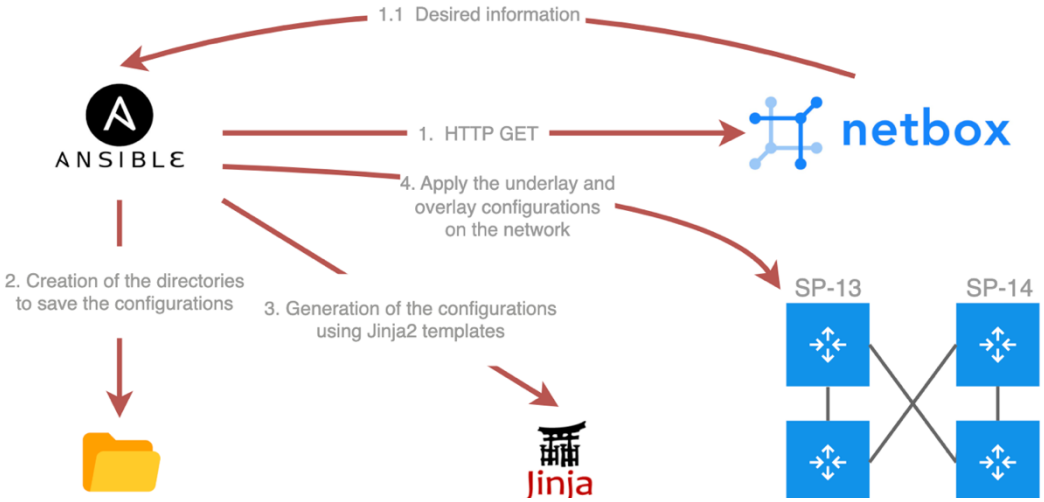


Figure 19 - Day 1 automation workflow

The workflow consists of four primary steps:

1. **Data Retrieval (HTTP GET):** The process begins when an Ansible playbook is executed. The first action performed by the `netbox.netbox.nb_inventory` plugin is to send an authenticated API request (HTTP GET) to the NetBox server. This request queries for all devices tagged with LEIRT, retrieving a comprehensive set of data for each device, including its name, role, site and interface connection details.
2. **Dynamic Inventory and Directory Creation:** Ansible receives the data from NetBox and dynamically builds its in-memory inventory of hosts to manage. Concurrently, a preparatory task in the playbook creates local directories on the control node. These directories are used to store the device configurations that will be generated in the next step.
3. **Configuration Generation (Jinja2 Templating):** Ansible processes a set of predefined Jinja2 templates for each device in the inventory. It combines the generic configuration logic within the templates with the specific device data retrieved from NetBox.
4. **Configuration Deployment:** In the final step, Ansible uses a network-specific connection module (in this case, one compatible with Nokia SR Linux's JSON-RPC API) to connect to each device and push the generated configuration to the device, only applying the changes necessary to bring the device to the desired state. On the very first run (Day 1), this means applying the entire configuration. On subsequent runs, if no data in NetBox has changed, Ansible will detect that the device is already in the correct state and will make no changes.

3.3.2 Templating and Deploying the Underlay Fabric

The first and most critical phase of the Day 1 deployment is establishing the underlay network. The underlay serves as the IP transport fabric over which all overlay services will operate. The deployment is managed by a dedicated Ansible role (`underlay_conf`). The role playbook is designed to be run against all devices in the fabric and executes a series of tasks that translate the underlay design from NetBox into running device configurations.

Phase 1 - Data Aggregation

Before any configuration can be generated, the playbook first gathers all necessary data points from the NetBox API. This involves a series of URI calls to specific NetBox endpoints to retrieve:

- The device's own loopback IP address (`system0`).
- The device's assigned ASN.
- A list of all interfaces on the device that will participate in the BGP fabric.
- The IP prefix for the loopback address pool, which is used to build a routing policy.

Phase 2 - Configuration Generation via ebgp.j2 Template

This single template contains all the necessary logic to generate a complete underlay configuration for any device, whether it's a leaf or a spine, generating:

- **System Loopback Interface (`system0`):** The playbook first configures the `system0` interface as shown in the code below. This is the most important interface on the device, as its IP address serves as the BGP Router ID and the next-hop for all overlay routes. The template retrieves the specific IP address assigned to `system0` in NetBox and renders it in the configuration.

```
{
  "interface": [
    {% for ip in ip_addresses.json.results if ip.assigned_object.name == "system0" %}
    {
      "name": "system0",
      "description": "system0 interface on {{ inventory_hostname }} node",
      "admin-state": "enable",
      "subinterface": [
        {
          "index": 0,
          "admin-state": "enable",
          "ipv4": {
            "admin-state": "enable",
            "address": [
              {
                "ip-prefix": "{{ ip.address }}"
              }
            ]
          }
        }
      ]
    },
    {% endfor %}
```

- **BGP Unnumbered Interfaces:** This is where the simplification of BGP unnumbered is realized as shown in the code below. The template iterates through all physical interfaces that are part of the fabric (identified by the presence of a `PEER_AS` custom field in NetBox). For each interface, it enables IPv6 and configures it to send ICMPv6 Router Advertisements. This is all that is needed for the devices to automatically discover each other's link-local

addresses and establish a BGP session. Noticeably absent is any IPv4 or IPv6 addressing, which dramatically simplifies the template and the data model in NetBox.

```
{% for interface in interfaces.json.results if interface.custom_fields.PEER_AS != none %}
    {
      "name": "{{ interface.display }}",
      "description": "{{ interface.display }} interface on {{ inventory_hostname }} node",
      "admin-state": "enable",
      "subinterface": [
        {
          "index": 0,
          "admin-state": "enable",
          "ipv6": {
            "admin-state": "enable",
            "router-advertisement": {
              "router-role": {
                "admin-state": "enable"
              }
            }
          }
        }
      ]
    }{% if not loop.last %},{% endif %}
    {% endfor %}
```

- **BGP Process and Routing Policy:** Finally, the template configures the BGP process itself, using the device specific ASN and `Router_ID` from NetBox. After that the dynamic neighbor's section is configured, which instructs BGP to automatically form peering's on the unnumbered interfaces with neighbors in the allowed peer AS range.

To ensure only the essential loopback routes are advertised across the fabric, a routing policy (`system-loopbacks-policy`) is created. This policy uses a prefix-set that matches the aggregate prefix of all loopbacks and is applied as both an import and export policy to the BGP group. This prevents unnecessary routes from being exchanged and keeps the underlay routing table clean and efficient.

After the ebgp`-device-name`.json file is rendered and saved locally, the final task in the playbook uses the `nokia.srlinux.config` module to apply the configuration.

3.3.3 Templating and Deploying the Overlay Fabric on Leafs

Phase 1 - Data Aggregation

Like the underlay deployment, the playbook of the role for the overlay (only for leafs: `overlay_conf_leaf`) begins by gathering all necessary data from the NetBox API. However, for the overlay, the queries are more complex, as they need to retrieve information about logical services and their relationships. This involves a series of API calls to retrieve:

- A list of all L2VPN services tagged for the fabric.
- The VRFs associated with these services.
- A list of all VLANs, which are used to model both access-side client separation and internal VNI-to-service mappings.
- A detailed list of all interfaces for each target device, including their mode (tagged) and which VLANs are assigned to them.
- The iBGP overlay Autonomous System number (AS 65535 in this project).

Phase 2 - Configuration Generation via evpn_leaf.j2 Template

The `evpn_leaf.j2` template is designed to be generic, capable of generating a complete overlay configuration for any leaf switch, regardless of how many services it supports, generating:

- **iBGP Overlay Peering:** The template first establishes the iBGP control plane for EVPN, configuring a BGP group named `ibgp-overlay` and enables the `evpn` address family. After that it iterates through the list of spine switches (identified by their device role in NetBox) and creates BGP neighbor sessions to each one, using their loopback addresses, establishing establishes the peering necessary to exchange EVPN routes, as shown in the code snippet below.

```
{
  "network-instance": [
    {
      "name": "default",
      "protocols": {
        "bgp": {
          "group": [
            {
              "group-name": "ibgp-overlay",
              "peer-as": "{{ ibgp_asn }}",
```

```
            "afi-safi": [
              { "afi-safi-name": "ipv4-unicast", "admin-state": "disable" },
              { "afi-safi-name": "evpn", "admin-state": "enable" }
            ],
            "timers": { "minimum-advertisement-interval": "1" },
            "local-as": { "as-number": "{{ ibgp_asn }}" }
          }
        ],
        "neighbor": [
          {%- for device in all_devices.json.results if device.role.display == "spine" -%}
          {
            "peer-address": "{{ device.custom_fields.Router_ID }}",
            "peer-group": "ibgp-overlay",
            "transport": { "local-address": "{{ devices.json.results[0].custom_fields.Router_ID
}}" }
          }
          {%- if not loop.last %},{% endif -%}
          {%- endfor %}
        ]
      }
    }
  }
```

- **Tenant Service Instantiation (MAC-VRF):** Next step, the template iterates through the list of VLANs that are tagged on the target device's interfaces. For each VLAN, it uses the `bound_l2vpn` custom field to find the corresponding L2VPN service, generating a `mac-vrf` network instance for that service, linking the tagged access sub-interface (e.g., `ethernet-1/49.1`) to the VXLAN sub-interface (e.g., `vxlan1.1`). This logic, shown in the code snippet below, ensures that a `mac-vrf` is only created on a leaf if a client for that service is connected to it.

```
    {% for vlan in vlans_with_service %}
    {%  set  l2vpn  =  (l2vpns.json.results  |  selectattr('id',  'equalto',
vlan.custom_fields.bound_l2vpn.id) | first) %}
    {% if l2vpn %}
      "name": "{{ found_vrf.name }}",
      "type": "mac-vrf",
      "admin-state": "enable",
      "interface": [ { "name": "{{ access_port_name.name }}.{{ vlan.vid }}" } ],
      "vxlan-interface": [ { "name": "vxlan1.{{ l2vpn.custom_fields.vni_vlan.vid }}" } ],
      "protocols": {
        {...}
      }
    }
  {% endif %}
  {% endfor %}
```

- **Access Interface Configuration:** After that, the template identifies all physical interfaces on the device that are configured in tagged mode and have VLANs assigned. For each of these "trunk" ports, it generates the necessary sub-interfaces, one for each tagged VLAN. The following code illustrates how the template creates a bridged sub-interface with desired encapsulation for each service.

```
{%- for iface in access_ifaces_for_loop -%}
{
  "name": "{{ iface.name }}",
  "vlan-tagging": true,
  "subinterface": [
    {%- for vlan in iface.tagged_vlans -%}
    {
      "index": {{ vlan.vid }},
      "type": "bridged",
      "admin-state": "enable",
      "vlan": {
        "encap": {
          "single-tagged": {
            "vlan-id": {{ vlan.vid }}
          }
        }
      }
    }
    {%- if not loop.last %},{% endif -%}
    {%- endfor %}
  ]
}
```

- **VXLAN Tunnel Interface Configuration:** Finally, the template configures the tunnel-interface `vxlan1`, creating a bridged sub-interface for each unique VNI that corresponds to a service active on the leaf switch. This maps the incoming traffic from the overlay network to the correct Layer 2 domain, as shown below.

```
"tunnel-interface": [
  {
    "name": "vxlan1",
    "vxlan-interface": [
      {%- for l2vpn in active_l2vpns_for_loop -%}
      {
        "index": {{ l2vpn.custom_fields.vni_vlan.vid }},
        "type": "bridged",
        "ingress": { "vni": {{ l2vpn.custom_fields.vni_vlan.vid }} }
      }
      {%- if not loop.last %},{% endif -%}
      {%- endfor %}]}]
```

3.3.4 Templating and Deploying the Overlay Fabric on Spines

While leaf switches are responsible for instantiating tenant services, spine switches act as Route Reflectors (RRs). This design choice eliminates the need for a full mesh of iBGP sessions between all leaf switches, which would be very hard to manage at scale. Instead, each leaf only needs to peer with the spine RRs, which then reflect the learned EVPN routes to all other leaf switches.

The configuration for the spines is generated with another template, `evpn_spine.j2`, which focuses exclusively on this RR functionality:

Phase 1 - Data Aggregation

The data required for spine configuration is minimal compared to the leafs. The Ansible playbook (from the role `overlay_conf_spine`) only needs to retrieve:

- A list of all devices in the fabric to identify which ones are leafs.
- The Router ID of the spine switch being configured.
- The iBGP overlay Autonomous System number.

Phase 2 - Configuration Generation via evpn_spine.j2 Template

- **Route Reflector Configuration** The template configures a BGP group named `ibgp-overlay-rr-clients`. Within this group, two key parameters are set, as shown in the following code:
    - `route-reflector.client: true`: To instruct BGP to reflect routes received from any peer in this group to all other peers in the same group.
    - `route-reflector.cluster-id`: This is set to the spine's own Router ID. In a scenario with multiple RRs, the cluster ID prevents routing loops by ensuring a route is not reflected to the cluster from which it originated.

The template also activates the `evpn` address family for this group, as it is the only type of route that needs to be reflected.

```
"group": [
            {
              "group-name": "ibgp-overlay-rr-clients",
              "admin-state": "enable",
              "peer-as": {{ ibgp_asn }},
```

```
            "local-as": {
              "as-number": {{ ibgp_asn }}
            },
            "route-reflector": {
              "client": true,
              "cluster-id": "{{ devices.json.results[0].custom_fields.Router_ID }}"
            },
            "afi-safi": [
              {
                "afi-safi-name": "ipv4-unicast",
                "admin-state": "disable"
              },
              {
                "afi-safi-name": "evpn",
                "admin-state": "enable"
              }
            ]
          }
        ]
```

- **BGP Neighbor Peering with Leafs:** Finally, the template iterates through all devices in the NetBox inventory that have the leaf role. For each leaf found, it creates a neighbor entry, assigning it to the `ibgp-overlay-rr-clients` peer group. The peering is established using the loopback IP addresses (Router IDs) of the leaf and the spine, which are reachable via the underlay network. This process is illustrated below.

```
"neighbor": [
            {% for device in all_devices.json.results if device.role.display == "leaf" %}
            {
              "peer-address": "{{ device.custom_fields.Router_ID }}",
              "peer-group": "ibgp-overlay-rr-clients",
              "transport": {
                "local-address": "{{ devices.json.results[0].custom_fields.Router_ID }}"
              }
            }{% if not loop.last %},{% endif %}
            {% endfor %}
          ]
```

CHAPTER FOUR

4 Use cases and Results

This chapter presents the results obtained from the implementation of the automated datacenter fabric. The primary goal is to validate both the functional correctness of the deployed network and the operational benefits of the automation framework. The tests were made in the Containerlab virtual environment, and the results were verified through a series of use cases and operational commands on the Nokia SR Linux devices.

4.1 Day 2: Ongoing Operations

4.1.1 Use Case 1: Underlay Fabric Validation

The main purpose of this first use case is to validate the connectivity of the IP transport network, known as the underlay. The objective is to be sure that all leaf and spine switches have established BGP peering sessions and can reach each other's loopback IP addresses.

4.1.1.1 Adding Data to NetBox

Before running the ansible playbook to configure the underlay, some critical data was created in NetBox such as:

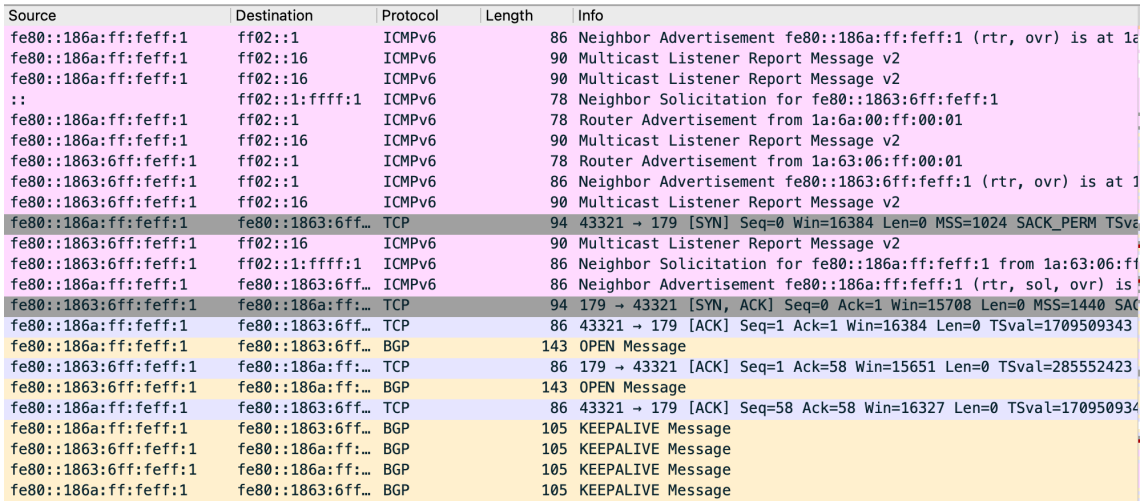
- `System0` interface for each device.
- A Loopback IP address (`10.0.0.XY`, where `XY` is the device number) was assigned to each `System0` interface.

- Custom fields on each device object were populated with the `Router_ID` (matching the loopback IP) and `DEVICE_ASN` for BGP autonomous system number.

4.1.1.2 Verification and Analysis of the Results

The `underlay_conf` role playbook was executed, generating and applying configurations for each device.

The captured traffic, shown in Figure 20, details the sequence of events that leads to a fully established BGP session without any pre-configured IPv4 or IPv6 addresses.



Source	Destination	Protocol	Length	Info
fe80::186a:ff:feff:1	ff02::1	ICMPv6	86	Neighbor Advertisement fe80::186a:ff:feff:1 (rtr, ovr) is at 1a
fe80::186a:ff:feff:1	ff02::16	ICMPv6	90	Multicast Listener Report Message v2
fe80::186a:ff:feff:1	ff02::16	ICMPv6	90	Multicast Listener Report Message v2
::	ff02::1:ffff:1	ICMPv6	78	Neighbor Solicitation for fe80::1863:6ff:feff:1
fe80::186a:ff:feff:1	ff02::1	ICMPv6	78	Router Advertisement from 1a:6a:00:ff:00:01
fe80::186a:ff:feff:1	ff02::16	ICMPv6	90	Multicast Listener Report Message v2
fe80::1863:6ff:feff:1	ff02::1	ICMPv6	78	Router Advertisement from 1a:63:06:ff:00:01
fe80::1863:6ff:feff:1	ff02::1	ICMPv6	86	Neighbor Advertisement fe80::1863:6ff:feff:1 (rtr, ovr) is at 1
fe80::1863:6ff:feff:1	ff02::16	ICMPv6	90	Multicast Listener Report Message v2
fe80::186a:ff:feff:1	fe80::1863:6ff…	TCP	94	43321 → 179 [SYN] Seq=0 Win=16384 Len=0 MSS=1024 SACK_PERM TSva
fe80::1863:6ff:feff:1	ff02::16	ICMPv6	90	Multicast Listener Report Message v2
fe80::1863:6ff:feff:1	ff02::1:ffff:1	ICMPv6	86	Neighbor Solicitation for fe80::186a:ff:feff:1 from 1a:63:06:ff
fe80::186a:ff:feff:1	fe80::1863:6ff…	ICMPv6	86	Neighbor Advertisement fe80::186a:ff:feff:1 (rtr, sol, ovr) is
fe80::1863:6ff:feff:1	fe80::186a:ff:…	TCP	94	179 → 43321 [SYN, ACK] Seq=0 Ack=1 Win=15708 Len=0 MSS=1440 SAC
fe80::186a:ff:feff:1	fe80::1863:6ff…	TCP	86	43321 → 179 [ACK] Seq=1 Ack=1 Win=16384 Len=0 TSval=1709509343
fe80::186a:ff:feff:1	fe80::1863:6ff…	BGP	143	OPEN Message
fe80::1863:6ff:feff:1	fe80::186a:ff:…	TCP	86	179 → 43321 [ACK] Seq=1 Ack=58 Win=15651 Len=0 TSval=285552423
fe80::1863:6ff:feff:1	fe80::186a:ff:…	BGP	143	OPEN Message
fe80::186a:ff:feff:1	fe80::1863:6ff…	TCP	86	43321 → 179 [ACK] Seq=58 Ack=58 Win=16327 Len=0 TSval=170950934
fe80::186a:ff:feff:1	fe80::1863:6ff…	BGP	105	KEEPALIVE Message
fe80::1863:6ff:feff:1	fe80::186a:ff:…	BGP	105	KEEPALIVE Message
fe80::1863:6ff:feff:1	fe80::186a:ff:…	BGP	105	KEEPALIVE Message
fe80::186a:ff:feff:1	fe80::1863:6ff…	BGP	105	KEEPALIVE Message

Figure 20 - Packet Capture of a BGP Unnumbered Session Establishment from LF-11 switch

The packet capture reveals the process described in section 2.2.1:

- **Neighbor Discovery (ICPMv6):** The initial packets are ICMPv6 Router Advertisement and Neighbor Solicitation messages. The devices use these standard IPv6 mechanisms to announce their presence on the link and discover each other's link-local addresses.
- **BGP Session Establishment (TCP/BGP):** Once the link-local addresses are discovered, the BGP process initiates the TCP three-way handshake on port 179.
    - A SYN packet is sent.
    - It is answered by an SYN + ACK.
    - The session is established with a final ACK.
    - Immediately following the successful TCP connection, the devices exchange BGP OPEN messages to negotiate session parameters, followed by a periodic KEEPALIVE messages to maintain the session.

To confirm the result of this process, the `show network-instance default route-table` command was executed on the leaf switch `lf-11`, the output shown in the Figure 21 confirms that the sessions were established successfully.

```
A:lf-11# / show network-instance default route-table
-------------------------------------------------------------------------------------------------------------------------------------------------------------
IPv4 unicast route table of network instance default
-------------------------------------------------------------------------------------------------------------------------------------------------------------
+----------------------+------+-----------+---------------------+---------+---------+--------+-----------+-----------------+-----------------+
|        Prefix        |  ID  |   Route   |     Route Owner     | Active  | Origin  | Metric |   Pref    |    Next-hop     |    Next-hop     |
|                      |      |   Type    |                     |         | Network |        |           |     (Type)      |    Interface    |
|                      |      |           |                     |         | Instanc |        |           |                 |                 |
|                      |      |           |                     |         |    e    |        |           |                 |                 |
+======================+======+===========+=====================+=========+=========+========+===========+=================+=================+
| 10.0.0.11/32         | 4    | host      | net_inst_mgr        | True    | default | 0      | 0         | None            | None            |
|                      |      |           |                     |         |         |        |           | (extract)       |                 |
| 10.0.0.12/32         | 0    | bgp       | bgp_mgr             | True    | default | 0      | 170       | fe80::1819:6    | ethernet-       |
|                      |      |           |                     |         |         |        |           | ff:feff:1       | 1/1.0           |
|                      |      |           |                     |         |         |        |           | (direct)        |                 |
| 10.0.0.13/32         | 0    | bgp       | bgp_mgr             | True    | default | 0      | 170       | fe80::1819:6    | ethernet-       |
|                      |      |           |                     |         |         |        |           | ff:feff:1       | 1/1.0           |
|                      |      |           |                     |         |         |        |           | (direct)        |                 |
| 10.0.0.14/32         | 0    | bgp       | bgp_mgr             | True    | default | 0      | 170       | fe80::183d:7    | ethernet-       |
|                      |      |           |                     |         |         |        |           | ff:feff:1       | 1/2.0           |
|                      |      |           |                     |         |         |        |           | (direct)        |                 |
```

Figure 21 - Route Table of lf-11 leaf switch

As expected, the route table on `lf-11` shows the routes to the loopback addresses of the other fabric devices. A route to `10.0.0.12` which is the loopback address for `lf-12`, and routes to `10.0.0.13` and `10.0.0.14` which are the addresses for the spine's switches, `sp-13` and `sp-14`, respectively. All these routes were learned via BGP, with the link-local IPv6 address to the next-hop, confirming that the BGP unnumbered configuration was successful and the underlay fabric is operational.

4.1.2 Use Case 2: Layer 2 Service Provisioning and Validation

This use case demonstrates the ability to provision a new, isolated Layer 2 tenant service across the fabric, by simply adding some information in NetBox.

4.1.2.1 Adding Data to NetBox

The following data was added to NetBox:

- **L2VPN**: A new object named `EVPN-VXLAN-SERVICE 1` was created, with Identifier (EVI) set to 111. A corresponding VRF named `vrf-1` and a Route Target of `65535:111` were also created and associated.
- **VLAN**: A VLAN object named `VXLAN service1` was created, with VID 1.
- **Linking L2VPN and VLAN**: `EVPN-VXLAN-SERVICE 1` and `VXLAN service1` were linked via `vni_vlan` and `bound_l2vpn` custom fields, respectively.

- **Interfaces:** The physical interfaces (in this case `ethernet-1/49`) on both `lf-11` and `lf-12` switches were set to Tagged mode and the `VXLAN service1` was added to their Tagged VLANs list.
- **ASN**: The iBGP Autonomous System Number was added in the ASN list (65535).

Note: Two test hosts, `mt1` and `mt2`, were configured to use the created VLAN and attached to `lf-11` and `lf-12` respectively.

4.1.2.2 Verification and Analysis of the Results

The `overlay_conf_leaf` and `overlay_conf_spine` roles playbooks were executed, generating and applying configurations for each device.

To understand how MAC address information is distributed, a packet capture was performed on the link between the Route Reflector (`sp-13`) and the leaf switch (`lf-11`). Figure 22 shows the BGP UPDATE message sent from the spine to the leaf, carrying the reachability information for the remote host `mt2`.

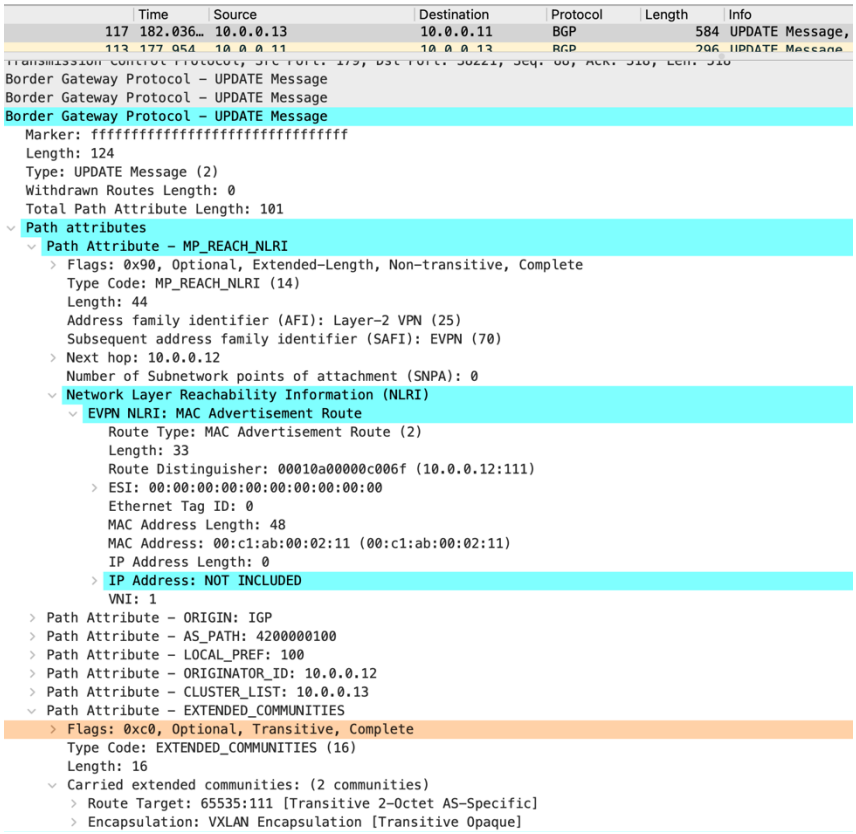


Figure 22 - Wireshark Capture of an EVPN Type 2 Route Advertisement

The captured BGP UPDATE message provides a view into the EVPN control plane such as:

- **MP_REACH_NLRI:** This BGP attribute confirms that the update is for the EVPN address family, also specifying the Next Hop (`10.0.0.12`) which is the VTEP address of the originating leaf switch, `lf-12`. This informs `lf-11` where to send the encapsulated data traffic to reach the destination MAC.
- **Route Type:** The packet is a MAC Advertisement Route (Type 2), which is the route type for advertising host reachability in a Layer 2 service.
- **MAC Address:** The carried MAC address `00:c1:ab:00:02:11` belongs to the remote host `mt2`.
- **VNI:** The advertisement is explicitly tied to VNI 1, ensuring the MAC address is installed in the correct bridge-table (`vrf-1`) on the receiving switch.
- **IP Address:** IP address field is NOT INCLUDED in this specific update. This demonstrates a feature of EVPN, where MAC-only reachability can be advertised independently, typically learned from pure Layer 2 frames before any IP-level traffic like ARP is seen.
- **Route Target:** The route is tagged with Route Target: `65535:111`, that act as a "service identifier," ensuring that only network instances (MAC-VRFs) configured to import this specific RT will install the route, providing tenant isolation.
- **Encapsulation Type:** The community for VXLAN Encapsulation is present, signaling to `lf-11` that data traffic destined for this MAC address must be encapsulated using VXLAN.

To confirm the result of this process, the `show network-instance vrf-1 bridge-table mac-table all` command was executed on the leaf switch `lf-11`, the output shown in the Figure 23 confirms that the sessions were established successfully.

```
A:lf-11# / show network-instance vrf-1 bridge-table mac-table all
------------------------------------------------------------------------------------------------------------
Mac-table of network instance vrf-1
------------------------------------------------------------------------------------------------------------
+-------------------+-----------------------------+-----------+----------+--------+-------+
|      Address      |         Destination         |   Dest    |  Type    | Active | Aging |
|                   |                             |   Index   |          |        |       |
+===================+=============================+===========+==========+========+=======+
| 00:C1:AB:00:01:11 | ethernet-1/49.1             | 7         | learnt   | true   | 300   |
| 00:C1:AB:00:02:11 | vxlan-interface:vxlan1.1    | 676007495 | evpn     | true   | N/A   |
|                   | vtep:10.0.0.12 vni:1        | 52        |          |        |       |
+-------------------+-----------------------------+-----------+----------+--------+-------+
Total Irb Macs                 :    0 Total    0 Active
Total Static Macs              :    0 Total    0 Active
Total Duplicate Macs           :    0 Total    0 Active
Total Learnt Macs              :    1 Total    1 Active
Total Evpn Macs                :    1 Total    1 Active
Total Evpn static Macs         :    0 Total    0 Active
Total Irb anycast Macs         :    0 Total    0 Active
Total Proxy Antispoof Macs     :    0 Total    0 Active
Total Reserved Macs            :    0 Total    0 Active
Total Eth-cfm Macs             :    0 Total    0 Active
```

Figure 23 - mac table of vrf-1 network instance

As expected, the MAC address of the local host 00:C1:AB:00:01:11 (mt1), was learned via the physical access interface ethernet-1/49.1. This is indicated by the learnt type.

The MAC address of the remote host, 00:C1:AB:00:02:11 (mt2), was learned via the evpn protocol. The switch correctly identifies the next-hop as the VTEP of the remote leaf switch, 10.0.0.12 (lf-12), and knows to encapsulate the traffic in VNI 1.

This successful distribution of MAC address information via BGP EVPN, followed by a successful ping test between mt1 and mt2, validates the entire overlay service path.

4.1.3 Use Case 3: Scaling the Fabric – Adding a New Leaf Switch and a New Service

For any modern datacenter fabric, the ability to scale horizontally with minimal operational overhead is a very important requirement. This use case demonstrates the framework's capacity to seamlessly integrate a new service and a new network element, in this case, a new leaf switch (lf-15), into the existing fabric.

4.1.3.1 Adding Data to NetBox

The following data was added to NetBox:

- A new Device object named clab-leirt-topology-lf-15, was created with the leaf role (Router_ID -> 10.0.0.15, DEVICE_ASN -> 4200000115).
- The system0 interface was created, with the address 10.0.0.15/32.
- Cable objects were created to model the physical connections from lf-15 interfaces (ethernet-1/1, ethernet-1/2) to the spine switches.
    - PEER_AS custom field on these interfaces were set to 4200000100 (Spine Autonomous System Number).
- Vxlan1 tunnel interface was created on the lf-15.
- A new object named EVPN-VXLAN-SERVICE 2 was created, with Identifier (EVI) set to 222. A corresponding VRF named vrf-2 and a Route Target of 65535:222 were also created and associated.
- A VLAN object named VXLAN service2 was created, with VID 2.
- EVPN-VXLAN-SERVICE 2 and VXLAN service2 were linked via vni_vlan and bound_l2vpn custom fields, respectively.

- The physical interface (in this case `ethernet-1/49`) on `lf-15` switch was set to Tagged mode and the `VXLAN service2` was added to his Tagged VLANs list.

4.1.3.2 Verification and Analysis of the Results

After executing the playbooks against the expanded inventory, a series of verification commands were run made the new leaf switch, `lf-15`, to confirm its successful and complete integration into the fabric's control planes.

To verify that `SERVICE2` was correctly extended to `lf-15` and that the EVPN control plane was properly distributing reachability information, a new test host (`mt3`) was attached to `lf-15`.

To provide a definitive proof of the control plane's scalability, a final packet capture was performed to observe how the reachability information for the new host (`mt3`) was propagated across the fabric. Figure 24 shows the BGP UPDATE message as it was reflected by the spine (`sp-13`) and received by one of the original leaf switches (`lf-11`).

```
| Time | Source | Destination | Protocol | Length | Info
97 46.1185… 10.0.0.13 10.0.0.11 BGP 584 UPDATE Message,

Path attributes
  Path Attribute - MP_REACH_NLRI
    Flags: 0x90, Optional, Extended-Length, Non-transitive, Complete
    Type Code: MP_REACH_NLRI (14)
    Length: 44
    Address family identifier (AFI): Layer-2 VPN (25)
    Subsequent address family identifier (SAFI): EVPN (70)
    Next hop: 10.0.0.15
    Number of Subnetwork points of attachment (SNPA): 0
    Network Layer Reachability Information (NLRI)
      EVPN NLRI: MAC Advertisement Route
        Route Type: MAC Advertisement Route (2)
        Length: 33
        Route Distinguisher: 00010a00000f00de (10.0.0.15:222)
        ESI: 00:00:00:00:00:00:00:00:00:00
        Ethernet Tag ID: 0
        MAC Address Length: 48
        MAC Address: 00:c1:ab:00:03:12 (00:c1:ab:00:03:12)
        IP Address Length: 0
        IP Address: NOT INCLUDED
        VNI: 2
  Path Attribute - ORIGIN: IGP
  Path Attribute - AS_PATH: 4200000100
  Path Attribute - LOCAL_PREF: 100
  Path Attribute - ORIGINATOR_ID: 10.0.0.15
  Path Attribute - CLUSTER_LIST: 10.0.0.13
  Path Attribute - EXTENDED_COMMUNITIES
    Flags: 0xc0, Optional, Transitive, Complete
    Type Code: EXTENDED_COMMUNITIES (16)
    Length: 16
    Carried extended communities: (2 communities)
      Route Target: 65535:222 [Transitive 2-Octet AS-Specific]
      Encapsulation: VXLAN Encapsulation [Transitive Opaque]
```

Figure 24 - Wireshark Capture of a Reflected EVPN Route from the New Leaf

The capture shows a Type 2 route that originated from the new leaf `lf-15`:

- **MAC Address:** The MAC address of the new host `00:C1:AB:00:03:12` (`mt3`) is advertised.
- **Next Hop:** The next hop address is `10.0.0.15`, the VTEP address for `lf-15` switch.
- **VNI:** The route is associated with VNI 2, corresponding to `SERVICE2`.
- **Route Target:** The route is tagged with Route Target: `65535:222`, ensuring it is only imported into the `vrf-2` network instance.

The `show network-instance default protocols bgp neighbor` command was executed on `lf-15` switch to provide a view of its peering status. The output, shown in Figure 25, serves as a validation of the entire process.

```
A:lf-15# / show network-instance default protocols bgp neighbor
-------------------------------------------------------------------------------------------------------------------------------------------------------------
BGP neighbor summary for network-instance "default"
Flags: S static, D dynamic, L discovered by LLDP, B BFD enabled, - disabled, * slow
-------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------------
+------------------+---------------------------+-----------------+-------+-----------+-------------+--------------+-------------+--------------------+
|     Net-Inst     |           Peer            |      Group      | Flags |  Peer-AS  |    State    |    Uptime    |  AFI/SAFI   |      [Rx/Active/Tx]|
+==================+===========================+=================+=======+===========+=============+==============+=============+====================+
| default          | 10.0.0.13                 | ibgp-overlay    | S     | 65535     | established | 0d:0h:7m:38s | evpn        | [2/2/2]            |
| default          | 10.0.0.14                 | ibgp-overlay    | S     | 65535     | established | 0d:0h:7m:38s | evpn        | [2/0/2]            |
| default          | fe80::1823:6ff:feff:3%et  | underlay        | D     | 42000001  | established | 0d:0h:7m:47s | ipv4-       | [3/3/2]            |
|                  | hernet-1/1.0              |                 |       | 00        |             |              | unicast     |                    |
| default          | fe80::184e:7ff:feff:3%et  | underlay        | D     | 42000001  | established | 0d:0h:7m:48s | ipv4-       | [3/1/4]            |
|                  | hernet-1/2.0              |                 |       | 00        |             |              | unicast     |                    |
+------------------+---------------------------+-----------------+-------+-----------+-------------+--------------+-------------+--------------------+
-------------------------------------------------------------------------------------------------------------------------------------------------------------
Summary:
2 configured neighbors, 2 configured sessions are established,0 disabled peers
2 dynamic peers
```

Figure 25 - BGP Neighbor Summary on the New Leaf Switch (lf-15)

- **Underlay Peering (Dynamic)**: The two entries marked with a D (Dynamic) flag show the BGP sessions established for the underlay. The peers are the IPv6 link-local addresses of the spine switches, discovered automatically via BGP Unnumbered. The State is established, and the AFI/SAFI is ipv4-unicast, confirming that the IP transport fabric is operational.

- **Overlay Peering (Static)**: The two entries marked with an S (Static) flag show the iBGP sessions for the overlay control plane. The peers are the loopback addresses of the spine switches (`10.0.0.13` and `10.0.0.14`), which act as Route Reflectors. The State is established, proving that the leaf can communicate with its Route Reflectors.

- **Route Exchange**: The `[Rx/Active/Tx]` columns show that routes are being successfully received, activated, and transmitted over these sessions.

The `show network-instance vrf-2 bridge-table mac-table all` command was executed. The output is shown in Figure 26.

```
A:lf-15# show network-instance vrf-2 bridge-table mac-table all
------------------------------------------------------------------------------------------
Mac-table of network instance vrf-2
------------------------------------------------------------------------------------------
+-------------------+-----------------------------+-----------+---------+--------+-------
|      Address      |         Destination         |   Dest    |  Type   | Active | Aging
|                   |                             |   Index   |         |        |
+===================+=============================+===========+=========+========+=======
| 00:C1:AB:00:01:12 | vxlan-interface:vxlan1.2    | 678746135 | evpn    | true   | N/A
|                   | vtep:10.0.0.11 vni:2        | 17        |         |        |
| 00:C1:AB:00:03:12 | ethernet-1/49.2             | 6         | learnt  | true   | 300
+-------------------+-----------------------------+-----------+---------+--------+-------
Total Irb Macs                 :    0 Total    0 Active
Total Static Macs              :    0 Total    0 Active
Total Duplicate Macs           :    0 Total    0 Active
Total Learnt Macs              :    1 Total    1 Active
Total Evpn Macs                :    1 Total    1 Active
Total Evpn static Macs         :    0 Total    0 Active
Total Irb anycast Macs         :    0 Total    0 Active
Total Proxy Antispoof Macs     :    0 Total    0 Active
Total Reserved Macs            :    0 Total    0 Active
Total Eth-cfm Macs             :    0 Total    0 Active
```

Figure 26 - MAC Address Table of vrf-2 on the New Leaf Switch (lf-15)

- **MAC Learning**: The MAC address of the new local host, `00:C1:AB:00:03:12` (mt3), was correctly learned via the physical access interface `ethernet-1/49.2`.
- **MAC Learning via EVPN:** The MAC address of the host connected to SERVICE2 (`00:C1:AB:00:01:12` for `mt1`) was learned via the `evpn` protocol. The switch correctly identifies the next-hop for this remote MAC as the VTEP IP address of `lf-11` (`10.0.0.11`)

This successful population of the MAC table, followed by a successful two-way ping test between `mt1` and `mt3`, validates that the new leaf switch was not only added to the fabric but that the new Layer 2 was successfully operational.

4.1.4 Operational Benefits and Framework Analysis

The successful execution of the use cases demonstrates the technical correctness of the automation framework. However, the true value of a data-driven approach lies in its operational benefits, especially when compared to traditional, manual configuration methods.

4.1.4.1 Reduction in Provisioning Time

In a manual workflow, provisioning a new end-to-end Layer 2 service would require to:

- Consult spreadsheets or documentation to find available VLANs and VNIs.
- Log into multiple devices via CLI.
- Manually type dozens of lines of configuration on each device to create sub-interfaces, MAC-VRFs and configure BGP parameters.
- Manually run verification commands on each device.

This process is very time-consuming but also scales with the number of devices, taking up easily up to 20-30 minutes for a single service

With the automated framework, this entire process is reduced to a few minutes of data entry in NetBox, followed by a single Ansible command. As illustrated in Figure 27, the time savings are substantial.

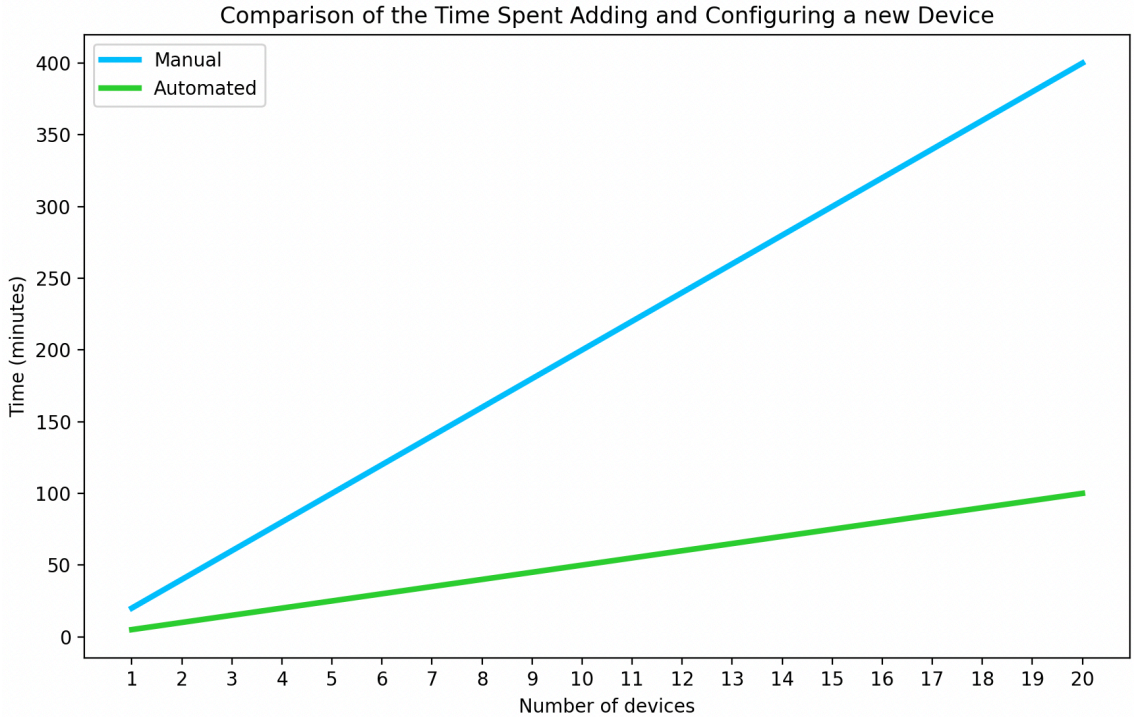


Figure 27 - Comparison of the Time Spent Adding and Configuring a new Device

Note: Manual configuration: 20 minutes each device; Automated configuration: 5 minutes each device

CHAPTER FIVE

5 Conclusion and Future Work

This final course project effectively showcased the creation of a comprehensive, data-driven automation framework for a datacenter fabric. By utilizing NetBox as a centralized Source of Truth and Ansible as the automation engine, the solution successfully deployed a robust EVPN-VXLAN network from scratch, facilitated the provisioning of isolated multi-tenant Layer 2 services, and allowed for easy horizontal scaling by integrating new network components effortlessly. The use cases outlined in the previous chapter affirmed not only the technical accuracy of the configuration but also highlighted substantial operational advantages in terms of speed, consistency, and reliability compared to conventional manual approaches.

The framework developed offers a strong basis for network automation. Yet, there are numerous opportunities for future improvements that could expand its capabilities, strengthen its robustness, and expand its scope.

5.1 Implementation of Layer 3 Services (IP-VRF)

The current approach is solely focused on delivering Layer 2 services (MAC-VRFs), which extend broadcast domains throughout the fabric. A logical and advantageous next step would be to expand the framework to incorporate Layer 3 services (IP-VRFs):

- **Data Model Extension:** The NetBox model must be refined to integrate with Layer 3 elements. Although VRF objects are already in use, they should have stronger ties to Layer 3 concepts. This would involve associating prefixes and IP addresses within NetBox directly with specific VRFs, effectively modeling the IP address space for each tenant.

- **Template Logic for IP-VRF:** Updates to the Ansible templates will be necessary to implement new logic for configuring IRB sub-interfaces within the MAC-VRF or to establish separate IP-VRF network instances.

Implementing L3 services would transform the fabric from a pure Layer 2 overlay into a complete multi-tenant cloud networking platform, capable of handling both bridged and routed tenant traffic.

5.2 Automated Configuration and Testing

While the current workflow is idempotent, its primary validation is the successful application of the configuration. A significant improvement would be the integration of an automated testing and state validation phase directly into the Ansible playbooks that not only configures but also verifies and reports on the network's operational state against the intended state in the Source of Truth.

This could be implemented through a dedicated Ansible playbook or role which would do:

- **Configuration Drift Detection**: Validate where the running configuration on a device has diverged from the state defined in NetBox.

- **Testing:** After a configuration is applied, a dedicated testing role could automatically verify that the change had the intended effect. For a new L2 service, this could involve:
    - Checking that the new mac-vrf instance exists.
    - Verifying that remote MAC addresses are being learned via EVPN in the correct bridge-table.
    - Automatically running ping or connectivity tests between hosts.

Bibliography

1) BGP Unnumbered, https://documentation.nokia.com/srlinux/24-3/books/routing-protocols/bgp.html#bgp-unnumbered-peer
2) Dutt, D. G. (2020). *Cloud Native Data Center Networking: Architecture, Protocols, and Tools*. O'Reilly Media
3) Getting Started With Network Automation: Netbox and Ansible, https://www.youtube.com/watch?v=BtzKX3Unuu0
4) JSON Interface, https://documentation.nokia.com/srlinux/23-10/books/system-mgmt/json-interface.html
5) Use of BGP for Routing in Large-Scale Data Centers (RFC 7938). IETF. https://datatracker.ietf.org/doc/html/rfc7938
6) Learn SR Linux – L2 EVPN with SR Linux, https://learn.srlinux.dev/tutorials/l2evpn/evpn/
7) Learn SR Linux – Overlay Routing, https://learn.srlinux.dev/tutorials/l3evpn/rt5-only/overlay/#__tabbed_2_1
8) Learn SR Linux – Underlay Routing, https://learn.srlinux.dev/tutorials/l3evpn/rt5-only/underlay/#__tabbed_9_3
9) NetBox – Zero To Hero Training, https://netboxlabs.com/netbox-zero-to-hero-training/
10) Nokia. (2025). *Nokia Data Center Fabric Fundamentals 4.0*. Nokia Learning Services.
11) Using Ansible Modules and Plugins, https://docs.ansible.com/ansible/latest/module_plugin_guide/index.html
12) Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks (RFC 7348). IETF. https://datatracker.ietf.org/doc/html/rfc7348

Appendices

Code Repository: https://github.com/MarcoMartinsCorreia/projeto-final-de-curso