Nokia Validated Design

3-stage EVPN/VXLAN Fabric

3HE-21632-AAAA-TQZZA
Issue 1
March 2025

Legal notice

Nokia is committed to diversity and inclusion. We are continuously reviewing our customer documentation and consulting with standards bodies to ensure that terminology is inclusive and aligned with the industry. Our future customer documentation will be updated accordingly.

Contents

1 Executive summary

Nokia Validated Designs (NVDs) is a workstream dedicated to producing validated recommendations to the consumer about Nokia's portfolio across market segments.

This is accomplished with extensive requirement analysis from a multitude of customers along with deep research of the technology development in the industry segment to form the solutions design.

Once the design has been compiled, it goes through an intense array of hardware, software, traffic and failure tests to form the validated design. The resultant design and collateral provide the consumer with a template which can be used to deploy the solution in their own environment.

NVDs are structured as core and ancillary (extension) designs. This document is based on a 3-stage Clos EVPN VXLAN design, covering various physical and logical connectivity aspects and associated technologies involved in a single site, multi-tiered data center architecture with EVPN as the control plane and VXLAN as the data plane.

2 Reference architecture overview

2.1 Design considerations and components

A high-level overview of the topology is shown in Figure 1.

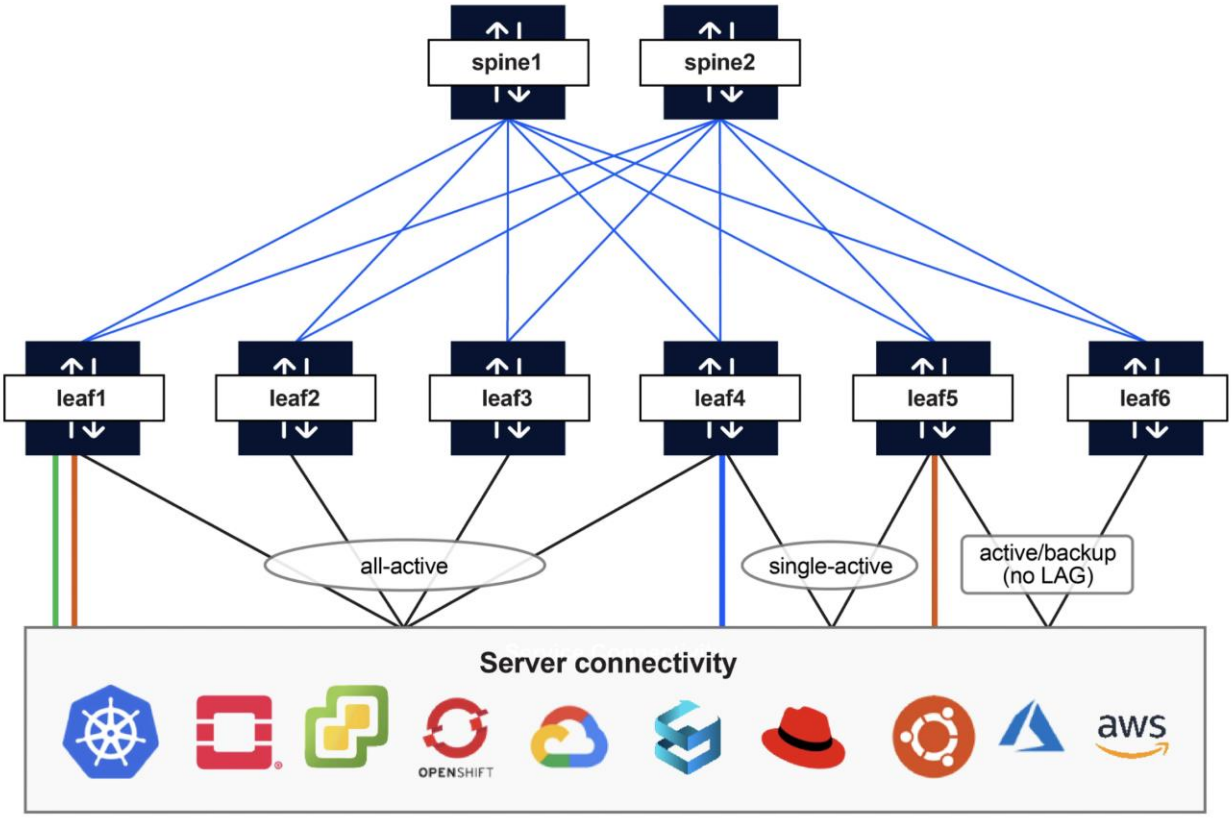

Figure 1. 3-stage EVPN VXLAN NVD architecture

This section describes the various components involved in this validated design and the design and technology choices that were made.

- The design strategically positions multiple Nokia data center platforms at the spine and leaf layers of a 3-stage Clos fabric. The purpose of positioning multiple variants of platforms is to help the consumer make informed decisions according to their sizing, scale, and needs.
- This also shows seamless interoperability with Broadcom Tomahawk platforms on the spines and Broadcom Trident platforms at the leaf layers.
- Server connectivity options are tested across all platforms.
- Trident-based platforms (7220 IXR-D3Ls, D4s and D5s) are positioned at the leaf layer for VXLAN support, and a Tomahawk-based platform (7220 IXR-H4) is at the spine layer for higher port radix since the fabric uses lean spines with need for only IP forwarding functionality (as this is an Edge-Routed Bridging [ERB] design).
- The design uses an IPv6-only underlay using IPv6 link-local addressing and Neighbor Discovery (ND).
- A single MP-BGP session is dynamically established using these IPv6 link-local addresses, and it can carry multiple AFIs/SAFIs (IPv4, IPv6, EVPN) as needed.
- During the establishment of this session, the extended next-hop encoding capability is exchanged, enabling IPv4 routes to be advertised with IPv6 next hops (RFC 8950). This enables the fabric underlay to be IPv6 link-local only, allowing operators to move away from the operational overhead of IPv4 underlay management while still providing an IPv4 overlay.
- This design covers the following server connectivity options:
    - Layer 2 untagged
    - Layer 2 tagged
    - Layer 3 point-to-point with static routes on the leaf (for subnets behind the server) exported as EVPN Type-5 routes into the fabric
    - 4-way ES-based LAG in all-active multihoming mode
    - 2-way ES-based LAG in single-active multihoming mode
    - Layer 2 untagged/tagged active/backup (Linux bond mode 1) with no LAG

2.2 High-level operational workflow

Figure 2 depicts a high-level operational workflow for the NVD-based fabric deployment and lifecycle management. The intent-based approach, combined with the prescriptive nature of the validated design and the flexibility of Nokia's Event Driven Automation (EDA), makes the deployment of the fabric effortless and reliable.

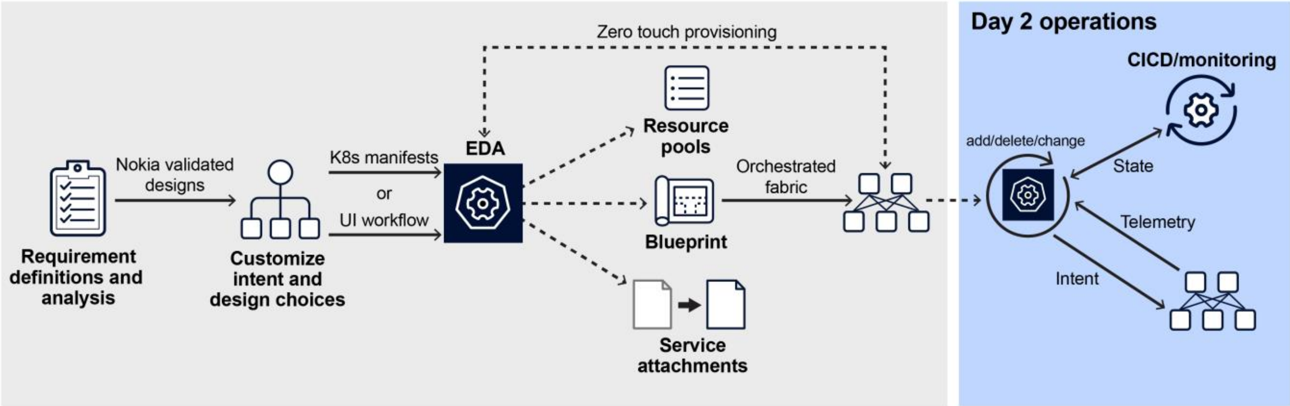

Figure 2. High-level operational flow diagram

- As with traditional deployments, broad customer requirements are gathered based on the applications and workloads that are going to be operational in the data center.
- Once analyzed and collated into infrastructure requirements, they are converted to an intent by customizing the closest available Nokia validated design (in this case, the 3-stage EVPN VXLAN NVD).
- Once the customization is complete, the intent can be described in EDA by using EDA K8s manifest files, REST APIs or the UI.
- EDA will then generate and push the per-node configuration (these are nodes already onboarded onto EDA using ZTP).
- Once the fabric is deployed, EDA provides comprehensive telemetry options that can be connected to CI/CD pipelines to modify the intent and the fabric as needed.
- Since EDA as a platform does not need to be reinstalled for new patches or apps, it provides a high degree of flexibility and customizability for modern DC fabric needs.

3 Network deployment

3.1 High-level design

Figures 3 and 4 depict a high-level design of the fabric. The topology is a 3-stage Clos fabric with BGP EVPN as the control plane and VXLAN as the data plane encapsulation method with point-to-point layer 3 links between the leafs and spines. These point-to-point interfaces are configured with IPv6-link local addressing (as shown in Figure 3), with each leaf advertising its IPv4 loopback address with an IPv6 next-hop (RFC 8950), as shown in Figure 4, using leaf1 as a reference. Each node in Figure 4 is labeled with sample IPv4 addresses assigned to the loopback interface.

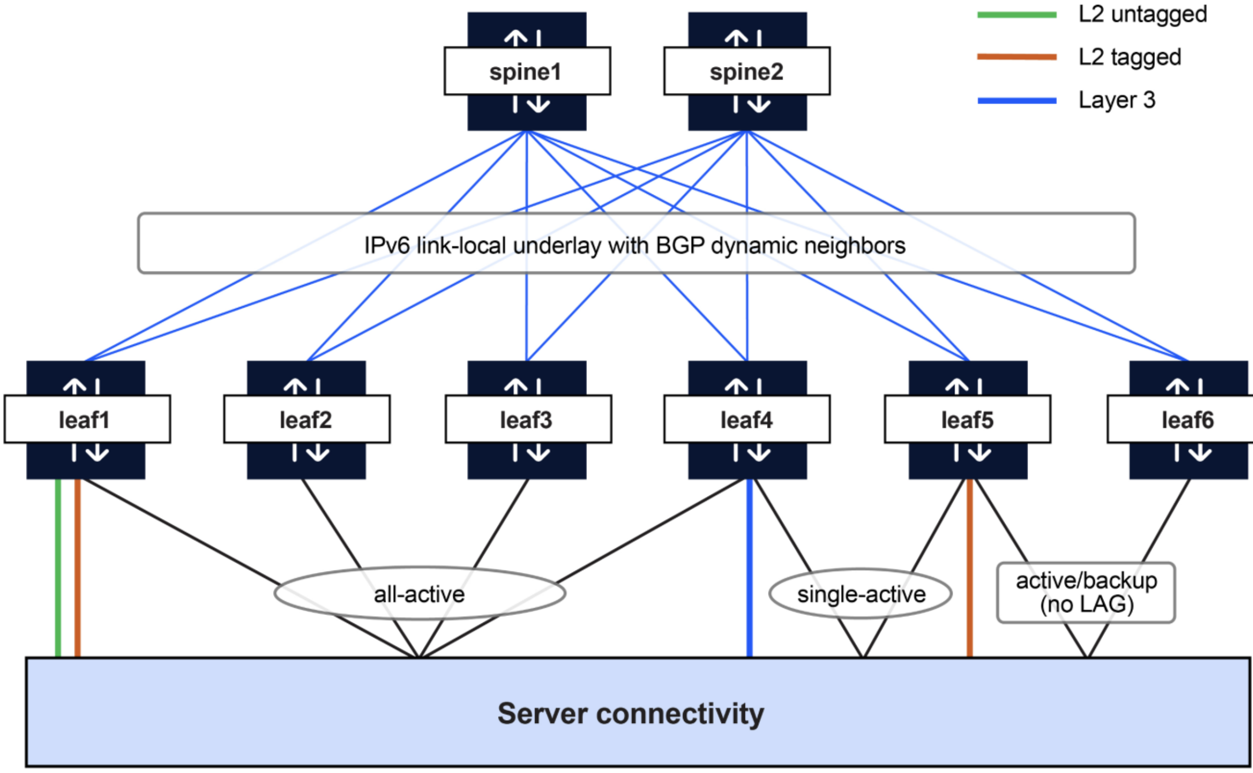

Figure 3. High-level diagram with underlay and overlay

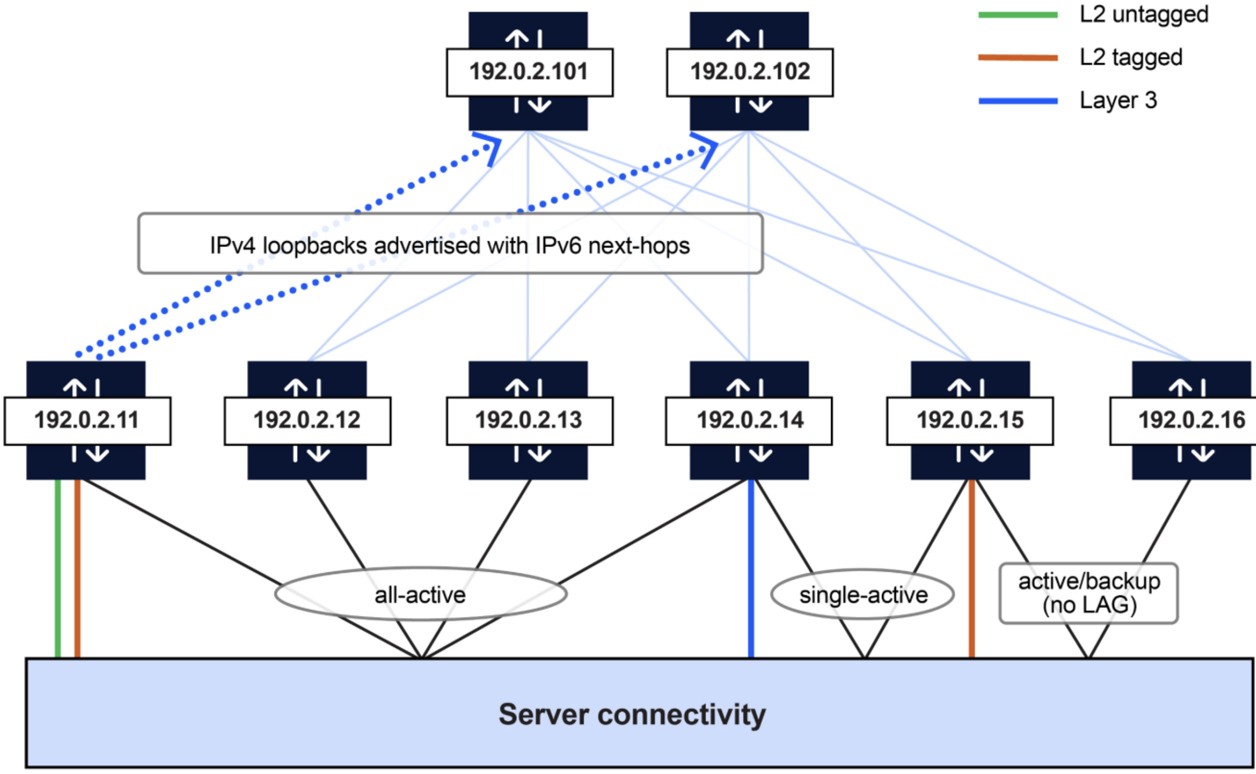

Figure 4. High-level diagram with underlay and overlay

This is an Edge-Routed Bridging (ERB) design with Integrated Routing and Bridging (IRB) interfaces configured on the leafs using a distributed anycast gateway model. All server connectivity terminates at the leafs, where the leafs act as VXLAN tunnel endpoints (VTEPs).

For routing between VNIs, this design uses an asymmetric routing model (as described in RFC 9135), along with symmetric routing using EVPN Type-5 routes for certain subnets.

3.2 Platform positioning

This section describes the Nokia platforms positioned for different roles in the 3-stage EVPN VXLAN validated design. Figure 5 provides a visual depiction while Table 1 lists all platforms and their count in the fabric.

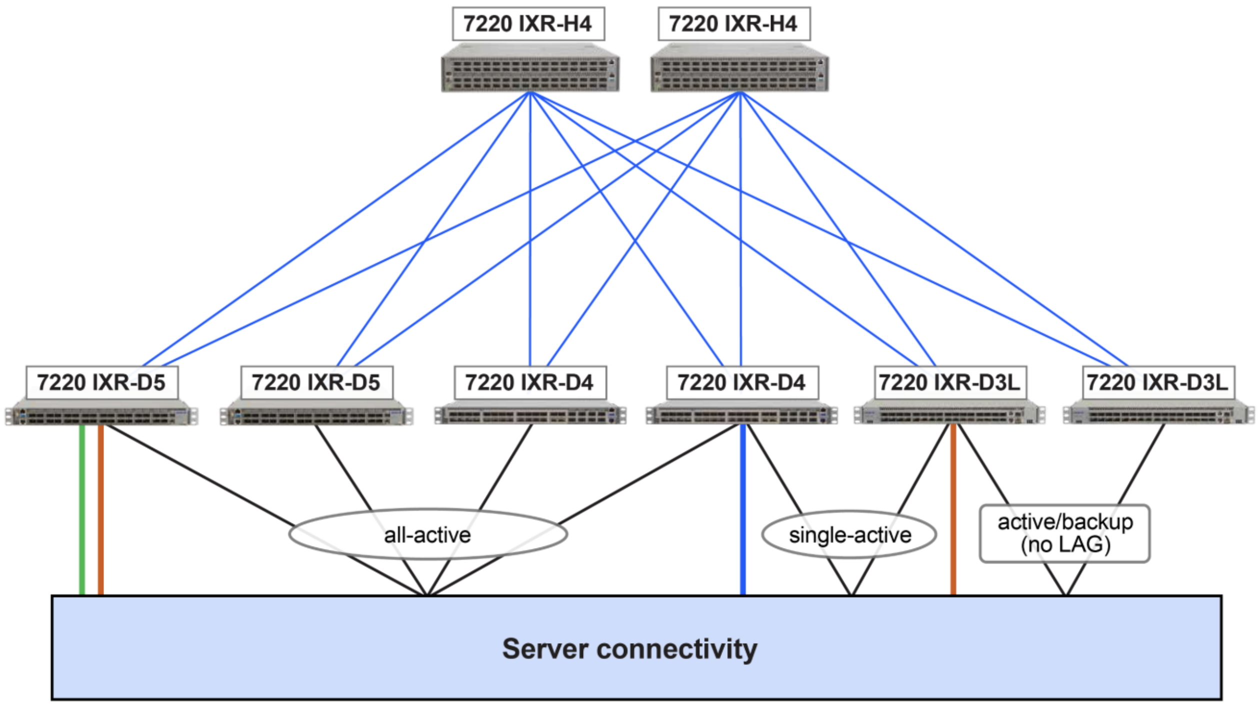

Figure 5. High-level diagram depicting platform positioning

Device	Role	Count
7220-IXR-H4	Spine	2
7220-IXR-D5	Leaf	2
7220-IXR-D4	Leaf	2
7220-IXR-D3L	Leaf	2

Table 1. Platform positioning

Note: Alternate platforms can be positioned in the roles shown above based on cost, hardware, and performance requirements.

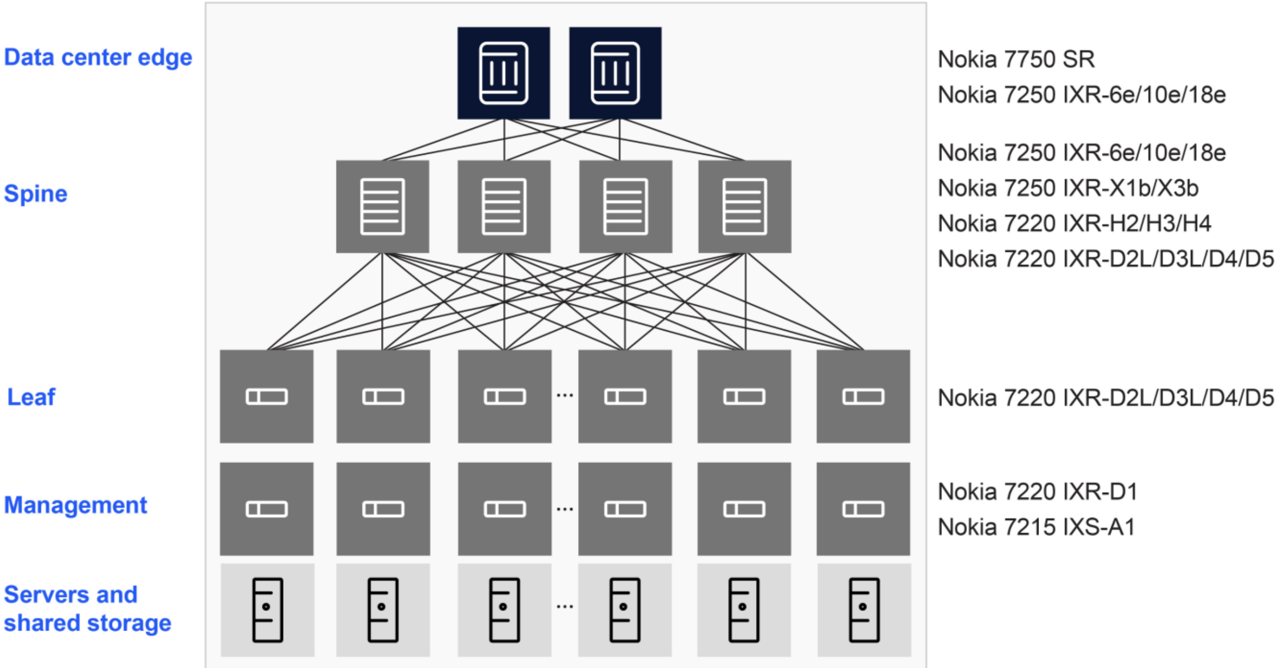

Figure 6. Nokia data center portfolio

3.3 Network architecture

In this section, we describe common traffic patterns that are validated in the 3-stage EVPN VXLAN NVD.

These traffic patterns include forwarding across Layer 2 tagged and untagged interfaces, Layer 3 interfaces, 4-way all-active Ethernet Segment LAG, 2-way single-active Ethernet Segment LAG, and active/backup server NIC-bonding with no Link Aggregation Group (LAG).

Figure 7 and 8 show traffic ingress on a single-homed interface and egress out of an Ethernet Segment member interface (either local or remote).

- When the ingress leaf (VTEP) is part of the egress Ethernet Segment, forwarding follows the local-bias rules, where the local egress member of the Ethernet Segment is selected as the exit interface.
- In Figure 8, if the local egress member link of the Ethernet Segment is not available (down), packets are forwarded over the fabric by encapsulating with VXLAN headers towards a remote leaf (VTEP) that is also part of the same Ethernet Segment, eventually leaving via the member interface of this Ethernet Segment.

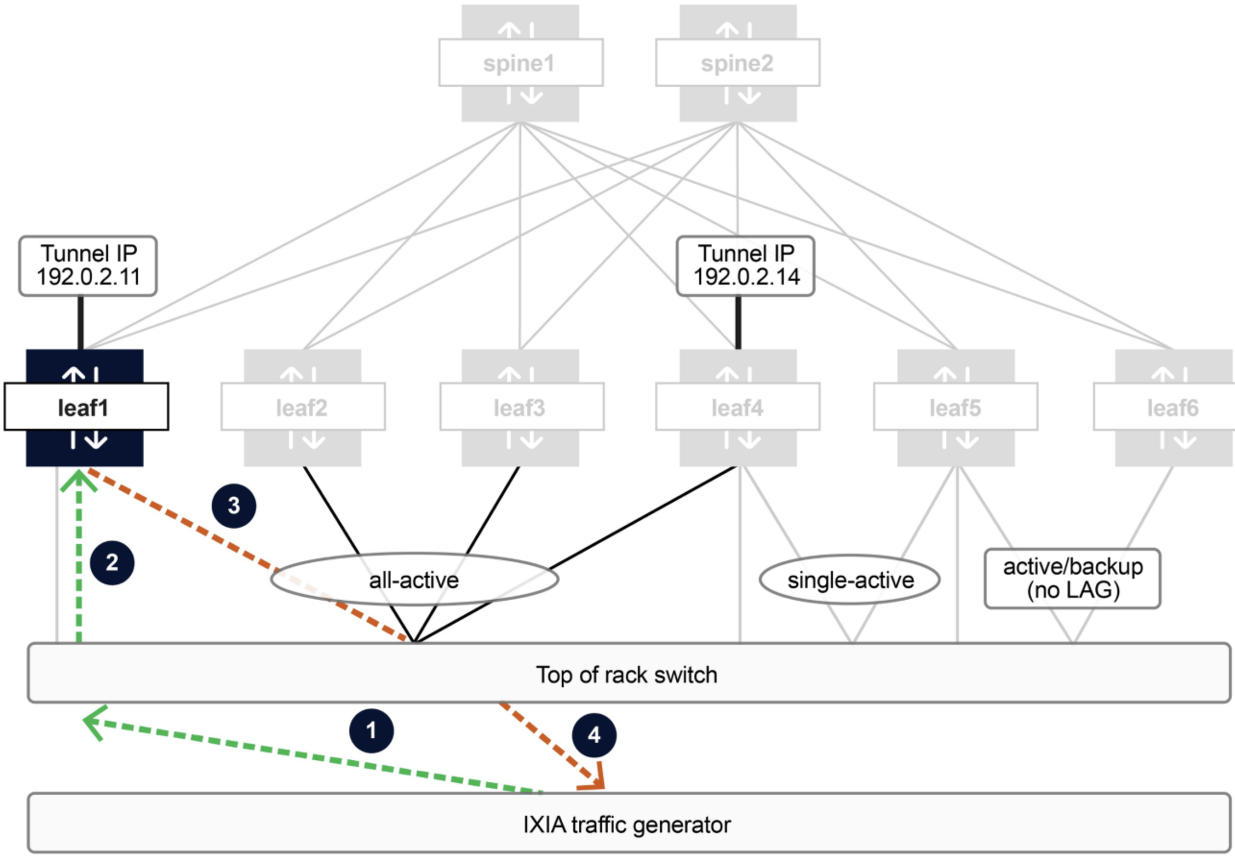

Figure 7. Packet flow for Layer 2 tagged and untagged traffic using local-bias forwarding

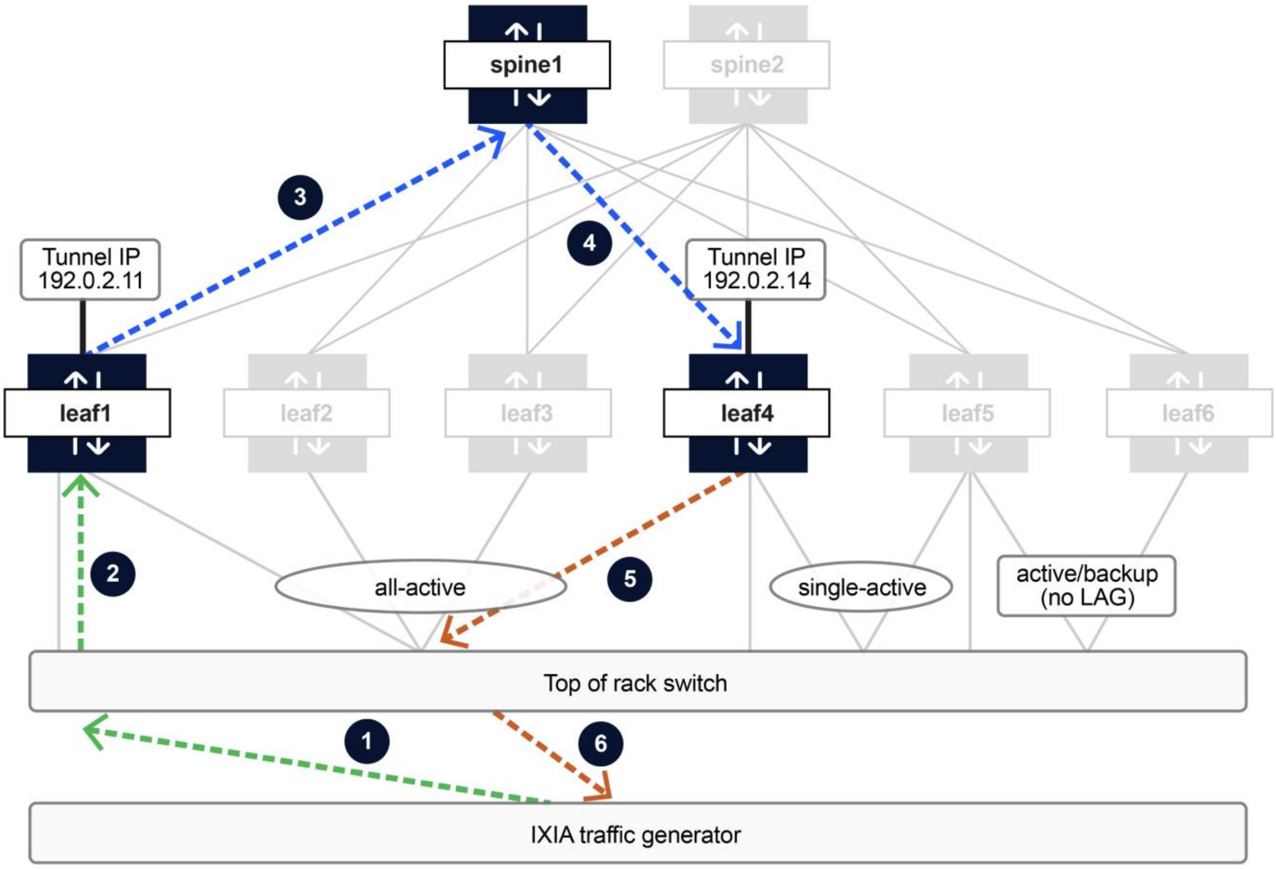

Figure 8. Packet flow for Layer 2 tagged and untagged traffic exiting via a remote VTEP when local member interface of Ethernet Segment is down on ingress VTEP

Figure 9 demonstrates traffic ingress on a 4-way all-active Ethernet Segment with the egress via a single-homed Layer 3 interface on a remote VTEP. In this case, the destination that is connected via a Layer 3 interface will be learnt using EVPN Type-5 routes.

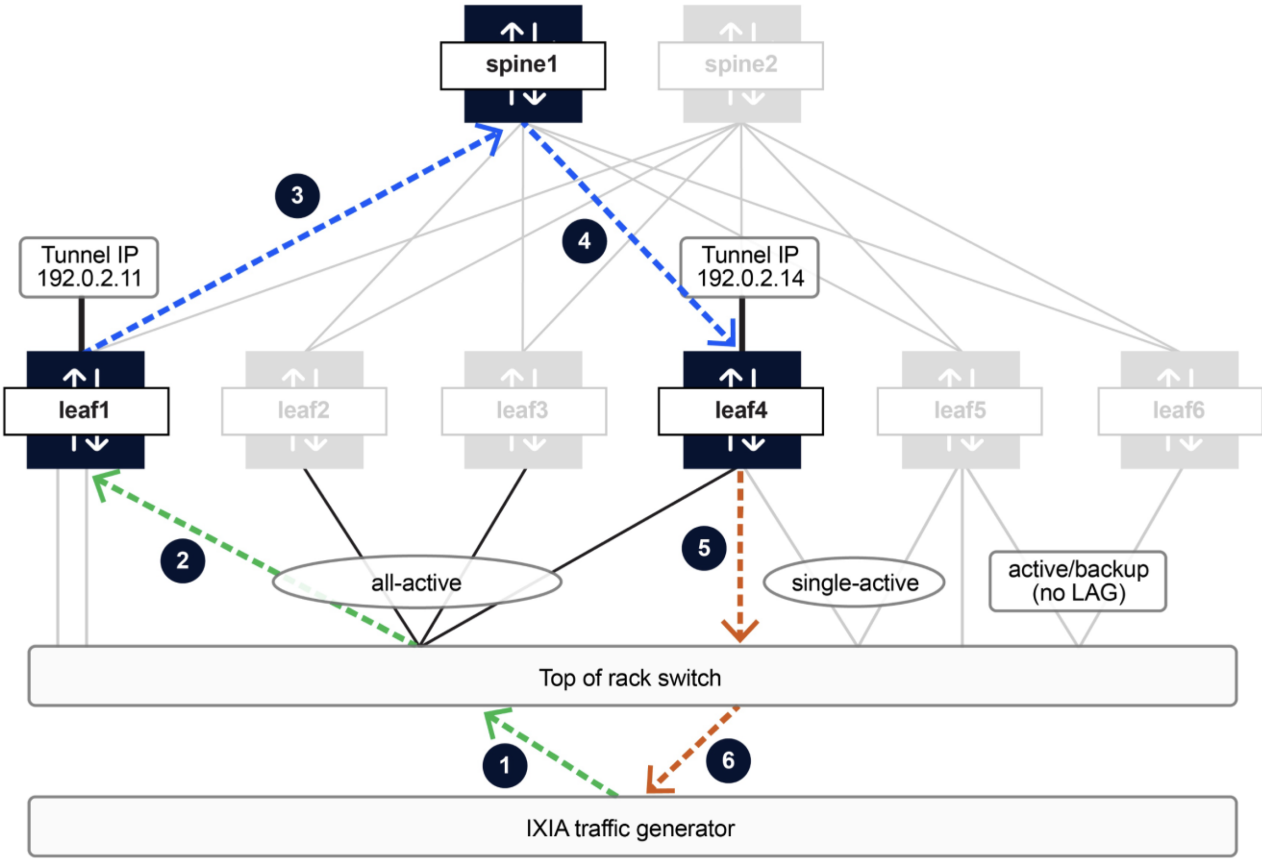

Figure 9. Packet flow for traffic ingress on a 4-way Ethernet Segment member interface directed to a destination behind a Layer 3 interface on a remote VTEP

Figure 10 and **Figure 11** demonstrate the traffic patterns for a destination that is behind a single-active Ethernet Segment.

- Figure 10 demonstrates traffic ingress via the active VTEP of a single-active Ethernet Segment and uses local-bias forwarding rules to send out another locally attached Ethernet Segment.
- Figure 11 demonstrates traffic ingress on a single-homed interface. It is forwarded over the fabric by encapsulating VXLAN headers with the egress via the interface of the single-active Ethernet Segment of the active, remote VTEP. On the ingress VTEP, the Ethernet Segment resolves to the VTEP address of the active node only.

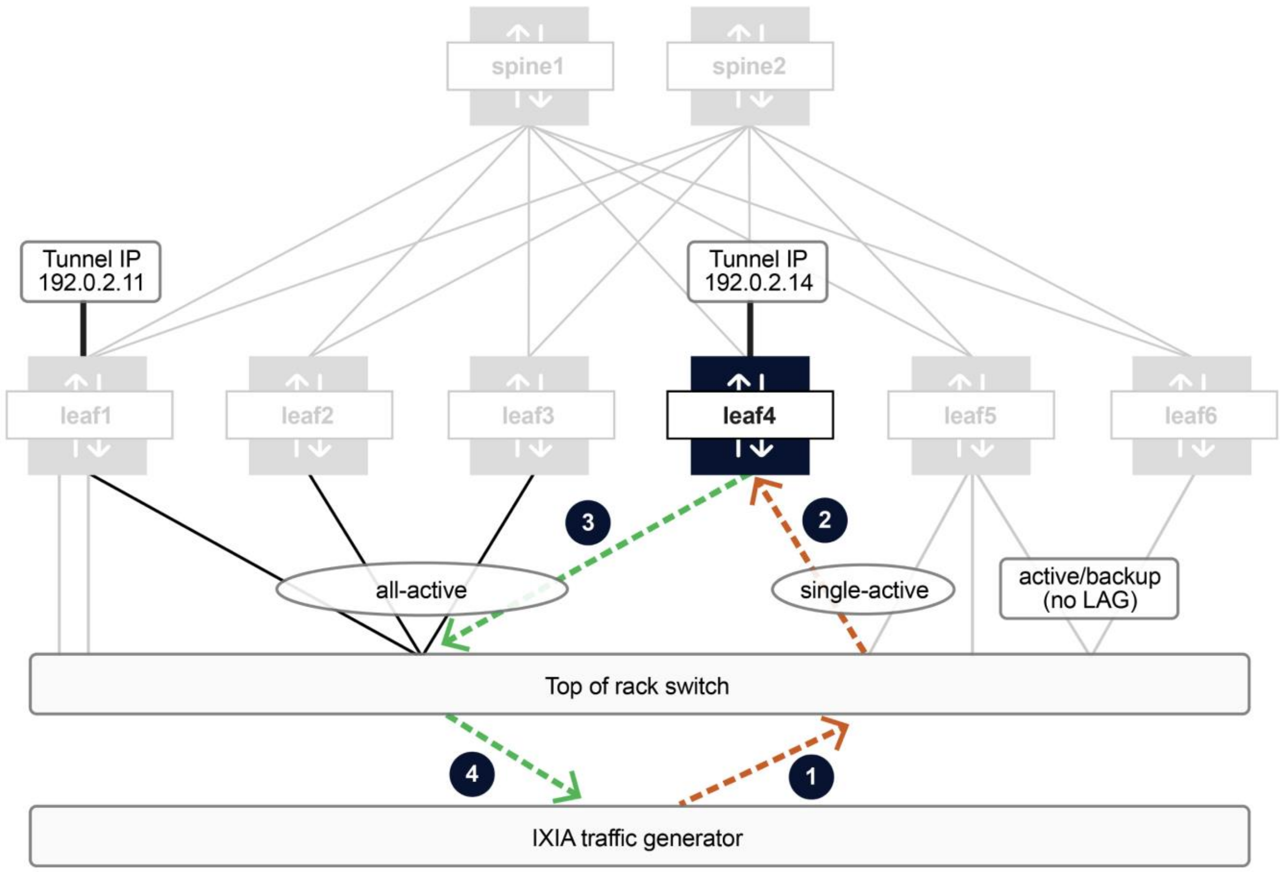

Figure 10. Single-active Ethernet Segment with local-bias forwarding

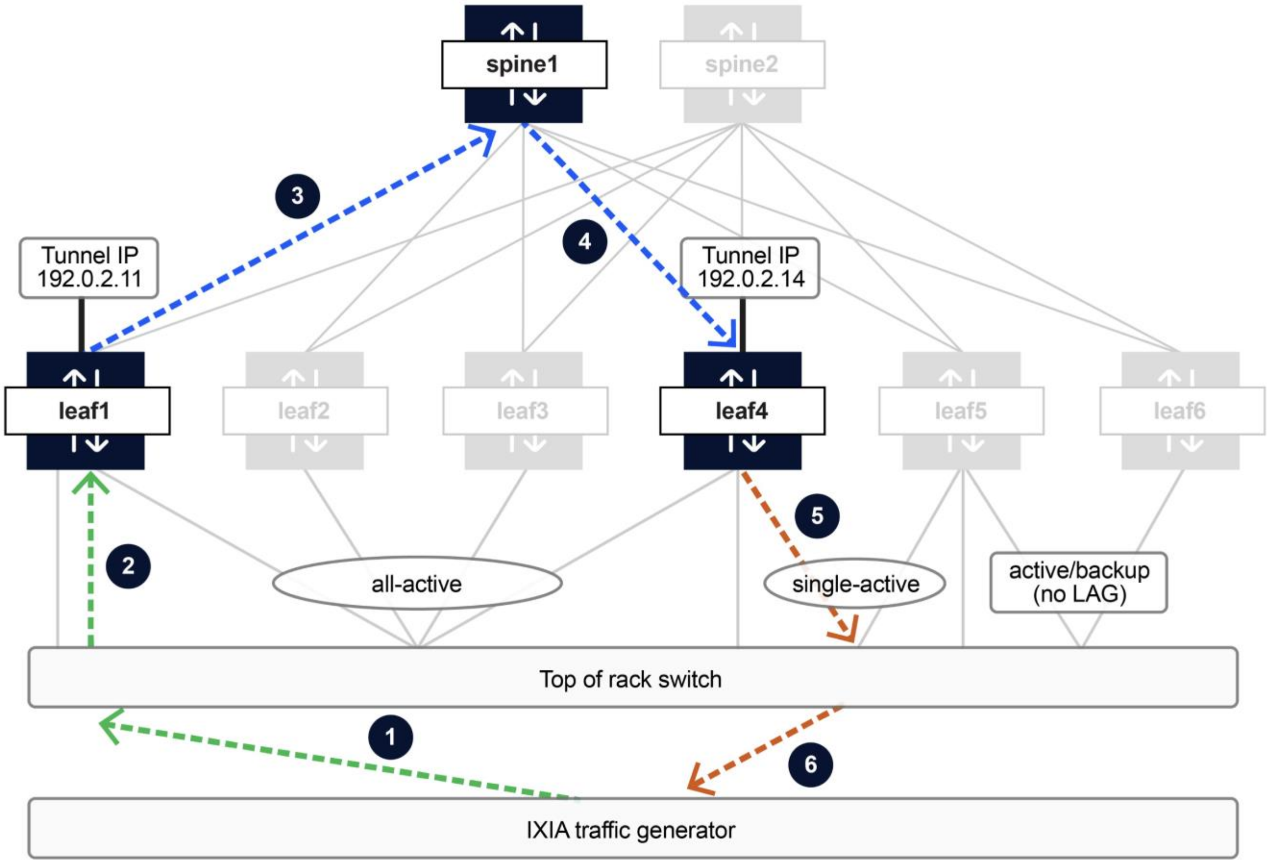

Figure 11. Single-active Ethernet Segment with forwarding over fabric

4 Feature configuration

4.1 Underlay with IPv6 link-local addressing for P2P interfaces between leafs and spines

The point-to-point interfaces between the leafs and the spines are enabled for IPv6 only, with link-local addressing. IPv6 Neighbor Discovery (ND) is used to resolve the peers' address. The addressing is enabled on subinterfaces within each physical interface. These subinterfaces are then mapped to the default network-instance.

The system0 interface, used as the VTEP address, is configured with a /32 address. These addresses are used as the source and destination addresses in the outer IP header for VXLAN tunnels. In this document, the IPv4 documentation range 192.0.2.0/24 is used for assignment.

```
// uplink to spine1

A:leaf1# info interface ethernet-1/29
    interface ethernet-1/29 {
        admin-state enable
        subinterface 0 {
            admin-state enable
            ipv6 {
```

```
                admin-state enable
                router-advertisement {
                    router-role {
                        admin-state enable
                        max-advertisement-interval 10
                        min-advertisement-interval 4
                    }
                }
            }
        }
    }

// system0 configuration

A:leaf1# info interface system0
    interface system0 {
        subinterface 0 {
            admin-state enable
            ipv4 {
                admin-state enable
                address 192.0.2.4/32 {
                }
            }
        }
    }
```

Example 1. Configuration of point-to-point interfaces and system0 interface

4.2 Default network-instance

The point-to-point interfaces between the leafs and the spines are mapped to the default network-instance in SR Linux. Additionally, the system0 subinterface used as the VXLAN tunnel endpoint (VTEP) source address is also mapped to the default network-instance. Since the NVD uses IPv4 addressing for the system0 interface (which is used for VXLAN tunnels), the IPv6 forwarding check must be disabled as IPv4 packets received on an IPv6-only interface are dropped by default. This is achieved by setting the *ip-forwarding receive-ipv4-check* configuration option to *false*.

```
// configuration of default network-instance which forms the underlay or IP fabric

A:leaf1# info network-instance default
    network-instance default {
        type default
        admin-state enable
        description "fabric: dc1 role: leaf"
        router-id 192.0.2.4
        ip-forwarding {
            receive-ipv4-check false
        }
        interface ethernet-1/29.0 {
        }
        interface ethernet-1/30.0 {
        }
        interface system0.0 {
        }

*snip*
```

Example 2 Configuration snippet of the default network-instance

4.3 BGP for underlay and overlay routes

The NVD uses an eBGP design (with all spines assigned the same ASN and each leaf assigned a unique ASN), utilizing MP-BGP functionality with multiple address families advertised as capabilities over a single BGP session. The eBGP sessions are configured for dynamic discovery, leveraging the IPv6 link-local underlay design and IPv6 ND capabilities. In addition, the following also apply to BGP:

- EDA generated routing policies for advertising underlay IPv4 routes and overlay EVPN routes
- Configuration option to allow IPv4 routes to be advertised with IPv6 next hops
- Configuration option to accept receipt of IPv4 routes with IPv6 next hops
- Multipath enabled for IPv4 unicast and L2VPN EVPN AFIs/SAFIs
- Configuration option to enable rapid withdrawal of BGP routes and rapid update of EVPN routes
- On the spines, *inter-as-vpn* configuration option must be set to *true* for an eBGP design since the spines are not configured with any VXLAN constructs; thus, drop all inbound BGP EVPN updates due to no corresponding route target.

```
// BGP configuration

A:leaf1# info network-instance default protocols bgp
    network-instance default {
        protocols {
            bgp {
                admin-state enable
                autonomous-system 65411
                router-id 192.0.2.4
                dynamic-neighbors {
                    interface ethernet-1/29.0 {
                        peer-group bgpgroup-ebgp-dc1
                        allowed-peer-as [
                            65500
                        ]
                    }
                    interface ethernet-1/30.0 {
                        peer-group bgpgroup-ebgp-dc1
                        allowed-peer-as [
                            65500
                        ]
                    }
                }
                ebgp-default-policy {
                    import-reject-all true
                    export-reject-all true
                }
                afi-safi evpn {
                    admin-state enable
                    multipath {
                        allow-multiple-as true
                        maximum-paths 64
                    }
                    evpn {
                        inter-as-vpn true
                        rapid-update true
                    }
```

```
                }
                afi-safi ipv4-unicast {
                    admin-state enable
                    multipath {
                        allow-multiple-as true
                        maximum-paths 2
                    }
                    ipv4-unicast {
                        advertise-ipv6-next-hops true
                        receive-ipv6-next-hops true
                    }
                    evpn {
                        rapid-update true
                    }
                }
                afi-safi ipv6-unicast {
                    admin-state enable
                    multipath {
                        allow-multiple-as true
                        maximum-paths 2
                    }
                    evpn {
                        rapid-update true
                    }
                }
                preference {
                    ebgp 170
                    ibgp 170
                }
                route-advertisement {
                    rapid-withdrawal true
                    wait-for-fib-install false
                }
                group bgpgroup-ebgp-dc1 {
                    admin-state enable
                    export-policy [
                        ebgp-isl-export-policy-dc1
                    ]
                    import-policy [
                        ebgp-isl-import-policy-dc1
                    ]
                    failure-detection {
                        enable-bfd true
                        fast-failover true
                    }
                    afi-safi evpn {
                        admin-state enable
                    }
                    afi-safi ipv4-unicast {
                        admin-state enable
                        ipv4-unicast {
                            advertise-ipv6-next-hops true
                            receive-ipv6-next-hops true
                        }
                    }
                    afi-safi ipv6-unicast {
                        admin-state enable
                    }
                }
            }
        }
    }
```

Example 3. BGP configuration from leaf1 for underlay and overlay routes

4.4 Maximum Transmission Unit (MTU)

System-wide MTUs are configured globally to accommodate larger-sized packets (considering 50 Bytes overhead is added as part of the overall VXLAN encapsulation). On Nokia 7220 IXR-D3Ls, D4s, and D5s (which comprise the leafs in the NVD topology), the following MTUs are configured:

```
// system-wide default MTU configuration on leafs

A:leaf1# info system mtu
    system {
        mtu {
            default-port-mtu 9412
            default-l2-mtu 9412
            default-ip-mtu 9200
        }
    }
```

Example 4. Configuration of system-wide default MTUs on a Nokia 7220 IXR-D4

On the spines, which are Nokia 7220 IXR-H4s, the following MTUs are configured:

```
// system-wide default MTU configuration on spines

A:spine1# info system mtu
    system {
        mtu {
            default-port-mtu 9412
            default-ip-mtu 9200
        }
    }
```

Example 5. Configuration of system-wide default MTUs on a Nokia 7220 IXR-H4

With a maximum configured IP MTU of 9200, the maximum sized payload within an IP packet that can be sent from the server is 9168.

4.5 Bidirectional Forwarding Detection (BFD)

BFD is enabled on the links between the leafs and the spines. BGP is enabled for fast-failover using BFD (with a failure detection time of 750ms).

```
// BGP configuration on point-to-point subinterface

A:leaf1# info bfd
    bfd {
        subinterface ethernet-1/29.0 {
            admin-state enable
            desired-minimum-transmit-interval 250000
            required-minimum-receive 250000
            detection-multiplier 3
            minimum-echo-receive-interval 250000
        }
        subinterface ethernet-1/30.0 {
            admin-state enable
```

```
            desired-minimum-transmit-interval 250000
            required-minimum-receive 250000
            detection-multiplier 3
            minimum-echo-receive-interval 250000
        }
    }

// BGP enabled for fast-failover with BFD

A:leaf1# info network-instance default protocols bgp group bgpgroup-ebgp-dc1 failure-detection
    network-instance default {
        protocols {
            bgp {
                group bgpgroup-ebgp-dc1 {
                    failure-detection {
                        enable-bfd true
                        fast-failover true
                    }
                }
            }
        }
    }
```

Example 6. Configuration of BFD and BGP enabled for fast-failover

4.6 Link Layer Discovery Protocol (LLDP)

LLDP is used to discover neighboring devices.

```
// LLDP enabled for neighbor discovery

A:leaf1# info system lldp
    system {
        lldp {
            interface ethernet-1/29 {
                admin-state enable
            }
            interface ethernet-1/30 {
                admin-state enable
            }
        }
    }
```

Example 7. LLDP configuration

4.7 Layer 2 server-facing interfaces

Untagged and tagged Layer 2 server-facing interfaces are tested as part of this NVD. A sample configuration is provided below with multiple subinterfaces configured on an interface, one tagged and another untagged. Use of subinterfaces in this fashion allows for a logical separation of the expected traffic on the physical interface. These subinterfaces are then mapped to their respective MAC-VRFs (shown later in this document).

```
// Layer 2 untagged and tagged subinterfaces

A:leaf1# info interface ethernet-1/1
    interface ethernet-1/1 {
```

```
        admin-state enable
        vlan-tagging true
        subinterface 40 {
            type bridged
            admin-state enable
            vlan {
                encap {
                    single-tagged {
                        vlan-id 40
                    }
                }
            }
        }
        subinterface 4096 {
            type bridged
            admin-state enable
            vlan {
                encap {
                    untagged {
                    }
                }
            }
        }
    }
```

Example 8. Configuration of Layer 2 untagged and tagger server-facing interfaces

4.8 All-active ES-based link aggregation group (LAG)

A 4-way all-active Ethernet Segment is tested as part of this NVD. Ethernet Segments are supported natively within EVPN as a standard, allowing more than just two VTEPs for multihoming. The configuration includes the following:

- Mapping the physical interface (meant to be part of a LAG in the case of this NVD) to a LAG interface
- Configuring the LAG interface with required LACP parameters
- Configuring an Ethernet Segment (and all required parameters) and mapping it to the respective LAG interface
- The Designated Forwarder election activation timer is set to 0 (the default timer is 3 seconds). This timer controls the delay of transition from non-DF to DF.

```
// physical interface mapped to LAG interface

A:leaf1# info interface ethernet-1/3
    interface ethernet-1/3 {
        description leaf1-leaf2-leaf3-leaf4-lag1
        admin-state enable
        ethernet {
            aggregate-id lag1
            lacp-port-priority 32768
            reload-delay 100
        }
    }

// LAG interface configured with untagged/tagged subinterfaces and LACP parameters

A:leaf1# info interface lag1
    interface lag1 {
```

```
        description leaf1-leaf2-leaf3-leaf4-lag1
        admin-state enable
        vlan-tagging true
        subinterface 50 {
            type bridged
            admin-state enable
            vlan {
                encap {
                    single-tagged {
                        vlan-id 50
                    }
                }
            }
        }
        subinterface 4096 {
            type bridged
            admin-state enable
            vlan {
                encap {
                    untagged {
                    }
                }
            }
        }
        lag {
            lag-type lacp
            min-links 1
            lacp-fallback-mode static
            lacp-fallback-timeout 60
            lacp {
                interval FAST
                lacp-mode ACTIVE
                admin-key 1
                system-id-mac 00:00:11:22:33:44
                system-priority 32768
            }
        }
    }

// Ethernet Segment configuration for all-active multihoming mode

A:leaf1# info system network-instance protocols
    system {
        network-instance {
            protocols {
                evpn {
                    ethernet-segments {
                        bgp-instance 1 {
                            ethernet-segment leaf1-leaf2-leaf3-leaf4-lag1 {
                                admin-state enable
                                esi 00:00:00:11:22:33:44:00:00:00
                                multi-homing-mode all-active
                                interface lag1 {
                                }
                                df-election {
                                    timers {
                                        activation-timer 0
                                    }
                                    algorithm {
                                        type default
                                    }
                                }
                            }
                        }
                    }
```

```
                }
                bgp-vpn {
                    bgp-instance 1 {
                    }
                }
            }
        }
    }
```

Example 9. Configuration of all-active ES-based LAG

4.9 Single-active ES-based link aggregation group (LAG)

Single-active Ethernet Segments (with port-active functionality, described in IETF draft https://www.ietf.org/archive/id/draft-ietf-bess-evpn-mh-pa-10.html, as of March 2025) are tested as part of this NVD. This is useful if the server requires only a single link to be active for proper functioning, while still offering server-uplink redundancy if the active link goes down.

During steady state, only the active link forwards traffic in such a design. This is enforced by sending LACP *out of sync* PDUs over the member interface of the LAG on the non-DF VTEP. SR Linux also supports powering off the port (by shutting off the laser) in cases where the server does not support LACP.

If the active link goes down, the directly connected VTEP (which was the DF for that Ethernet Segment) withdraws its EVPN Type-4 route, triggering the peer VTEP to move from non-DF to DF. The new DF now starts sending LACP *in sync* PDUs, causing the connected server interface to be bundled back into the LAG, which can now actively forward traffic.

Like all-active, the configuration of single-active ES-based LAG includes the following:

- Mapping physical interface to a LAG interface
- Configuring the LAG interface with required LACP parameters
- Configuring subinterfaces within the LAG interface to accept tagged or untagged Layer 2 packets as required
- Configuring an Ethernet Segment (and all required parameters) and mapping it to the respective LAG interface (notably, the *multi-homing-mode* configuration option is set to single-active)
- The Ethernet Segment is configured on the active node with a higher preference, with a preference-based algorithm being used for Designated Forwarder (DF) election.
- The *interface-standby-signaling-on-non-df* configuration is set under the *df-election* hierarchy. This sends a LACP out-of-sync on non-DF nodes, keeping the server links connected to the non-DF nodes in a *down* state.
- The Designated Forwarder election activation timer is set to 0 (the default timer is 3 seconds). This timer controls the delay of transition from non-DF to DF.

```
// physical interface mapped to LAG interface
```

```
A:leaf5# info interface ethernet-1/2
    interface ethernet-1/2 {
        description leaf5-leaf6-lag1
        admin-state enable
        ethernet {
            aggregate-id lag1
            lacp-port-priority 32768
            reload-delay 100
        }
    }

// configuration of LAG interface

A:leaf5# info interface lag1
    interface lag1 {
        description leaf5-leaf6-lag1
        admin-state enable
        vlan-tagging true
        ethernet {
            standby-signaling lacp
        }
        subinterface 60 {
            type bridged
            admin-state enable
            vlan {
                encap {
                    single-tagged {
                        vlan-id 60
                    }
                }
            }
        }
        subinterface 4096 {
            type bridged
            admin-state enable
            vlan {
                encap {
                    untagged {
                    }
                }
            }
        }
        lag {
            lag-type lacp
            min-links 1
            lacp-fallback-mode static
            lacp-fallback-timeout 60
            lacp {
                interval FAST
                lacp-mode ACTIVE
                admin-key 2
                system-id-mac 00:00:00:00:55:66
                system-priority 32768
            }
        }
    }

// Ethernet Segment configuration on active VTEP

A:leaf5# info system network-instance protocols
    system {
        network-instance {
            protocols {
                evpn {
                    ethernet-segments {
```

```
                        bgp-instance 1 {
                            ethernet-segment leaf5-leaf6-lag1 {
                                admin-state enable
                                esi 00:00:00:00:00:55:66:00:00:00
                                multi-homing-mode single-active
                                interface lag1 {
                                }
                                df-election {
                                    timers {
                                        activation-timer 0
                                    }
                                    interface-standby-signaling-on-non-df {
                                    }
                                    algorithm {
                                        type preference
                                        preference-alg {
                                            preference-value 800
                                            capabilities {
                                                non-revertive true
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
                bgp-vpn {
                    bgp-instance 1 {
                    }
                }
            }
        }
    }

// Ethernet Segment configuration on standby VTEP

A:leaf6# info system network-instance protocols
    system {
        network-instance {
            protocols {
                evpn {
                    ethernet-segments {
                        bgp-instance 1 {
                            ethernet-segment leaf5-leaf6-lag1 {
                                admin-state enable
                                esi 00:00:00:00:00:55:66:00:00:00
                                multi-homing-mode single-active
                                interface lag1 {
                                }
                                df-election {
                                    timers {
                                        activation-timer 0
                                    }
                                    interface-standby-signaling-on-non-df {
                                    }
                                    algorithm {
                                        type preference
                                        preference-alg {
                                            preference-value 500
                                            capabilities {
                                                non-revertive true
                                            }
                                        }
                                    }
```

```
                                }
                            }
                        }
                    }
                }
                bgp-vpn {
                    bgp-instance 1 {
                    }
                }
            }
        }
    }

// LAG interface state on active VTEP

A:leaf5# show lag lag1 lacp-state  | as yaml
---
LacpHeader:
  - Lag Id: lag1
    LacpBrief:
      Interval: FAST
      Mode: ACTIVE
      System Id: '00:00:00:00:55:66'
      System Priority: 32768
    LacpState:
      - Members: ethernet-1/2
        Oper state: up
        Activity: ACTIVE
        Timeout: SHORT
        State: IN_SYNC/True/True/True
        System Id: '00:00:00:00:55:66'
        Oper key: 2
        Partner Id: '00:00:00:00:99:99'
        Partner Key: 32769
        Port No: 1
        Partner Port No: 5

// LAG interface state on standby VTEP

A:leaf6# show lag lag1 lacp-state | as yaml
---
LacpHeader:
  - Lag Id: lag1
    LacpBrief:
      Interval: FAST
      Mode: ACTIVE
      System Id: '00:00:00:00:55:66'
      System Priority: 32768
    LacpState:
      - Members: ethernet-1/1
        Oper state: down(lacp-down)
        Activity: ACTIVE
        Timeout: SHORT
        State: OUT_SYNC/True/False/False
        System Id: '00:00:00:00:55:66'
        Oper key: 2
        Partner Id: '00:00:00:00:99:99'
        Partner Key: 32769
        Port No: 1
        Partner Port No: 6
```

Example 10. Configuration of single-active ES-based LAG

4.10 Active/backup with no Link Aggregation Group (LAG)

Active/backup functionality and convergence is tested by using a server with two NICs (one to each leaf/VTEP) configured for Linux bond mode 1 (active/backup). The NICs function without being aggregated into a Link Aggregation Group (LAG), with one NIC being the active link passing traffic from the server. From the perspective of the leafs (VTEPs), the convergence is purely a function of MAC mobility since only the link towards the active NIC of the server will be receiving traffic at any given time.

```
// sample configuration from an Ubuntu 22.04 server for active/backup bond mode
// to make this persistent, configure using netplan instead

sudo ip link add bond0 type bond mode active-backup primary ens5f0np0
sudo ip link set bond0 type bond miimon 100
sudo ip link set ens5f0np0 down
sudo ip link set ens5f1np1 down
sudo ip link set ens5f0np0 master bond0
sudo ip link set ens5f1np1 master bond0
sudo ip addr add 172.16.10.10/24 dev bond0
sudo ip link set ens5f0np0 up
sudo ip link set ens5f1np1 up
sudo ip link set bond0 up
sudo ip route add 0.0.0.0/24 via 172.16.10.254

// interface configuration from leaf5 and leaf6 (VTEPs to which server is attached)

A:leaf5# info interface ethernet-1/5
    interface ethernet-1/5 {
        admin-state disable
        vlan-tagging true
        subinterface 4096 {
            type bridged
            admin-state enable
            vlan {
                encap {
                    untagged {
                    }
                }
            }
        }
    }

A:leaf6# info interface ethernet-1/5
    interface ethernet-1/5 {
        admin-state enable
        vlan-tagging true
        subinterface 4096 {
            type bridged
            admin-state enable
            vlan {
                encap {
                    untagged {
                    }
                }
            }
        }
    }
```

Example 11. Configuration of active/backup (Linux bond mode 1) server connectivity with no LAG

4.11 Layer 3 server-facing interfaces

Layer 3 server-facing interfaces are commonly deployed for cloud-native environments, enabling an end-to-end routing design. While the NVD is tested using static routes configured on a leaf to container subnets behind a Layer 3 attached server (these static routes are exported into the fabric as EVPN Type-5 routes and distributed to other VTEPs using BGP EVPN), you can also choose to run BGP between the leaf and the server for dynamic exchange of routes.

The Layer 3 interface is mapped to its respective IP VRF with static routes for subnets behind the container defined within this IP VRF.

```
A:d4-leaf4# info interface ethernet-1/3
    interface ethernet-1/3 {
        admin-state enable
        subinterface 4097 {
            type routed
            description d4-leaf4-l3-1
            admin-state enable
            ip-mtu 9200
            ipv4 {
                admin-state enable
                address 172.16.100.0/31 {
                    primary
                }
                arp {
                    timeout 250
                }
            }
        }
    }

A:d4-leaf4# info network-instance vrf1
    network-instance vrf1 {
        type ip-vrf
        admin-state enable
        description vrf1
        interface ethernet-1/3.4097 {
        }

*snip*

A:d4-leaf4# info network-instance vrf1 static-routes
    network-instance vrf1 {
        static-routes {
            route 172.16.92.0/22 {
                admin-state enable
                next-hop-group static-d4-leaf4
            }
        }
    }

A:d4-leaf4# info network-instance vrf1 next-hop-groups group static-d4-leaf4
    network-instance vrf1 {
        next-hop-groups {
            group static-d4-leaf4 {
                admin-state enable
                nexthop 0 {
                    ip-address 172.16.100.1
                    admin-state enable
```

```
                    resolve false
                }
            }
        }
    }
```

Example 12. Configuration of Layer 3 server-facing interface

4.12 IRB interfaces

IRB interfaces are configured in an anycast, distributed gateway model with each leaf using the same IP address and MAC address (auto derived; in this case, using the VRRP MAC address range, as part RFC 9135). The IRB subinterfaces are also enabled with L3 proxy-ARP, with BGP EVPN configured to advertise entries in the ARP table as EVPN Type-2 routes. The ARP timeout, for each IRB subinterface, is configured to be lower than the default MAC addressing aging timer (300 seconds).

These IRB interfaces are the default gateways for the servers.

```
// IRB subinterface

A:leaf1# info interface irb0 subinterface 0
    interface irb0 {
        subinterface 0 {
            ip-mtu 9200
            ipv4 {
                admin-state enable
                address 172.16.30.254/24 {
                    anycast-gw true
                    primary
                }
                arp {
                    timeout 250
                    learn-unsolicited true
                    proxy-arp true
                    evpn {
                        advertise dynamic {
                        }
                    }
                }
            }
            anycast-gw {
                virtual-router-id 1
            }
        }
    }
```

Example 13. Configuration of IRB interfaces on leaf nodes

4.13 VXLAN tunnels

VXLAN tunnels are created as tunnel interfaces on SR Linux, where each subinterface is mapped to a bridged VNI (L2VNI) or routed VNI (L3VNI). A sample configuration is provided below, demonstrating a bridged tunnel and a routed tunnel. These bridged and routed tunnel interfaces are associated to their corresponding network instances – bridged

VXLAN tunnel interfaces for MAC VRFs (Layer 2) and routed VXLAN interfaces to IP VRFs (Layer 3).

```
// Bridged and routed VXLAN tunnels

A:leaf1# info tunnel-interface vxlan0 vxlan-interface {505,506}
    tunnel-interface vxlan0 {
        vxlan-interface 505 {
            type bridged
            ingress {
                vni 10060
            }
            egress {
                source-ip use-system-ipv4-address
            }
        }
        vxlan-interface 506 {
            type routed
            ingress {
                vni 10501
            }
            egress {
                source-ip use-system-ipv4-address
            }
        }
    }
```

Example 14. Configuration of bridged and routed VXLAN tunnel interfaces

4.14 MAC VRFs

MAC VRFs are created for Layer 2 isolation. These MAC VRFs are mapped to a bridged VXLAN tunnel-interface and the bridge domains' corresponding IRB interface, along with the required Layer 2 server-facing subinterfaces (these can be subinterfaces of a physical or LAG interface). Every MAC VRF is associated with a corresponding import and export Route Target which facilitates the import and export of BGP EVPN routes for this MAC VRF. In addition to this, MAC VRFs are configured with the following options:

- For overlay ECMP, the *ecmp* configuration option is used and set to a value of 8.
- The configuration option *advertise-arp-nd-only-with-mac-table-entry* is set to *true*. This is necessary for multihoming segments, without which misleading MAC mobility events might occur.
- Each MAC VRF is enabled with the default duplicate MAC detection timers.

```
// VLAN-based MAC VRF configuration

A:leaf1# info network-instance macvrf-v10
    network-instance macvrf-v10 {
        type mac-vrf
        admin-state enable
        description macvrf-v10
        interface ethernet-1/1.4096 {
        }
        interface irb0.4 {
        }
        interface lag1.4096 {
```

```
        }
        vxlan-interface vxlan0.500 {
        }
        protocols {
            bgp-evpn {
                bgp-instance 1 {
                    vxlan-interface vxlan0.500
                    evi 10
                    ecmp 8
                    routes {
                        bridge-table {
                            mac-ip {
                                advertise-arp-nd-only-with-mac-table-entry true
                            }
                        }
                    }
                }
            }
            bgp-vpn {
                bgp-instance 1 {
                    route-target {
                        export-rt target:1:10
                        import-rt target:1:10
                    }
                }
            }
        }
        bridge-table {
            mac-learning {
                admin-state enable
                aging {
                    admin-state enable
                    age-time 300
                }
            }
            mac-duplication {
                admin-state enable
                monitoring-window 3
                num-moves 5
                hold-down-time 9
                action stop-learning
            }
        }
    }
```

Example 15. Configuration of MAC VRFs

4.15 IP VRFs

IP VRFs are used for Layer 3 isolation and to enable the use of a common, physical infrastructure for multiple, logically isolated tenants/services. The respective IRB subinterfaces are mapped to their corresponding IP VRFs along with a routed VXLAN tunnel-interface (which is the L3VNI for that IP VRF).

Like MAC VRFs, each IP VRF is associated with an export and import Route Target.

```
// IP VRF configuration

A:leaf1# info network-instance vrf1
    network-instance vrf1 {
        type ip-vrf
```

```
        admin-state enable
        description vrf1
        interface irb0.0 {
        }
        interface irb0.2 {
        }
        interface irb0.4 {
        }
        interface irb0.5 {
        }
        vxlan-interface vxlan0.507 {
        }
        protocols {
            bgp-evpn {
                bgp-instance 1 {
                    vxlan-interface vxlan0.507
                    evi 500
                    ecmp 8
                    routes {
                        route-table {
                            mac-ip {
                                advertise-gateway-mac true
                            }
                        }
                    }
                }
            }
            bgp-vpn {
                bgp-instance 1 {
                    route-target {
                        export-rt target:1:500
                        import-rt target:1:500
                    }
                }
            }
        }
    }
```

Example 16. Configuration of IP VRFs

4.16 Node isolation

Node isolation is used in situations where a VTEP loses its core-facing uplinks while retaining server-facing downlinks. For dual-homed servers, this can create a situation where an alternate path is available but may not be used since traffic is hashed to an impacted leaf node.

In SR Linux v24.10.2, node isolation is implemented using the combination of a user-defined upython script (provided in its entirety below) and event-handlers that leverage operational groups. The idea is to monitor the number and state of BGP EVPN peers on a VTEP that has downstream LAG interfaces mapped to an Ethernet Segment. If all BGP EVPN peers are down (i.e. there are no BGP EVPN peers in an *Established* state), then the tracked downstream interfaces are brought down as well.

```
// SRL event-handler that tracks BGP EVPN state and takes appropriate action when triggered on
specified down-links

A:leaf1# info system event-handler
```

```
    system {
        event-handler {
            instance overlay-bgp {
                admin-state enable
                upython-script node-isolation.py
                paths [
                    "network-instance default protocols bgp neighbor * session-state"
                ]
                options {
                    object down-links {
                        values [
                            ethernet-1/3
                        ]
                    }
                    object hold-down-time {
                        value 20000
                    }
                    object required-bgp-sessions-established {
                        value 1
                    }
                }
            }
        }
    }

// Node isolation upython script stored in the path /etc/opt/srlinux/eventmgr where all user-
defined scripts are expected to be stored

admin@leaf1:/etc/opt/srlinux/eventmgr$ pwd
/etc/opt/srlinux/eventmgr

admin@leaf1:/etc/opt/srlinux/eventmgr$ cat node-isolation.py
import sys
import json

# count_bgp_sessions_established returns the number of monitored BGP sessions that are
established  {established=up}
def count_bgp_sessions_established(paths):
    up_cnt = 0
    for path in paths:
        if path.get("value") == "established":
            up_cnt = up_cnt + 1
    return up_cnt

# required_bgp_sessions_established returns the value of the `required-bgp-sessions-established`
option
def required_bgp_sessions_established(options):
    return int(options.get("required-bgp-sessions-established", 1))

# hold down timer after recovery
def hold_time(options):
    return int(options.get('hold-down-time', '0'))

def bool_to_oper_state(val):
    return ('down','up')[bool(val)]

# main entry function for event handler
def event_handler_main(in_json_str):
    # parse input json string passed by event handler
    in_json = json.loads(in_json_str)
    paths = in_json["paths"]
    options = in_json["options"]
    persist = in_json.get('persistent-data', {})

    num_up_bgp_sessions = count_bgp_sessions_established(paths)
```

```
    downlink_should_be_up = required_bgp_sessions_established(options) <= num_up_bgp_sessions
    needs_hold_down = False

    # down->up transition will be held for optional hold-time
    if (hold_time(options) > 0) and downlink_should_be_up:
        needs_hold_down = persist.get("last-state", "up") == "down"

    if options.get("debug") == "true":
        print(
            f"hold down time = {hold_time(options)}ms\n\
num of required bgp_sessions = {required_bgp_sessions_established(options)}\n\
detected num of bgp_sessions = {num_up_bgp_sessions}\n\
downlinks new state = {bool_to_oper_state(downlink_should_be_up)}\n\
needs_hold_down = {str(needs_hold_down)}"
        )

    response_actions = []

    oper_state_str = bool_to_oper_state(not needs_hold_down and downlink_should_be_up)
    for downlink in options.get('down-links'):
        response_actions.append({'set-ephemeral-path' : {'path':'interface {0} oper-
state'.format(downlink),'value':oper_state_str}})

    if needs_hold_down:
        response_actions.append({'reinvoke-with-delay' : hold_time(options)})
    response_persistent_data = {'last-state':bool_to_oper_state(downlink_should_be_up)}

    response = {'actions':response_actions,'persistent-data':response_persistent_data}
    return json.dumps(response)
```

Example 17. Node isolation upython script and SRL event-handler configuration

5 Test summary

5.1 Feature matrix

Feature	SRL 24.10.2	EDA 24.12.1	
		Validation State	**EDA Configlets**
IPv6 link-local addressing with IPv6 ND for fabric underlay	Validated	Validated	No
Advertise and receive BGP IPv4 NLRIs with IPv6 next hops (RFC 8950)	Validated	Validated	No
MP-BGP style eBGP peering for underlay and overlay routes	Validated	Validated	Yes
2-byte BGP ASN support	Validated	Validated	No
Routing policies for underlay and overlay	Validated	Validated	No

Sub-second BFD convergence (750ms)	Validated	Validated	No
LLDP	Validated	Validated	No
Layer 2 untagged server-facing interfaces	Validated	Validated	No
Layer 2 tagged server-facing interfaces	Validated	Validated	No
Layer 3 server-facing interfaces	Validated	Validated	No
Jumbo MTU	Validated	Validated	No
Anycast GWs	Validated	Validated	No
ESI-based LAG in all-active mode	Validated	Validated	Yes
ESI-based LAG in single-active mode	Validated	Validated	Yes
Active/backup server link connectivity with no LAG	Validated	Validated	No
ECMP for underlay and overlay	Validated	Validated	No
Asymmetric IRB routing	Validated	Validated	No
Symmetric IRB routing with Type-5	Validated	Validated	No
VLAN-based MAC VRFs	Validated	Validated	Yes
IP VRFs	Validated	Validated	No
Node isolation	Validated	Validated	Yes
gNMI-based telemetry	Validated	Validated	No

Table 2. Feature matrix

Test	Description	Approximate convergence time
BGP reset	Traffic flows are enabled, BGP peers are reset, and traffic convergence is measured.	39.5ms
Active/Backup with no Link Aggregation Group (LAG)	Dual links on a server are configured with Linux bond mode 1 (active/backup). This implies that the NICs function without being aggregated into a LAG, and simply in an active/backup fashion. The active link is shut down and the traffic convergence time is measured	50ms

	(convergence is purely a function of EVPN MAC mobility)	
4-way ES-based LAG in all-active mode (ingress VTEP local-bias forwarding)	4-way all-active ES-based LAG is configured and E-W traffic flows are enabled. Traffic exits the local interface (part of ES) of ingress VTEP (as part of local-bias forwarding). During this state, the local exit interface is shut down and the convergence time is measured for traffic to move to a remote VTEP over the fabric.	33.6ms
4-way ES-based LAG in all-active mode (ES interface down convergence)	4-way all-active ES-based LAG is configured and E-W traffic flows are enabled. Traffic enters ingress leaf on a L2/L3 single-homed interface and exits a remote VTEP on an ethernet segment. During this state, the exit interface is shut down on egress leaf and the convergence time is measured.	5.9ms
2-way ESI-based LAG in single-active mode	2-way single-active ESI-LAG is configured with E-W traffic flowing through the active VTEP. During this steady state, the active ES member interface is shut down and the convergence time is measured	100-200ms
Leaf reboot with E-W traffic	Inter-VLAN and intra-VLAN traffic is flowing from the source interface on an ingress leaf (VTEP) to a destination interface on an egress leaf (VTEP). During this steady state, the ingress leaf (VTEP) is rebooted and the convergence time is measured.	165 seconds
Spine reboot with dual spines connected and active	E-W traffic is flowing through the fabric with some flows hashed to spine1 and others to spine2. During this steady state, spine1 is rebooted and convergence time is measured for all traffic that was flowing through spine1.	26.06ms
Spine reboot with single spine connected and active	E-W traffic is flowing through the fabric via the only spine that is connected and active. During this state, the spine is rebooted and the convergence time is measured.	215 seconds
MAC mobility	Flap the ports so the MAC address moves from local to remote and vice versa, and observe the convergence time.	24.6ms

Table 3. Traffic convergence metrics

6 EDA integration

6.1 EDA architecture

Nokia’s Event Driven Automation (EDA) platform is a cloud-native platform deployed on top of Kubernetes, leveraging the Kubernetes-provided declarative API, tooling, and the ecosystem around it. EDA can be deployed as a single or multimode cluster.

The various components of the EDA/K8s tech stack are shown below, instantiated as Kubernetes pods.

```
:~$ kubectl get pods -A
NAMESPACE            NAME                                            READY   STATUS    RESTARTS
cert-manager         cert-manager-767c6596b-4xfnj                    1/1     Running   1
cert-manager         cert-manager-cainjector-78f86bf99f-d8pj4        1/1     Running   1
cert-manager         cert-manager-webhook-5b8cb89ffc-pvlt7           1/1     Running   1
eda-system           cert-manager-csi-driver-6t9vj                   3/3     Running   3
eda-system           eda-api-9985cb78-cn689                          1/1     Running   1
eda-system           eda-appstore-5db7b8c746-7hwzn                   1/1     Running   1
eda-system           eda-asvr-68bc7c86b6-7cz8r                       1/1     Running   1
eda-system           eda-bsvr-6bf77b64c-6mk85                        1/1     Running   1
eda-system           eda-ce-5c8d5b5969-h5qgr                         1/1     Running   1
eda-system           eda-fe-547cb647df-tm2c6                         1/1     Running   1
eda-system           eda-fluentbit-txn7r                             1/1     Running   1
eda-system           eda-fluentd-54cf4bd5d7-4kn4f                    1/1     Running   1
eda-system           eda-git-754df68df5-lqqfd                        1/1     Running   1
eda-system           eda-git-replica-784dbdbfc8-j8fjb                1/1     Running   1
eda-system           eda-keycloak-6b5655dbcc-h2g4c                   1/1     Running   0
eda-system           eda-metrics-server-7c495c6bf-575dj              1/1     Running   1
eda-system           eda-npp-eda-d3-leaf5                            1/1     Running   0
eda-system           eda-npp-eda-d3-leaf6                            1/1     Running   0
eda-system           eda-npp-eda-d4-leaf3                            1/1     Running   0
eda-system           eda-npp-eda-d4-leaf4                            1/1     Running   0
eda-system           eda-npp-eda-d5-leaf1                            1/1     Running   0
eda-system           eda-npp-eda-d5-leaf2                            1/1     Running   0
eda-system           eda-npp-eda-spine1                              1/1     Running   0
eda-system           eda-npp-eda-spine2                              1/1     Running   0
eda-system           eda-postgres-5c8dc78fbf-vmjz4                   1/1     Running   0
eda-system           eda-sa-576c98865f-44lwb                         1/1     Running   1
eda-system           eda-sc-84546648c5-5ncgh                         1/1     Running   1
eda-system           eda-se-1                                        1/1     Running   1
eda-system           eda-toolbox-84c95bd8c6-ptbt4                    1/1     Running   1
eda-system           trust-manager-567f4b65fb-4dllq                  1/1     Running   1
kube-system          coredns-6f6b679f8f-9m8rx                        1/1     Running   1
kube-system          coredns-6f6b679f8f-gvjf8                        1/1     Running   1
kube-system          etcd-eda-demo-control-plane                     1/1     Running   1
kube-system          kindnet-d5sbh                                   1/1     Running   1
kube-system          kube-apiserver-eda-demo-control-plane           1/1     Running   1
kube-system          kube-controller-manager-eda-demo-control-plane  1/1     Running   1
kube-system          kube-proxy-s7rv6                                1/1     Running   1
kube-system          kube-scheduler-eda-demo-control-plane           1/1     Running   1
local-path-storage   local-path-provisioner-57c5987fd4-m5p4p         1/1     Running   1
metallb-system       controller-fbf54885d-8j5qf                      1/1     Running   1
metallb-system       speaker-g74r6                                   1/1     Running   2
```

Example 18. EDA namespaces and pods

Some of the more commonly used pods and their functionalities are listed below:

- **eda-asvr** - the artifact server stores common artifacts used in EDA functionality. Examples include SRLinux image, SRL MD5 hash, yang path.zip, and so forth. The availability of an artifact can be verified with “kubectl get artifacts -A”.
- **eda-bsvr** – the bootstrap server is responsible for all onboarding of nodes (virtual or hardware). This involves gNMI discovery, gNMI management, and instantiation of NPP pods for node lifecycle management.

- **eda-ce** – the configuration engine keeps track of all the dependencies amongst the application resources and runs the application intents when needed.
- **eda-npp** – the eda-npp pod is responsible for schema validation of the generated configuration. Additionally, it is responsible for all communications to the devices for both setting configuration and retrieving state.
- **eda-api** – the eda-api pod is the REST API server which is accessible to end users and is consumed by the GUI.
- **eda-cx** – sandbox controller that spins up simulated nodes for building digital twins of the fabric (the example above has the mode set to physical hardware only, hence the EDA CX functionality has been disabled)
- **eda-toolbox** – provides tools such as *edactl* for insight into EDA transactions and EDA topology generator that can generate a topology from a YAML file

Figure 12 demonstrates the high-level workflow required to build the prescriptive 3-stage EVPN VXLAN NVD. The resources shown can be created using either the EDA UI or natively using Kubernetes manifest files.

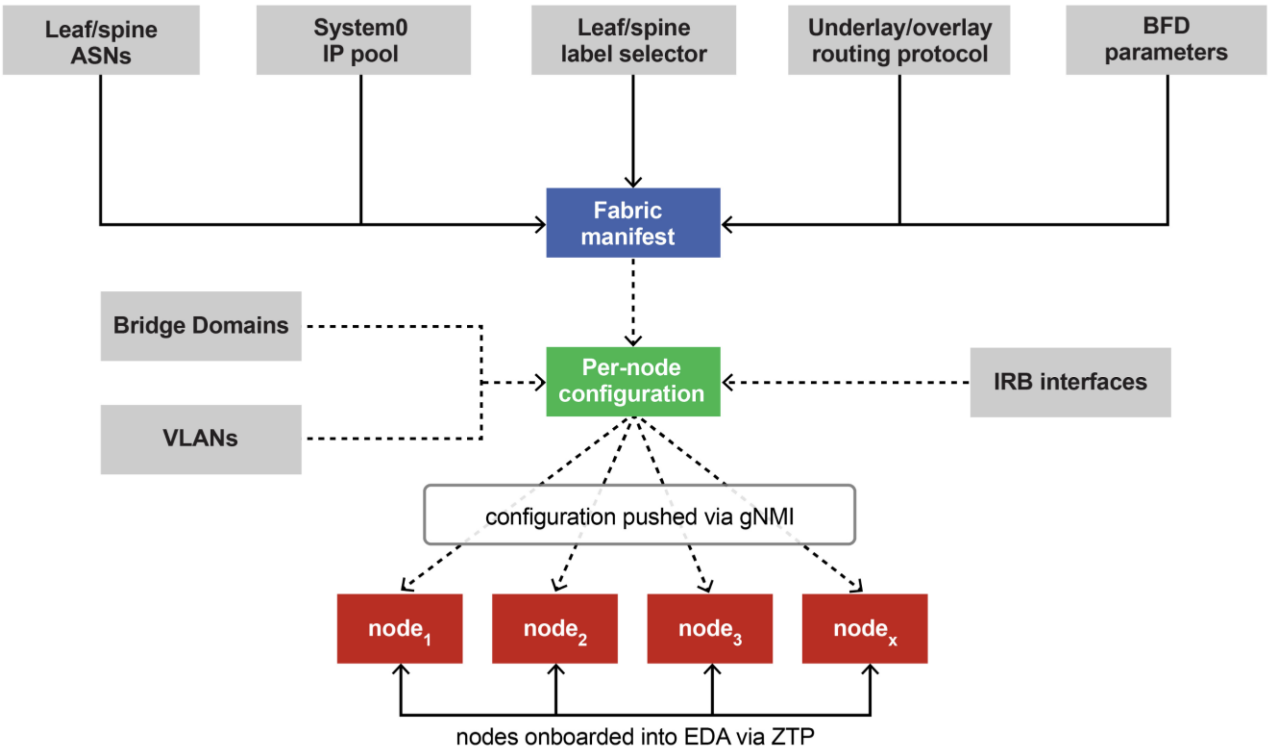

Figure 12. EDA workflow

Once this workflow is completed with all nodes onboarded and the fabric fully deployed, the topology can be viewed by navigating to **Main -> Topologies -> Physical**. See Figure 13 for reference.

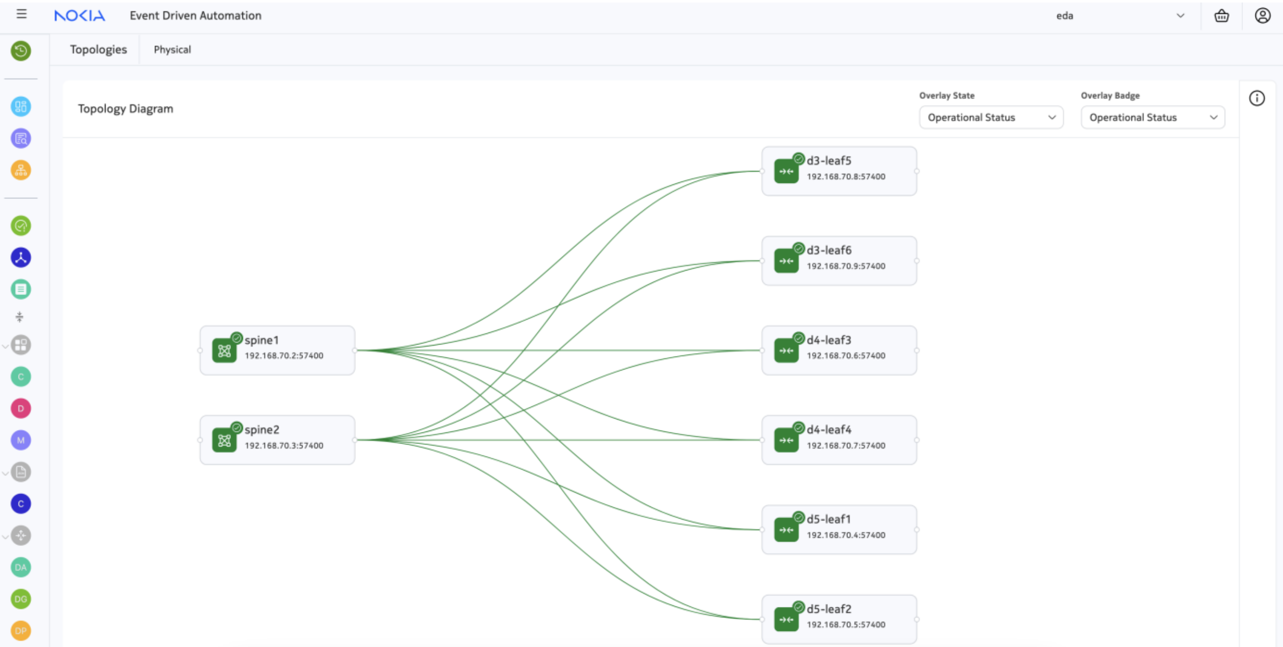

Figure 13. 3-stage EVPN VXLAN NVD fabric onboarded and deployed in EDA

6.2 EDA onboarding with ZTP

EDA has the capability to onboard fabric nodes via Zero Touch Provisioning (ZTP). EDA, as the ZTP server, can fully automate the end-to-end deployment of Nokia SRL nodes. Nodes which are in a factory default state only need to be plugged into the out-of-band (OOB) infrastructure and EDA can onboard the devices, along with pushing expected configuration (based on user intent) to them.

Figure 14 provides a high-level overview of the ZTP workflow and Example 19 displays console logs from a Nokia 7220 IXR-D4 being onboarded.

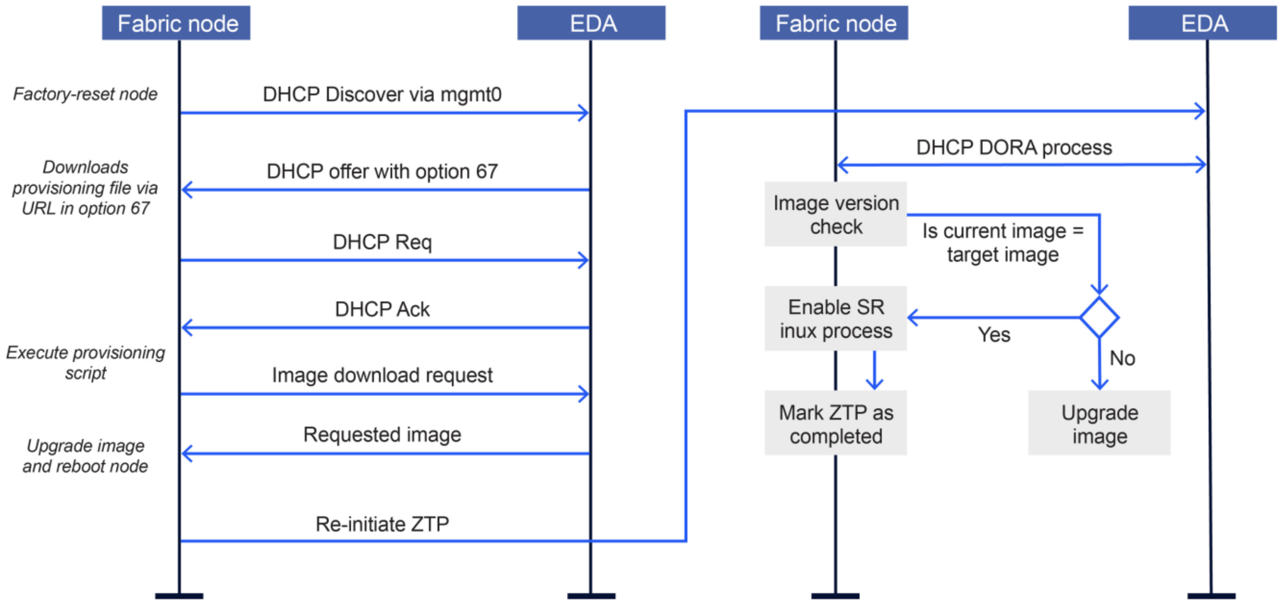

Figure 14. ZTP workflow

```
2024:12:24 12:08:36:51 | EVENT | ZTP Perform DHCP_V4
2024:12:24 12:08:36:72 | EVENT | Received dhcp lease on mgmt0 for 192.168.70.3/24, from server
100.116.161.50
2024:12:24 12:08:36:83 | EVENT | option 67 provided by dhcp:
http://100.116.161.50:9200/core/httpproxy/v1/asvr/eda/init-base/bootscript-d4-leaf4/d4-leaf4-
provision.py
2024:12:24 12:08:36:99 | EVENT | Updated hostname to d4-leaf4
2024:12:24 12:08:36:99 | EVENT | option 67 provided by dhcp:
http://100.116.161.50:9200/core/httpproxy/v1/asvr/eda/init-base/bootscript-d4-leaf4/d4-leaf4-
provision.py
2024:12:24 12:08:36:99 | EVENT | Url to fetch provisioning script:
http://100.116.161.50:9200/core/httpproxy/v1/asvr/eda/init-base/bootscript-d4-leaf4/d4-leaf4-
provision.py
2024:12:24 12:08:36:99 | EVENT | Executing provisioning script
2024:12:24 12:08:37:06 | EVENT | Downloaded provisioning script to
/etc/opt/srlinux/ztp/script/provision.py
2024:12:24 12:09:07:61 | EVENT | Upgrade failed: Recv failure: Connection reset by peer
2024:12:24 12:09:24:17 | EVENT | Installing image. Url:
http://100.116.161.50:9200/core/httpproxy/v1/asvr/eda-system/srlimages/srlinux-24.10.1-
bin/srlinux.bin
2024:12:24 12:09:32:34 | EVENT | Version of new image 24.10.1-492
2024:12:24 12:09:32:34 | EVENT | Current version: 24.10.1-492, New version: 24.10.1-492
2024:12:24 12:09:32:34 | EVENT | New image version 24.10.1-492 is same as active version 24.10.1-
492
2024:12:24 12:09:32:34 | EVENT | Not performing image upgrade
2024:12:24 12:09:37:61 | EVENT | Srlinux is running
2024:12:24 12:09:38:81 | EVENT | Execution of /etc/opt/srlinux/ztp/script/provision.py completed
with exit code 0
2024:12:24 12:09:38:81 | EVENT | Provisioning script execution successful
2024:12:24 12:09:38:82 | EVENT | Completed ZTP process
```

Example 19. Console logs on a Nokia 7220 IXR-D4 during successful ZTP onboarding

6.3 EDA Kubernetes workflow for NVD deployment

This section describes various manifest files that can be used to deploy an EDA-orchestrated EVPN VXLAN fabric in accordance with the prescriptive validated design described in this document.

6.3.1 EDA artifacts for SR Linux version 24.10.2

Kubernetes artifacts are created for target SR Linux version and used in the EDA node profile, Custom Resource. This includes the creation of manifest files for the .bin image, the md5 hash file, and the YAML zip file - samples of which are shown below.

```yaml
# artifacts for 24.10.2
apiVersion: artifacts.eda.nokia.com/v1
kind: Artifact
metadata:
  name: srlinux-24.10.2-bin
  namespace: eda-system
spec:
  repo: srlimages
  filePath: srlinux.bin
  remoteFileUrl:
    fileUrl: https://{file-path}/srlinux-24.10.2-357.bin
```

```
---
apiVersion: artifacts.eda.nokia.com/v1
kind: Artifact
metadata:
  name: srlinux-24.10.2-md5
  namespace: eda-system
spec:
  repo: srlimages
  filePath: srlinux.md5
  remoteFileUrl:
    fileUrl: https://{file-path}/srlinux-24.10.2-357.bin.md5
---
apiVersion: artifacts.eda.nokia.com/v1
kind: Artifact
metadata:
  name: srlinux-24.10.2
  namespace: eda-system
spec:
  repo: schemaprofiles
  filePath: srlinux-24.10.2.zip
  remoteFileUrl:
    fileUrl: https://{file-path}/srlinux-24.10.2-357.zip
```

Example 20. EDA artifact manifest

6.3.2 Subnet allocation for management of SR Linux fabric nodes

A manifest file is created to instantiate an IPv4/IPv6 subnet pool for the management of SR Linux fabric nodes.

```
# subnet allocation for IPv4 mgmt of SRL nodes
apiVersion: core.eda.nokia.com/v1
kind: IPInSubnetAllocationPool
metadata:
  name: hw-ipv4-mgmt-pool
  namespace: eda
spec:
  segments:
  - subnet: 192.168.70.0/24
    allocations:
    - name: gateway$$
      value: 192.168.70.1/24
---
# subnet allocation for IPv6 mgmt of SRL nodes
apiVersion: core.eda.nokia.com/v1
kind: IPInSubnetAllocationPool
metadata:
  name: hw-ipv6-mgmt-pool
  namespace: eda
spec:
```

```
  segments:
  - subnet: fd00:192:168:70::/64
    allocations:
    - name: gateway$$
      value: fd00:192:168:70:250::ffff/64
```

Example 21. EDA subnet and IP pool allocation manifest

6.3.3 EDA node profile for node onboarding

An EDA node profile facilitates the onboarding of fabric nodes, including the username/password for authentication into the node, a DHCP scope for assignment, and image version check (the node profile image is the expected target image).

```
# node profile for 24.10.2
apiVersion: core.eda.nokia.com/v1
kind: NodeProfile
metadata:
  name: real-srlinux-24.10.2
  namespace: eda
spec:
  dhcp:
    managementPoolv4: hw-ipv4-mgmt-pool
  images:
  - image: eda-system/srlimages/srlinux-24.10.2-bin/srlinux.bin
    imageMd5: eda-system/srlimages/srlinux-24.10.2-md5/srlinux.md5
  nodeUser: admin
  onboardingUsername: "admin"
  onboardingPassword: "NokiaSrl1!"
  operatingSystem: srl
  port: 57400
  version: 24.10.2
  versionMatch: v24\.10\.2.*
  versionPath: .system.information.version
  yang: https://eda-asvr.eda-system/eda-system/schemaprofiles/srlinux-24.10.2/srlinux-24.10.2.zip
```

Example 22. Node profile

6.3.4 Modify existing init-base CR to save on commit for SR Linux nodes

The existing init-base custom resource is modified to set commitSave to *true* so that SR Linux fabric nodes save to startup configuration on commit.

```
# modify exiting init-base CR to set commitSave to true
apiVersion: bootstrap.eda.nokia.com/v1alpha1
kind: Init
metadata:
  name: init-base
  namespace: eda
spec:
```

```
  commitSave: true
  mgmt:
    ipv4DHCP: true
    ipv6DHCP: true
```

Example 23. Resource to enable commit save to startup

6.3.5 Create node user to manage SR Linux nodes from EDA

The following example manifest file demonstrates how a node user is modified for management of SR Linux nodes from EDA. The nodeSelector label determines which nodes are allowed to be managed – an empty value selects all, as shown below.

```
# node user for SRL node management
# this modifies the default admin node user shipped with EDA v24.12.1
apiVersion: core.eda.nokia.com/v1
kind: NodeUser
metadata:
 name: admin
 namespace: eda
spec:
 groupBindings:
  - groups:
    - sudo
   nodeSelector:
    - ""
 password: NokiaSrl1!
 username: admin
```

Example 24. Node management

6.3.6 Onboarding nodes in EDA with using a TopoNode Custom Resource

SR Linux nodes can be onboarded into EDA using the TopoNode custom resource. This includes the creation of labels as metadata that will be attached to the node (these labels are used as selectors when deploying the fabric), a node profile name, the platform, and serial number of the node. See Example 25 for reference.

```
apiVersion: core.eda.nokia.com/v1
kind: TopoNode
metadata:
  labels:
    eda.nokia.com/hostname: spine2
    eda.nokia.com/role: spine
    eda.nokia.com/security-profile: managed
  name: spine2
  namespace: eda
spec:
  nodeProfile: real-srlinux-24.10.2
```

```
  npp:
    mode: normal
  onBoarded: false
  operatingSystem: srl
  platform: "7220 IXR-H4"
  version: 24.10.2
  serialNumber: {serial-number}
```

Example 25. TopoNode resource for node metadata

6.3.7 Building ASN pools for leafs and spines of the fabric

The following manifest file demonstrates how ASN pools can be built to be used during fabric deployment. In this case, two pools are created: *leaf-asn* and *spine-asn*.

```
# ASN pool creation for leafs
apiVersion: core.eda.nokia.com/v1
kind: IndexAllocationPool
metadata:
  name: leaf-asn
  namespace: eda
spec:
  segments:
    - start: 65411
      size: 20

# ASN pool creation for spines
apiVersion: core.eda.nokia.com/v1
kind: IndexAllocationPool
metadata:
  name: spine-asn
  namespace: eda
spec:
  segments:
    - start: 65500
      size: 10
```

Example 26. ASN pool allocation

6.3.8 System0 IP pool allocation

In an EVPN VXLAN fabric deployed with VXLAN tunnel endpoint (VTEP) functionality of SR Linux nodes, the system0 IP address is used as the VTEP source address by default. The following manifest file demonstrates how an IP allocation pool is created for assignment as system0 IP address on leaf nodes of the fabric.

```
# system0 IP pool allocation
apiVersion: core.eda.nokia.com/v1
kind: IPAllocationPool
metadata:
```

```
  name: system0
  namespace: eda
  labels: {}
  annotations: {}
spec:
  segments:
    - subnet: 192.0.2.0/24
      allocations: []
      reservations: []
```

Example 27. IP pool for system 0 addresses

6.3.9 Interface creation

The following manifest file demonstrates how interfaces are instantiated in EDA per onboarded SR Linux node. The example below uses a point-to-point interface connecting a leaf named d5-leaf1 to a spine named spine1.

```
# d5-leaf1 interface to spine1
apiVersion: interfaces.eda.nokia.com/v1alpha1
kind: Interface
metadata:
  labels:
    eda.nokia.com/role: interSwitch
  name: d5-leaf1-ethernet-1-29
  namespace: eda
spec:
  enabled: true
  lldp: true
  members:
    - enabled: true
      interface: ethernet-1-29
      node: d5-leaf1
  type: interface
```

Example 28. Interface resource

6.3.10 Link creation

The following manifest file demonstrates how links are created between two onboarded nodes in EDA.

```
# link between d5-leaf1 and spine1
apiVersion: core.eda.nokia.com/v1
kind: TopoLink
metadata:
  labels:
    eda.nokia.com/role: interSwitch
  name: d5-leaf1-spine1
  namespace: eda
```

```
spec:
  links:
    - local:
        node: d5-leaf1
        interface: ethernet-1-29
        interfaceResource: "d5-leaf1-ethernet-1-29"
      remote:
        node: spine1
        interface: ethernet-1-29
        interfaceResource: "spine1-ethernet-1-29"
      type: interSwitch
```

Example 29. TopoLink resource

6.3.11 Fabric creation (underlay and overlay)

The following manifest file demonstrates how an EVPN VXLAN fabric is orchestrated via EDA by using IPv6 link-local addressing and enabling MP-BGP peering (eBGP) between the leafs and the spines, carrying multiple address families. Several inputs are provided into the manifest file, which includes the IP pool for system0 assignment, ASN pools for leafs and spines, label selectors for leaf and spine nodes, and the interswitch links between the leafs and the spines. The fabric is also enabled for BFD.

```
# fabric creation with IPv6 link-local addressing (IPv6 unnumbered)
apiVersion: fabrics.eda.nokia.com/v1alpha1
kind: Fabric
metadata:
  name: dc1
  namespace: eda
spec:
  systemPoolIPV4: system0
  leafs:
    leafNodeSelector:
      - eda.nokia.com/role=leaf
    asnPool: leaf-asn
  spines:
    spineNodeSelector:
      - eda.nokia.com/role=spine
    asnPool: spine-asn
  interSwitchLinks:
    unnumbered: IPV6
    linkSelector:
      - eda.nokia.com/role=interSwitch
  underlayProtocol:
    protocol:
      - EBGP
    bfd:
      enabled: true
      detectionMultiplier: 3
      minEchoReceiveInterval: 250000
```

```
      desiredMinTransmitInt: 250000
      requiredMinReceive: 250000
    bgp:
      asnPool: asn-pool
  overlayProtocol:
    protocol: EBGP
```

Example 30. Underlay orchestration manifest

6.3.12 Bridge domain creation

Bridge domains, created in EDA for a VXLAN environment, are instantiated as MAC VRFs on SR Linux nodes. MAC VRFs map to a VXLAN VNI and an EVPN Instance (EVI), which enables it for EVPN learning.

```
# bridge domain created for VNI 10010
apiVersion: services.eda.nokia.com/v1alpha1
kind: BridgeDomain
metadata:
  name: macvrf-v10
  namespace: eda
spec:
  type: EVPNVXLAN
  vni: 10010
  evi: 10
  tunnelIndexPool: tunnel-index-pool
  macAging: 300
  macDuplicationDetection:
    enabled: true
    holdDownTime: 9
    monitoringWindow: 3
    action: StopLearning
    numMoves: 5
```

Example 31. MAC-VRFs, VNIs and EVIs

6.3.13 IRB interfaces

IRB interfaces, when deployed within the fabric, facilitate routing between L2 VNIs in an EVPN VXLAN deployment. The following manifest file demonstrates how IRB interfaces are created in EDA. In the case of a VLAN (bridge domain) that is Layer 2 stretched across the fabric, the Layer 3 proxy-ARP functionality should be enabled for the respective IRB sub interface.

```
# IRB interface for VLAN 10, VNI 10010
apiVersion: services.eda.nokia.com/v1alpha1
kind: IRBInterface
metadata:
  name: irb-v10
```

```
  namespace: eda
spec:
  bridgeDomain: macvrf-v10
  router: vrf1
  learnUnsolicited: BOTH
  ipMTU: 9200
  ipAddresses:
    - ipv4Address:
        ipPrefix: 172.16.10.254/24
        primary: true
  arpTimeout: 250
  evpnRouteAdvertisementType:
    arpDynamic: true
  hostRoutePopulate:
    dynamic: false
    evpn: false
    static: false
  l3ProxyARPND:
    proxyARP: true
    proxyND: false
```

Example 32. IRB interface manifest

6.3.14 IP VRF creation

The following manifest file demonstrates how IP VRFs are created in EDA. This includes a Layer 3 VNI (which has a 1:1 mapping to the IP VRF) and an EVPN Instance (EVI) along with a label selector to determine where the IP VRFs are deployed.

```
# VRF creation for vrf1
apiVersion: services.eda.nokia.com/v1alpha1
kind: Router
metadata:
  name: vrf1
  namespace: eda
spec:
  type: EVPNVXLAN
  vni: 10500
  evi: 500
  tunnelIndexPool: tunnel-index-pool
  nodeSelector:
    - eda.nokia.com/role=leaf
```

Example 33. VRF creation

6.3.15 VLAN creation

The following manifest file demonstrates how VLANs are created in EDA, using examples of an untagged and tagged Layer 2 deployment. The label selectors determine which interfaces the VLANs are deployed on.

```yaml
# VLAN creation for an untagged Layer 2 interface
apiVersion: services.eda.nokia.com/v1alpha1
kind: VLAN
metadata:
  name: untagged-v10
  namespace: eda
spec:
  bridgeDomain: macvrf-v10
  interfaceSelector:
    - eda.nokia.com/untagged-v10=enabled
  vlanID: untagged
---
# VLAN creation for an tagged Layer 2 interface
apiVersion: services.eda.nokia.com/v1alpha1
kind: VLAN
metadata:
  name: tagged-v20
  namespace: eda
spec:
  bridgeDomain: macvrf-v20
  interfaceSelector:
    - eda.nokia.com/tagged-v20=enabled
  vlanID: "20"
```

Example 34. VLAN creation

6.3.16 EDA configlets

EDA provides the flexibility to input user-defined configuration (that may not be auto-generated by EDA in a particular version). This functionality is achieved via configlets. The 3-stage EVPN VXLAN validated design uses configlets for the following purposes:

- Enable BGP rapid advertisement and withdraw
- Enable Designated Forwarder (DF) election activation timer for Ethernet Segment (for transition from non-DF to DF)
- Enable the advertisement of ARP/ND entries only when corresponding MAC entries exist for a MAC VRF
- Enable node isolation functionality
- Configure system-wide default MTUs

The following manifest file shows an example configlet (using BGP rapid advertisement and withdraw as a reference).

```yaml
# configlet for BGP EVPN rapid withdraw
apiVersion: config.eda.nokia.com/v1alpha1
kind: Configlet
metadata:
  name: bgp-evpn-rapid
  namespace: eda
```

```
spec:
  endpointSelector:
    - eda.nokia.com/role=leaf
    - eda.nokia.com/role=spine
  operatingSystem: srl
  priority: 100
  configs:
    - path: .network-instance{.name=="default"}.protocols.bgp.afi-safi{.afi-safi-
name=="evpn"}.evpn
      operation: Update
      config: |-
        {
          "rapid-update": "true"
        }
---
# configlet for BGP rapid withdrawal
apiVersion: config.eda.nokia.com/v1alpha1
kind: Configlet
metadata:
  name: bgp-rapid-route-withdraw
  namespace: eda
spec:
  endpointSelector:
    - eda.nokia.com/role=leaf
    - eda.nokia.com/role=spine
  operatingSystem: srl
  priority: 100
  configs:
    - path: .network-instance{.name=="default"}.protocols.bgp.route-advertisement
      operation: Update
      config: |-
        {
          "rapid-withdrawal": "true"
        }
```

Example 35. Configlet – BGP EVPN rapid withdrawal and EVPN rapid update

6.4 EDA workflows via user interface (UI)

6.4.1 Node profiles for node onboarding

Node profiles are specified during node onboarding and are used to determine the IP pool from which to assign an IP address to the node, what the gNMI discovery port is, and the username/password credentials to log into the device. Node profiles can be created by navigating to **Main -> Node Profiles**.

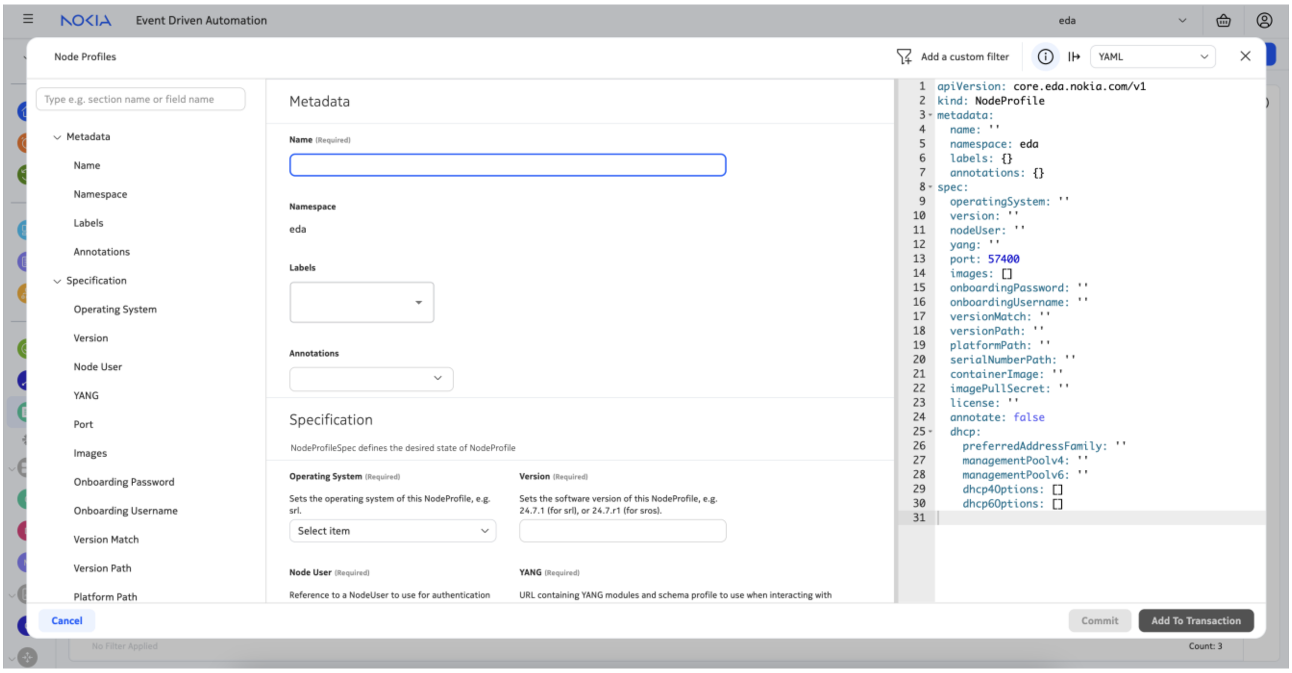

Figure 15. Node profile creation page in EDA UI

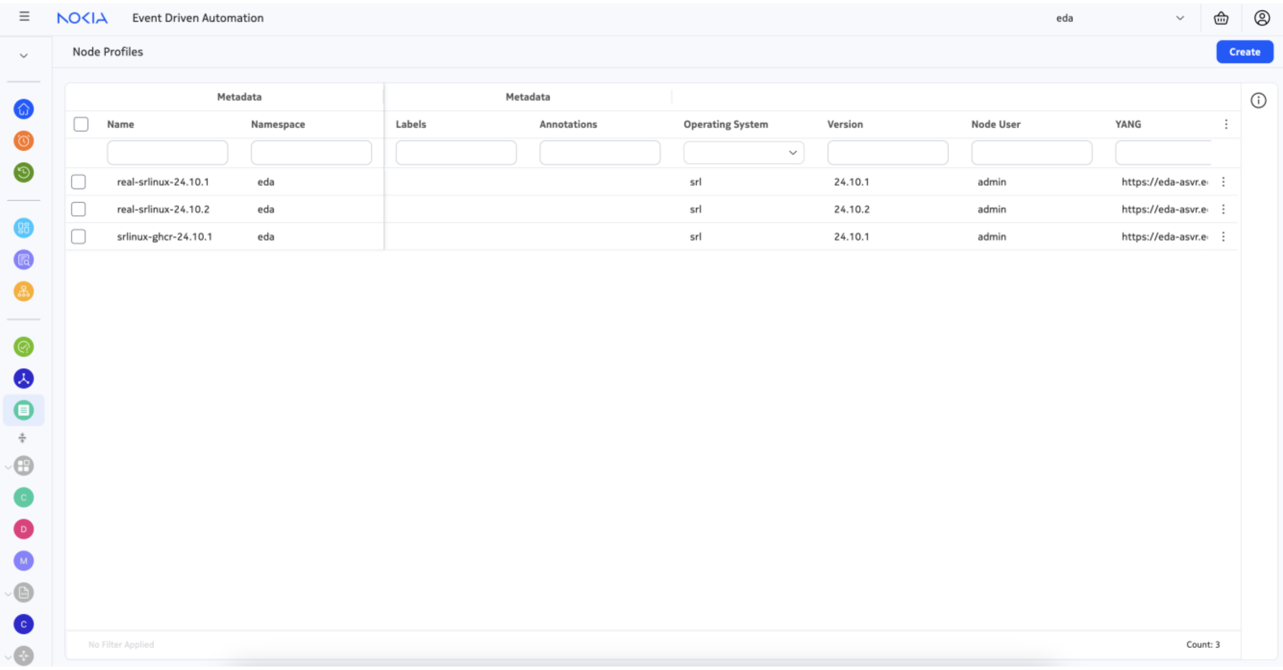

Figure 16. List of all created node profiles (default and user-defined) in EDA UI

6.4.2 ASN pools for leafs and spines

The ASN pools are created as indices pools, which can then be assigned to leafs and spines during fabric creation. These indices pools can be viewed and created by navigating to **Main -> Indices**.

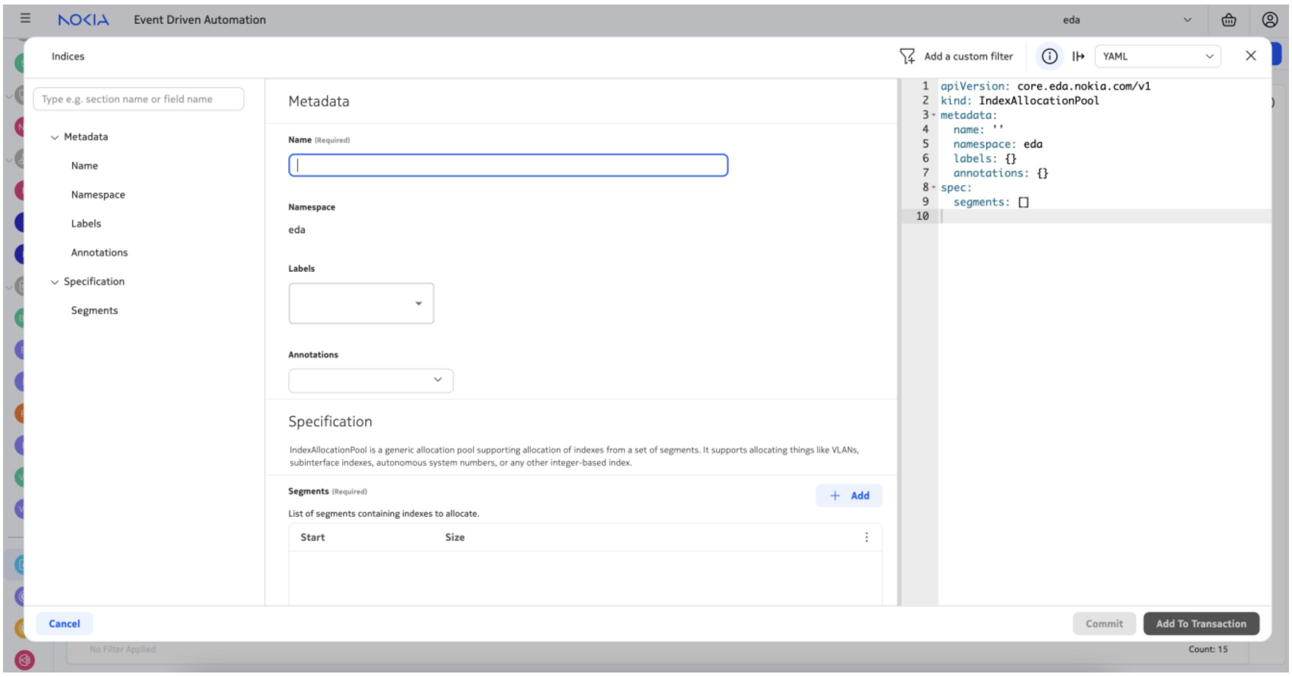

Figure 17. ASN creation as an indices pool in EDA UI

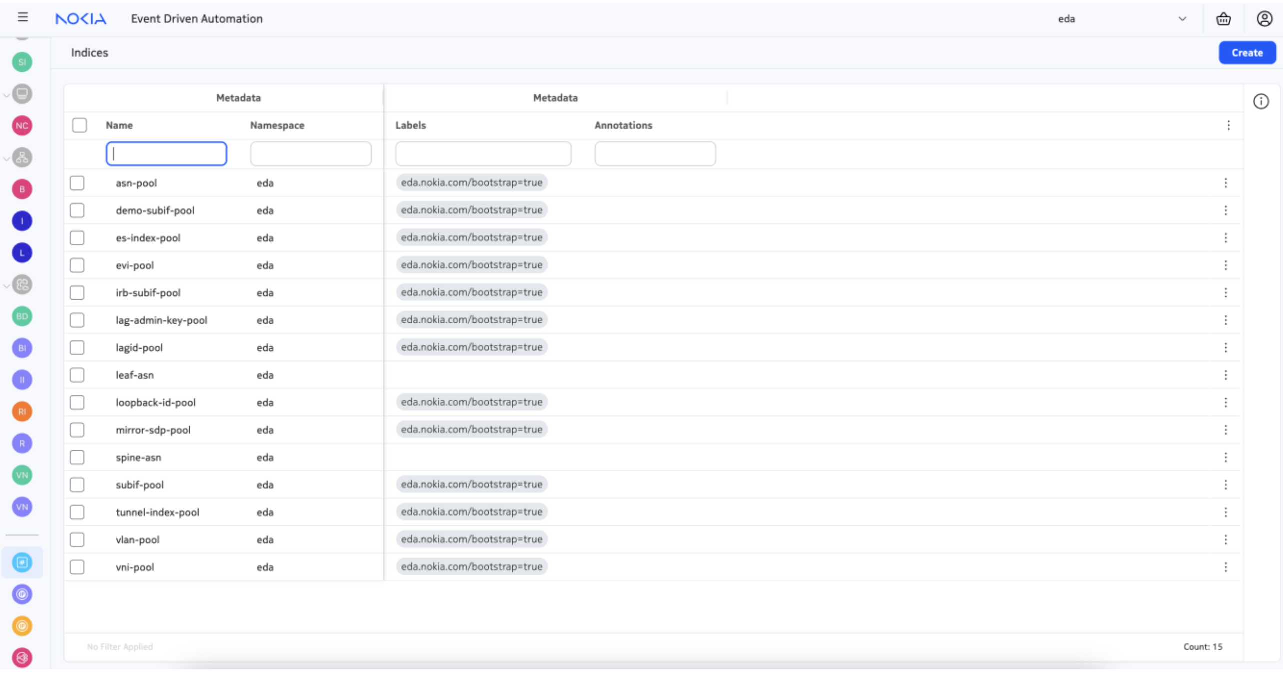

Figure 18. List of all indices pools (default and user-defined) in EDA UI

6.4.3 IP pool creation allocation

IP pools can be created for multiple reasons – a subnet allocation or an exact IP address allocation, for example. In the case of this NVD, an IP pool of type *IP Addresses* is created to assign a unique IPv4 address from an IPv4 subnet for the system0 interface of nodes in the fabric. This can be created by navigating to **Main -> IP Addresses**.

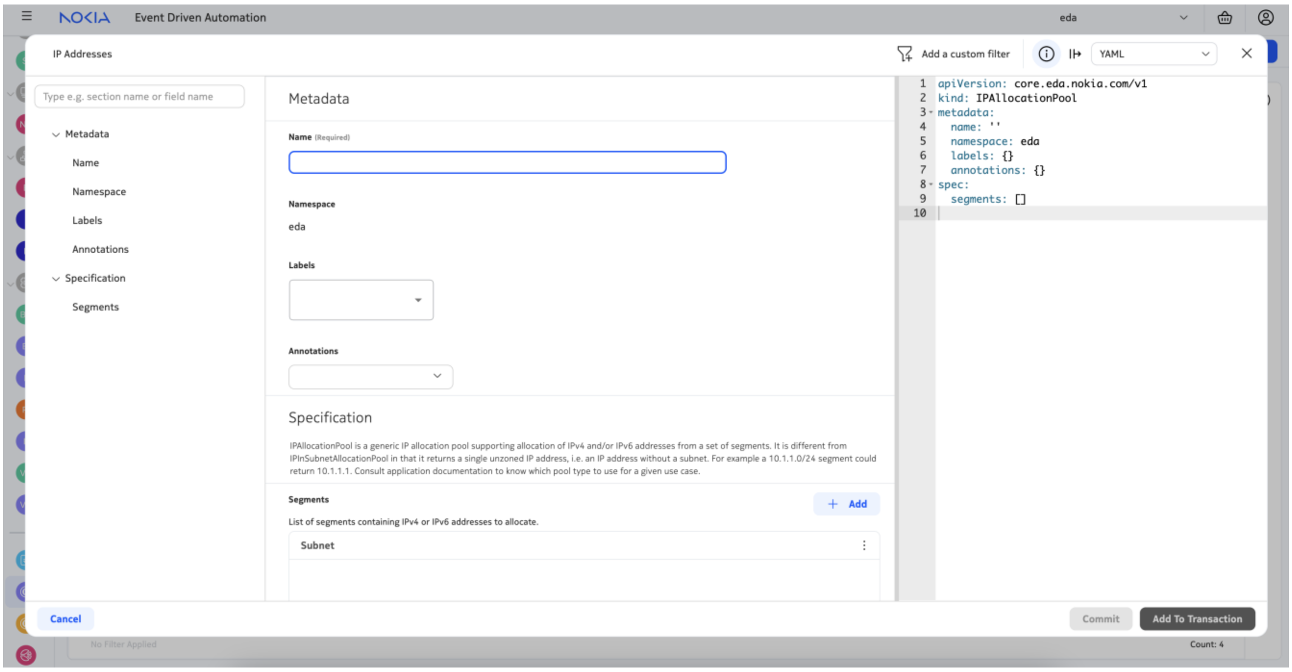

Figure 19. IP pool creation in EDA UI

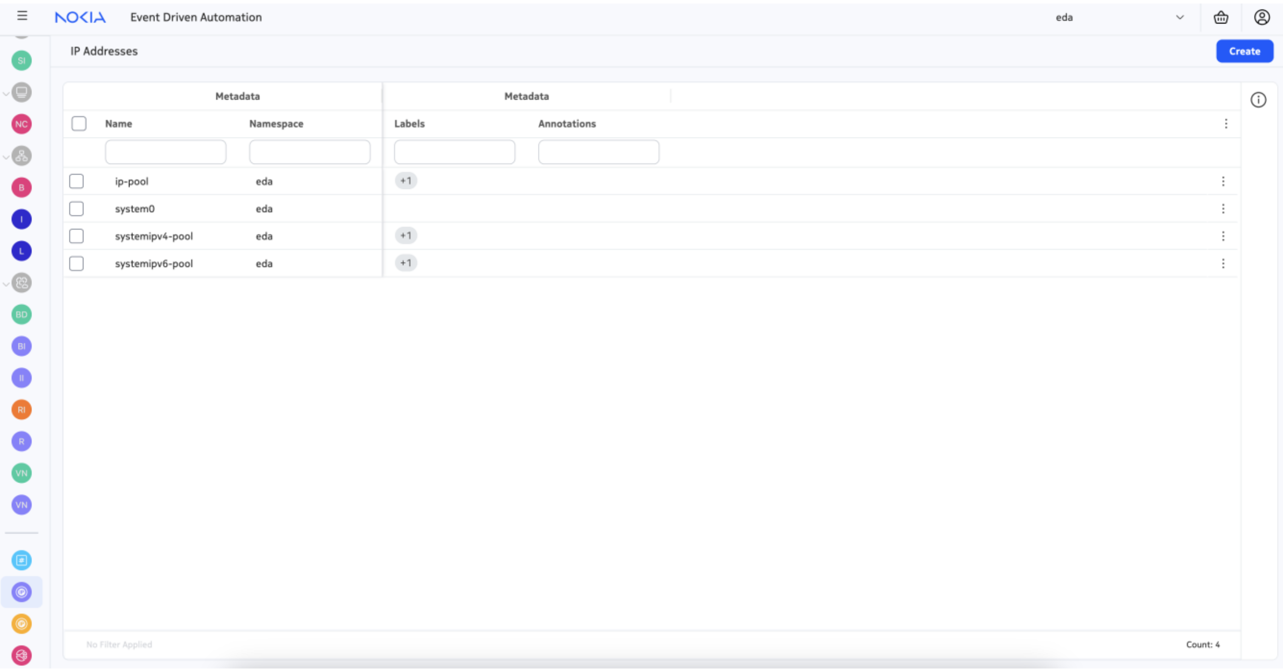

Figure 20. List of all IP pools (default and user-defined) in EDA UI

6.4.4 Onboarding nodes

Nodes can be created and viewed by navigating to **Main -> Nodes**. These are nodes onboarded into the fabric and represented in the topology view.

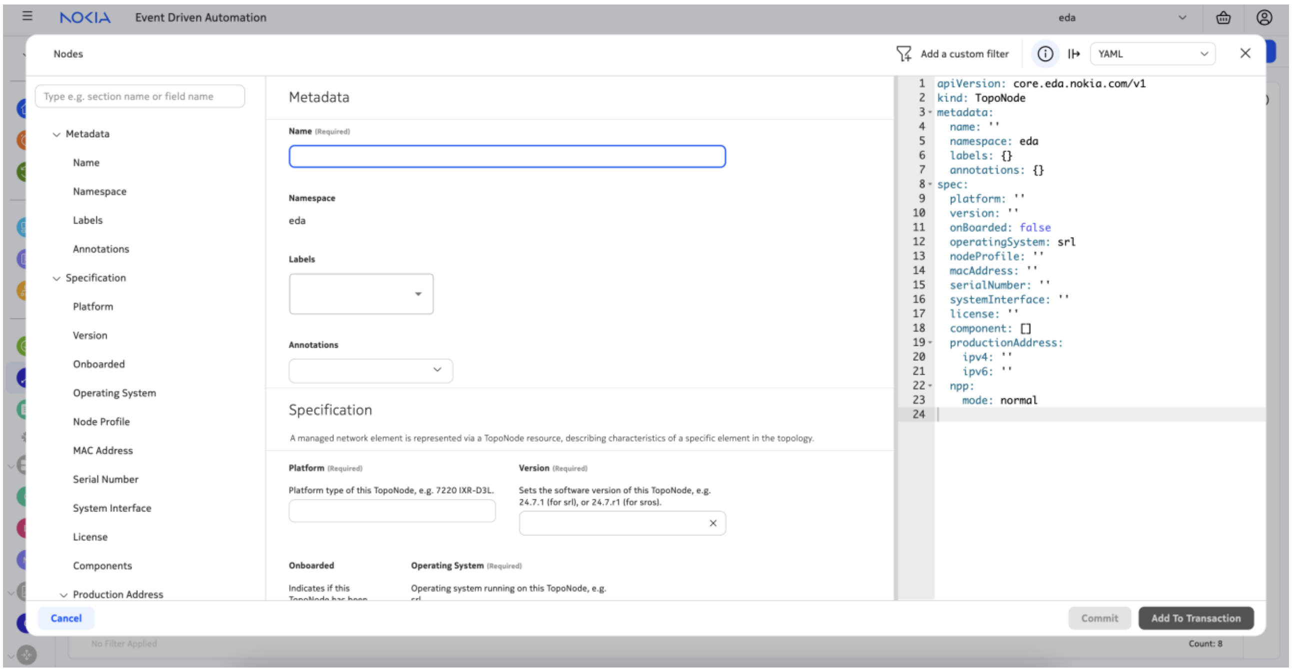

Figure 21. Node creation (onboarding) in EDA UI

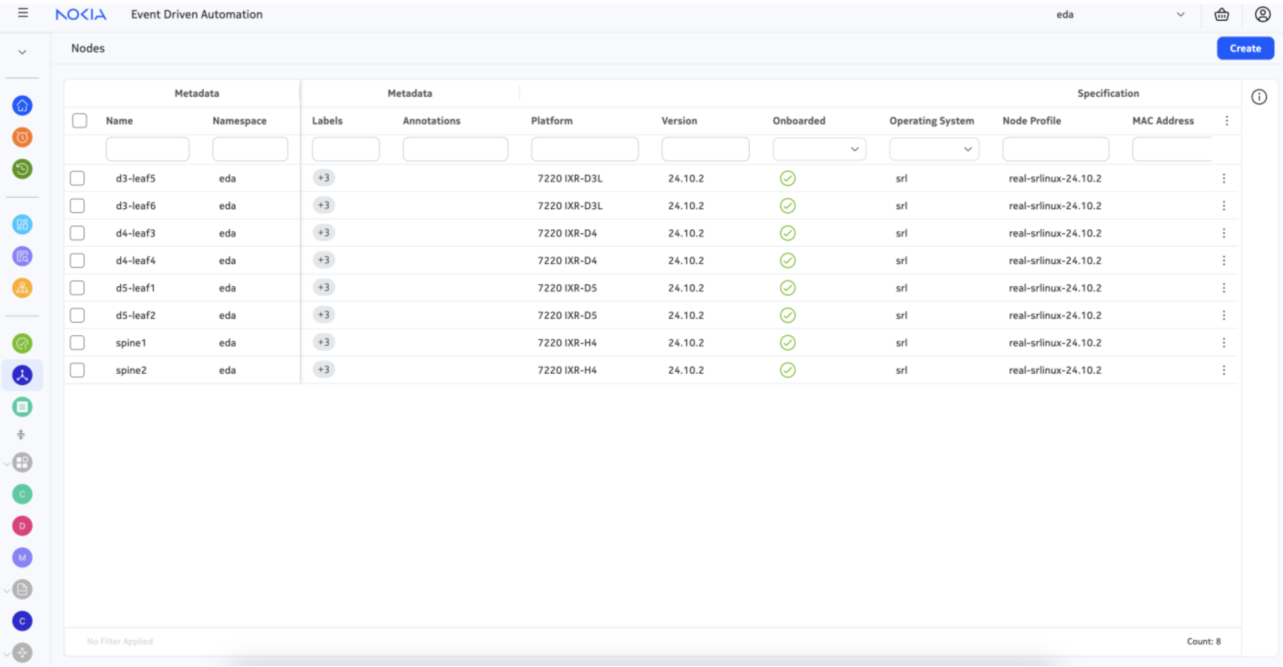

Figure 22. List of all onboarded nodes (along with different monitored parameters) in EDA UI

6.4.5 Fabric creation

Fabrics can be created by navigating to **Main -> Fabrics**. This instantiates all fabric nodes (based on label selector) and pushes the generated fabric configuration per-node.

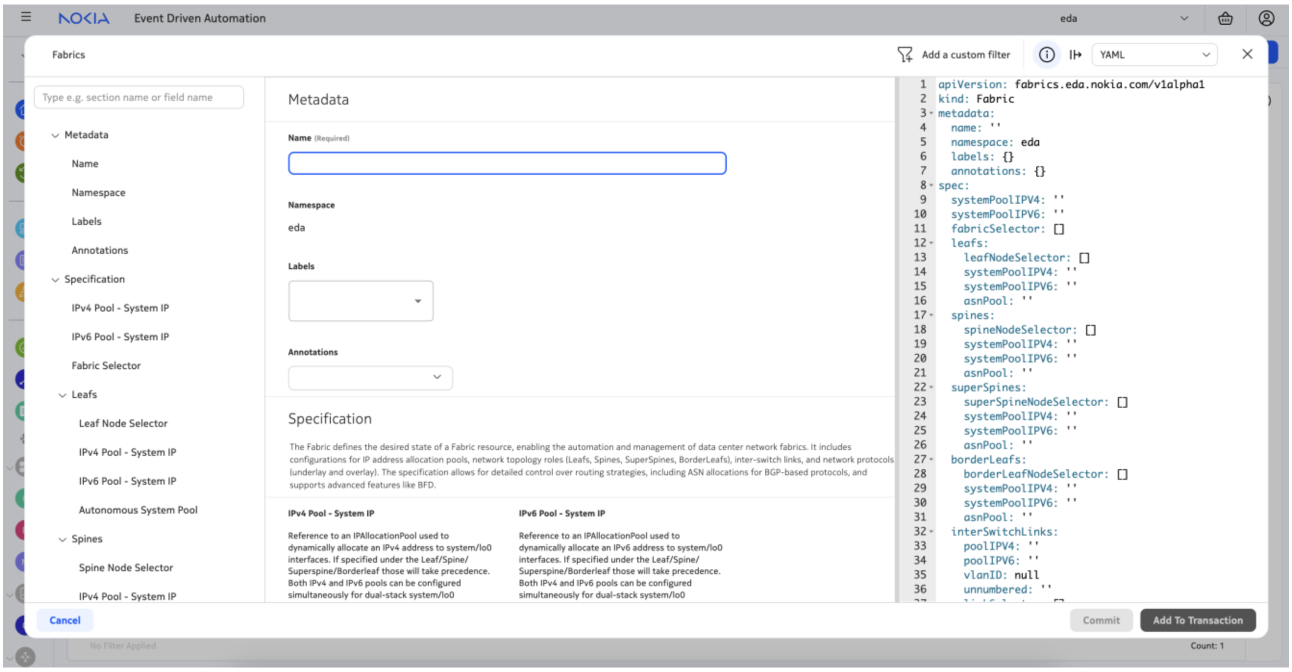

Figure 23. Fabric creation in EDA UI

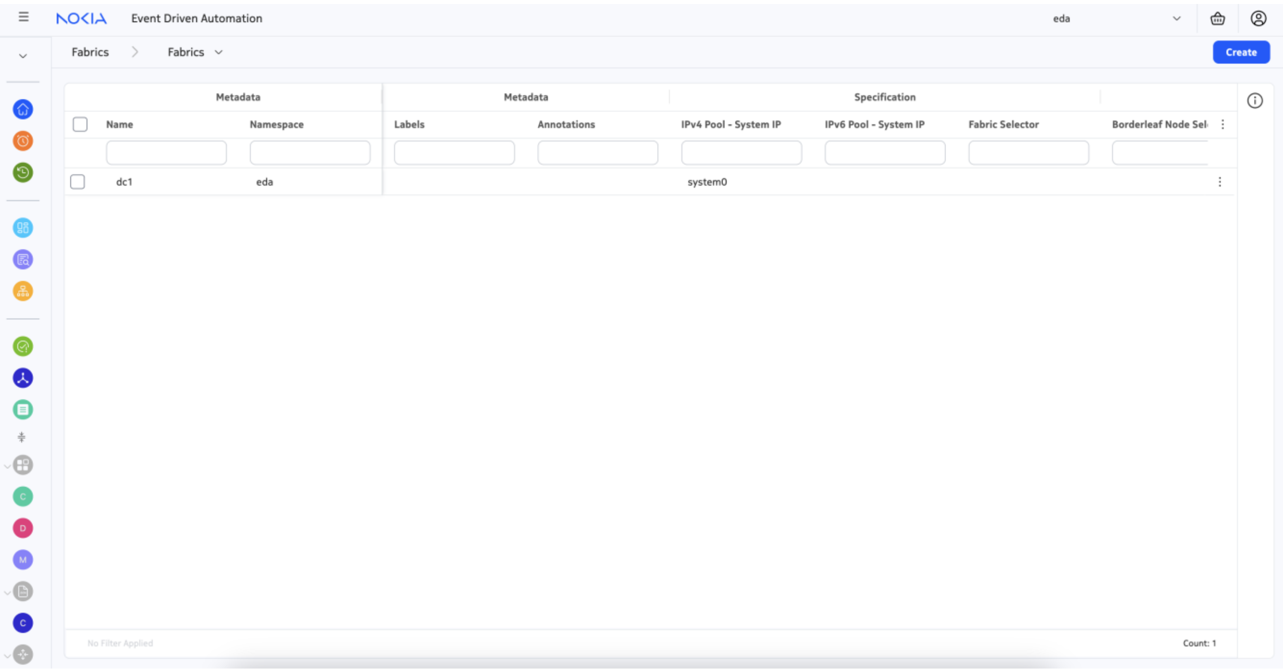

Figure 24. List of all fabrics in EDA UI

6.4.6 Bridge domains

Bridge domains are instantiated as MAC VRFs on fabric nodes and can be created by navigating to **Main -> Virtual Networks -> Bridge Domains**.

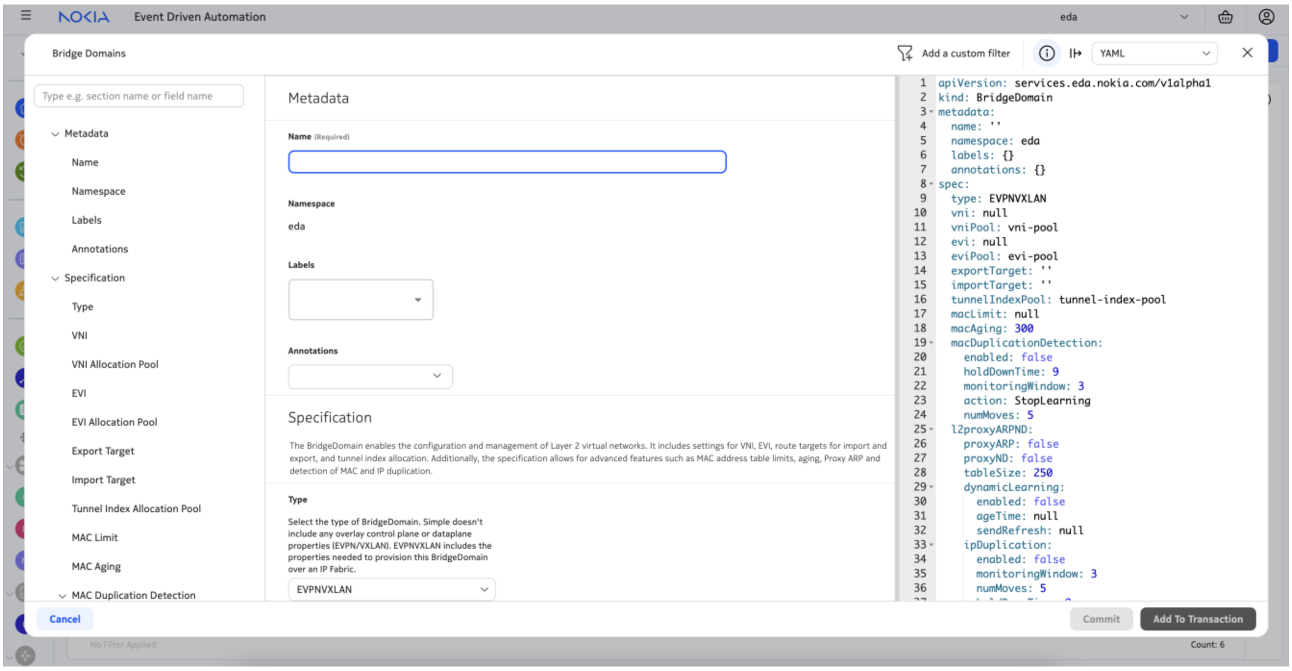

Figure 25. Bridge domain creation in EDA UI

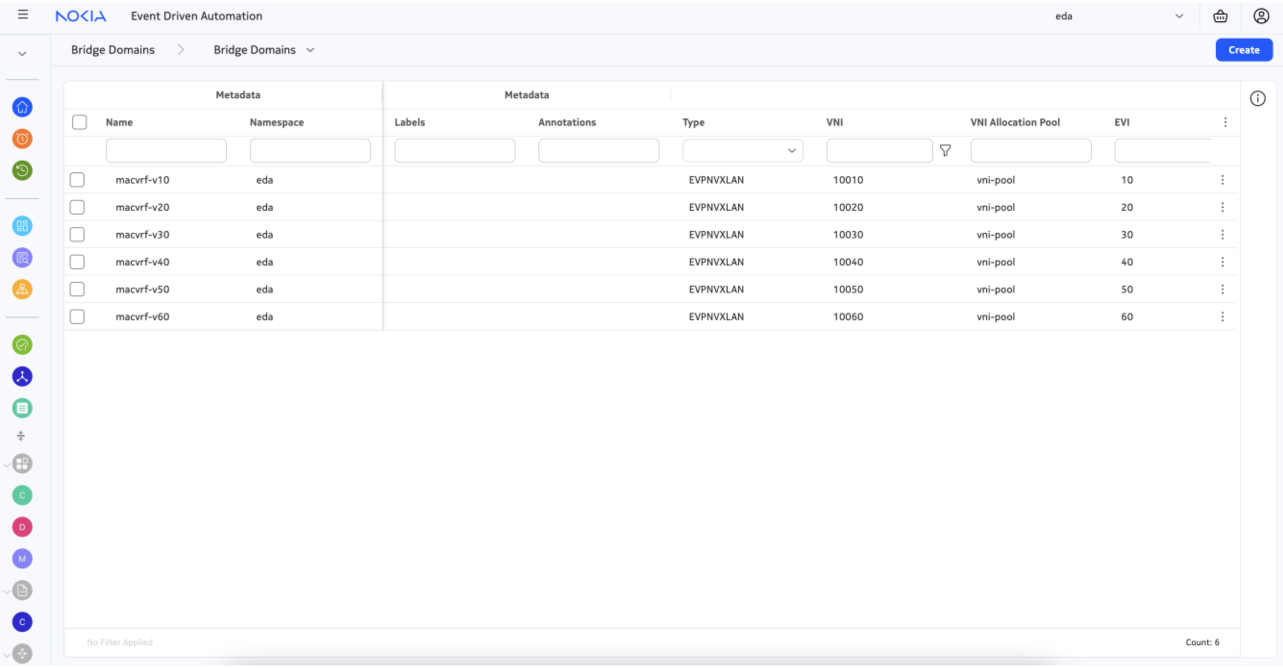

Figure 26. List of all bridge domains in EDA UI

6.4.7 IRB interfaces

IRB interfaces act as the default gateway for services connected to the leafs and are deployed using an anycast, distributed gateway model. IRB interfaces can be created by navigating to **Main -> Virtual Networks -> IRB Interfaces**.

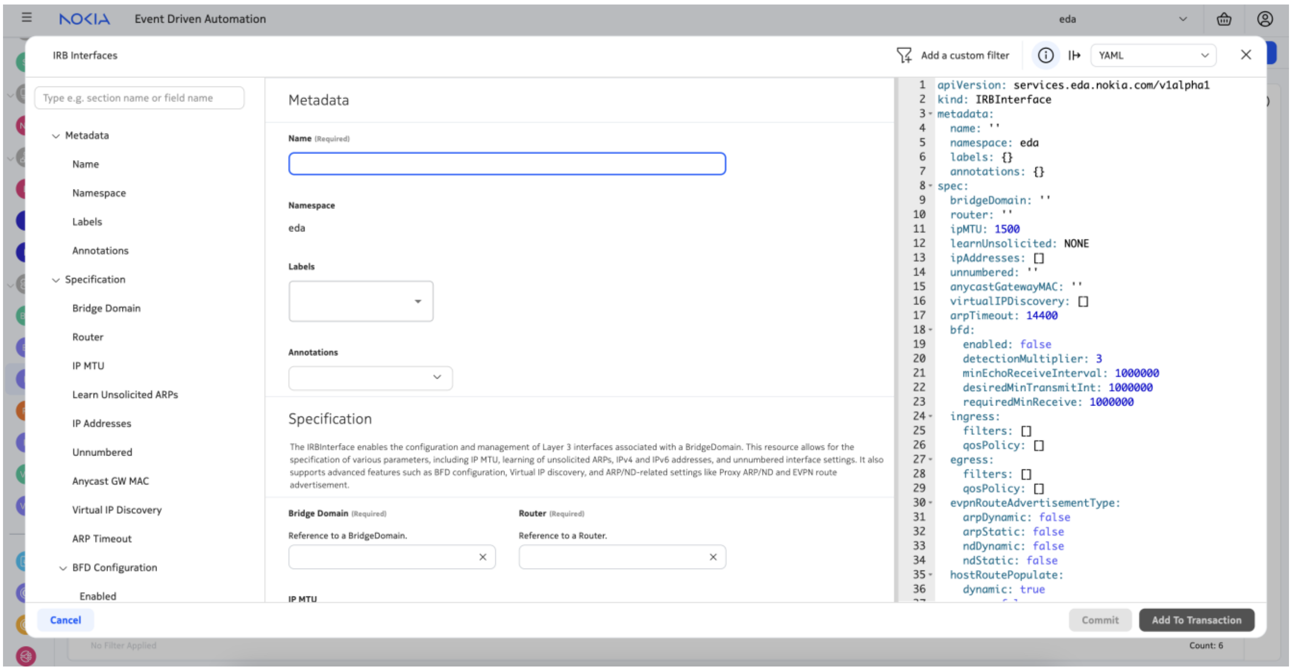

Figure 27. IRB interface creation in EDA UI

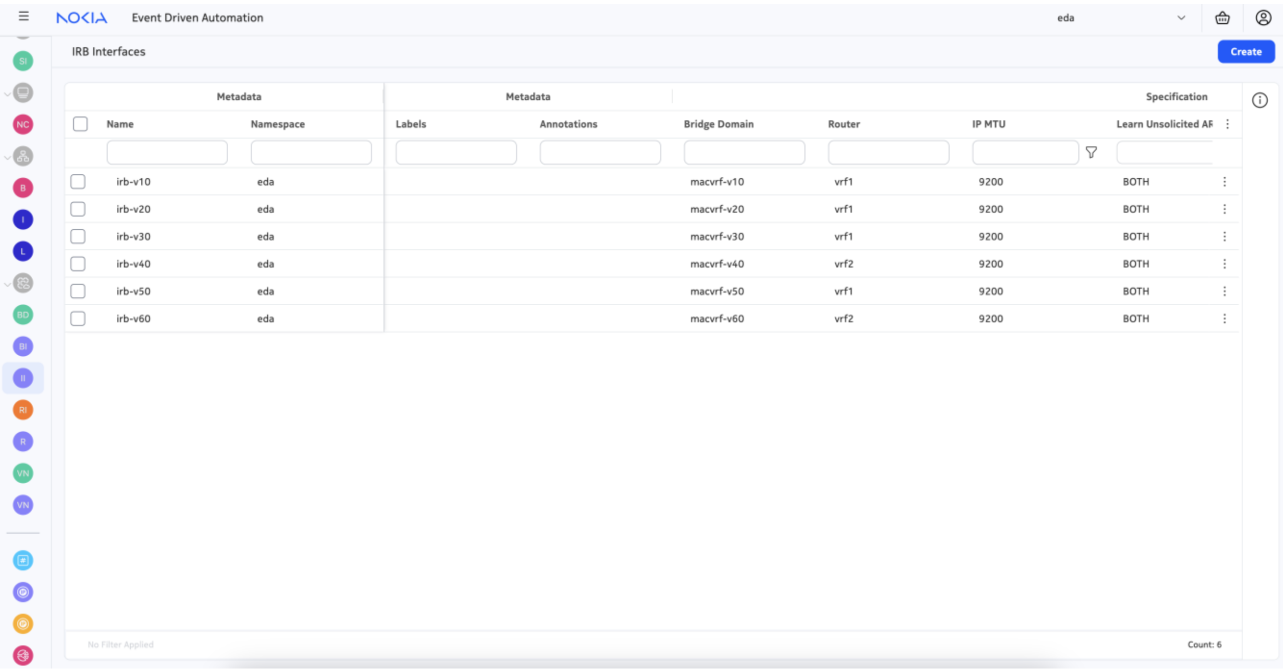

Figure 28. List of all IRB interfaces in EDA UI

6.4.8 IP VRFs (Routers)

IP VRFs are used to provide multitenancy and Layer 3 isolation. IP VRFs can be created by navigating to **Main -> Virtual Networks -> Routers**.

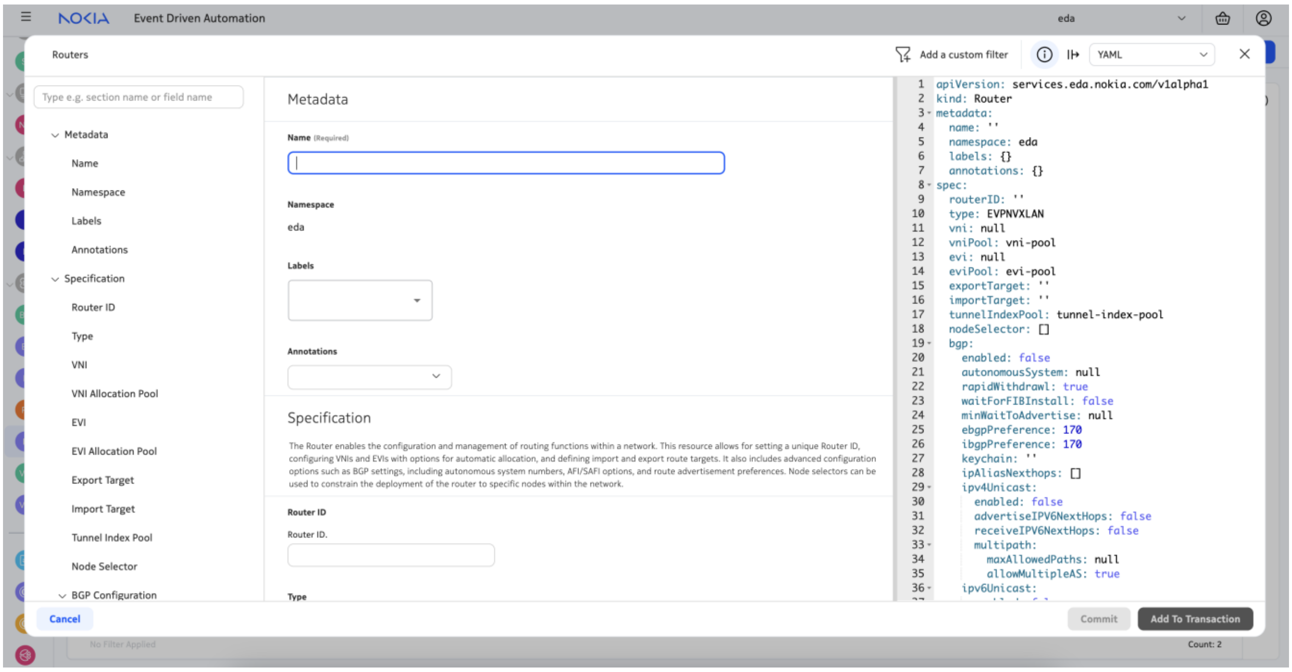

Figure 29. IP VRF (router) creation in EDA UI

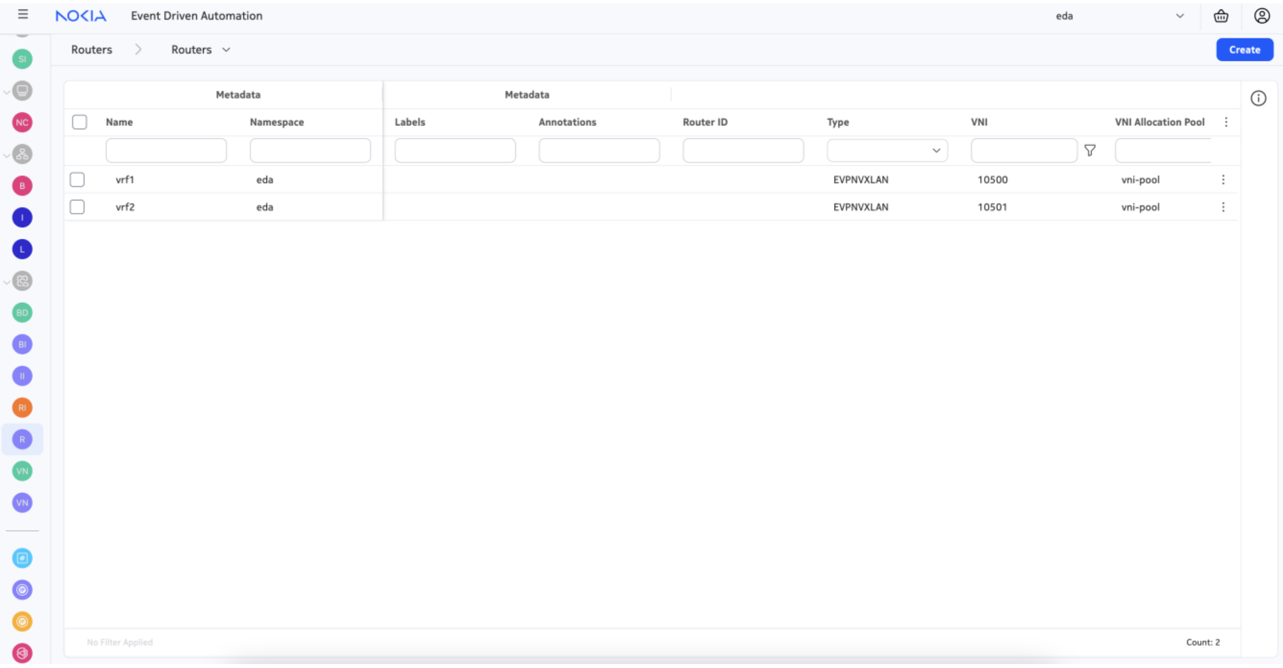

Figure 30. List of all IP VRFs in EDA UI

6.4.9 VLANs

VLANs can be created by navigating to **Main -> Virtual Networks -> VLANs**.

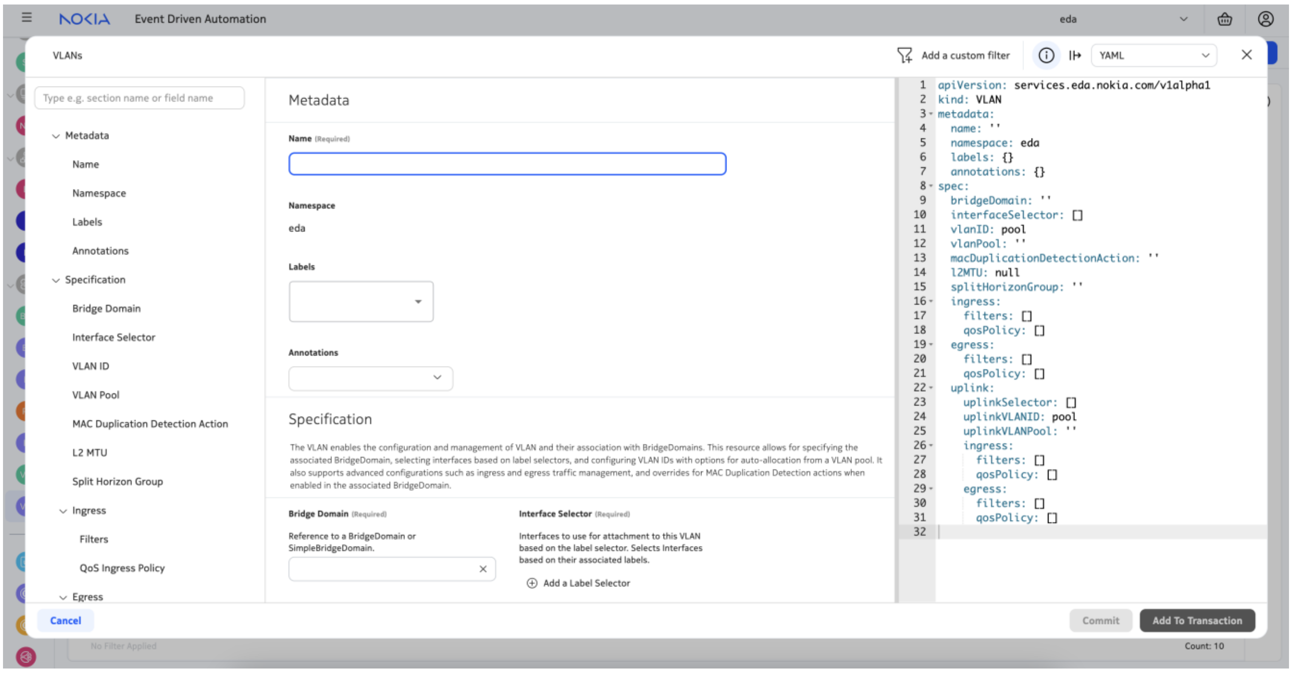

Figure 31. VLAN creation in EDA UI

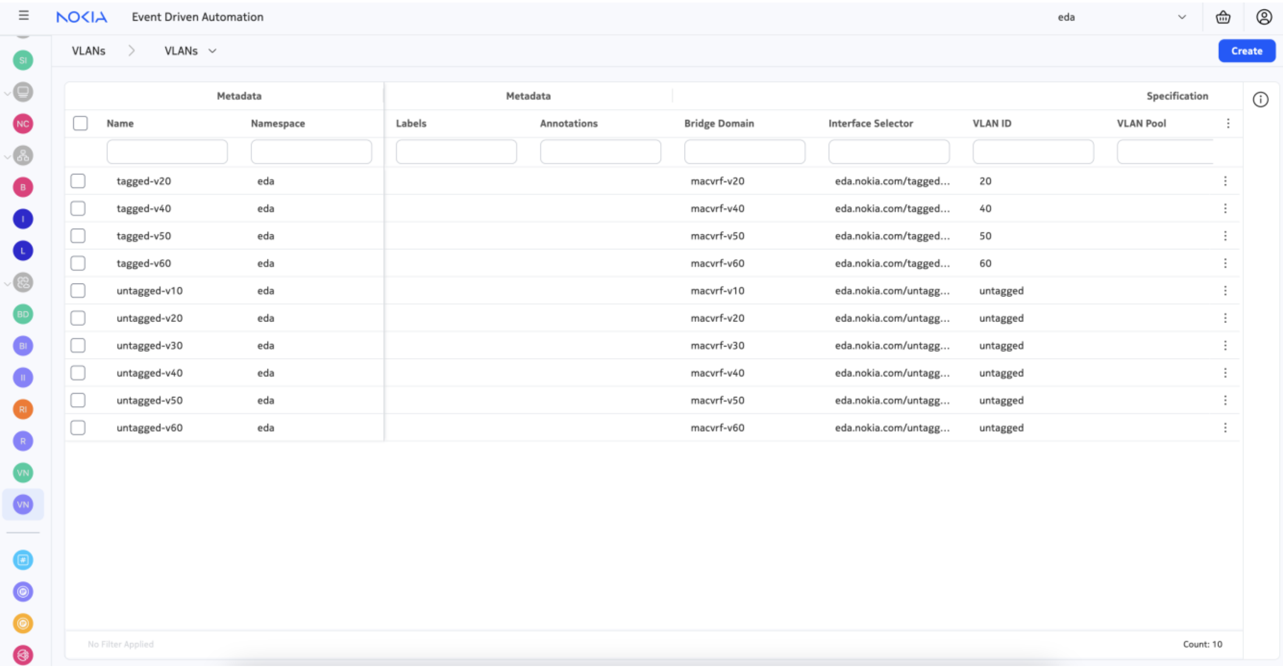

Figure 32. List of all VLANs in EDA UI

6.4.10 Configlets for custom configuration

Configlets allow for supplemental configuration that can be added to the per-node configuration generated by EDA. Configlets can be created by navigating to **Main -> Configuration -> Configlets**.

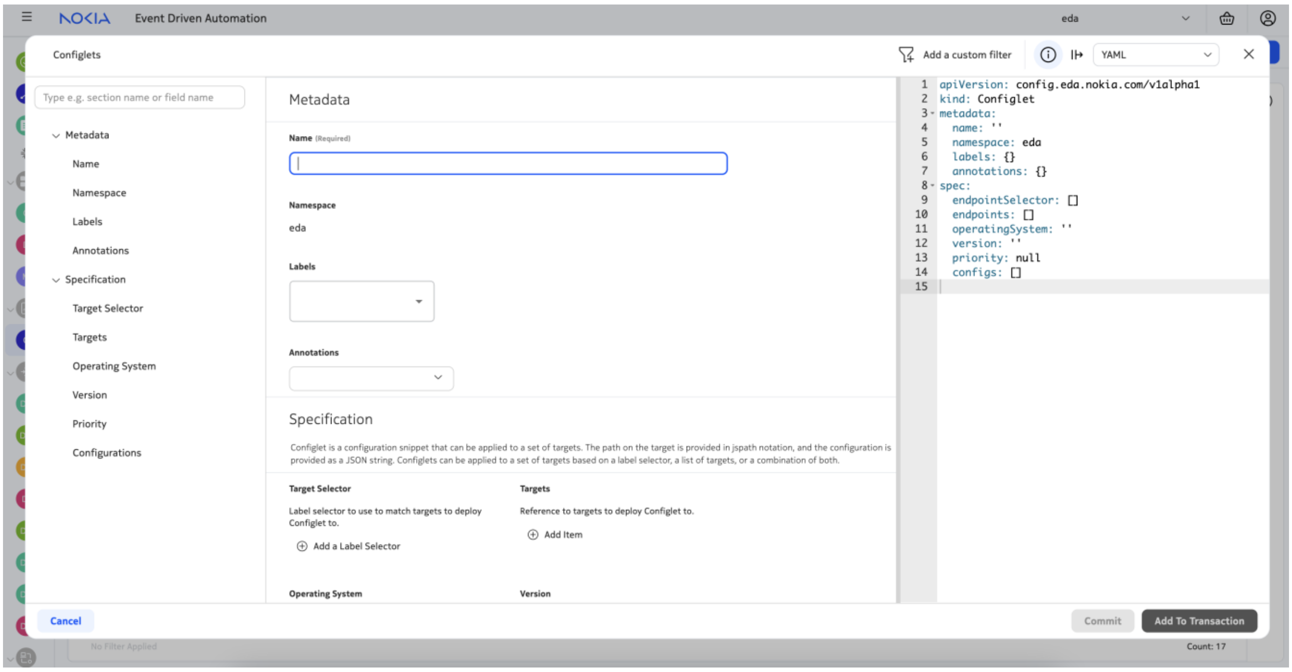

Figure 33. Configuration configlets creation in EDA UI

7 Validation

7.1 Network validation

7.1.1 Underlay and overlay

The underlay comprises of point-to-point IPv6 link-local addressing with IPv6 Neighbor Discovery to discover the peer on its local link. BGP dynamic discovery is then used to establish MP-BGP peering with the neighbor, with IPv4, IPv6, and EVPN address families (and sub-address families) exchanged as capabilities over this single peering.

The discovered neighbors on an IPv6 interface can be confirmed using *info from state interface [interface] subinterface [subinterface] ipv6 neighbor-discovery*.

```
A:d5-leaf1# info from state interface ethernet-1/29 subinterface 0 ipv6 neighbor-discovery
neighbor * | as yaml
---
interface:
  - name: ethernet-1/29
    subinterface:
      - index: 0
        ipv6:
          neighbor-discovery:
            neighbor:
              - ipv6-address: 'fe80::429b:21ff:fed8:83f0'
                link-layer-address: '40:9B:21:D8:83:F0'
                origin: dynamic
                is-router: true
                current-state: stale
                next-state-time: '2024-10-27T21:39:53.886Z (3 minutes from now)'
                datapath-programming:
```

```
                   status: success
```

Example 36. Interface discovery

The state of BGP neighbors can be confirmed using *show network-instance default protocols bgp neighbor*. This is with the assumption that the BGP peers are configured for the default network-instance.

```
A:d5-leaf1# show network-instance default protocols bgp neighbor
fe80::429b:21ff:fed8:83f0%ethernet-1/29.0 | as yaml
---
neighbor summary:
  - network-instance: default
    state:
      - Net-Inst: default
      - Peer: 'fe80::429b:21ff:fed8:83f0%ethernet-1/29.0'
        Group: bgpgroup-ebgp-dc1
        Flags: DB
        Peer-AS: 65500
        State: established
        Uptime: '8d:0h:22m:30s'
        AFI/SAFI: evpn\nipv4-unicast\nipv6-unicast

*snip*
```

Example 37. BGP neighborship

BFD is used for fast-failover in the NVD. The BFD session state can be confirmed using *info from state bfd network-instance default peer [index]*.

```
A:d5-leaf1# info from state bfd network-instance default peer 16385 | as yaml
---
bfd:
  network-instance:
    - name: default
      peer:
        - local-discriminator: 16385
          oper-state: up
          ipv6-link-local-interface: ethernet-1/29.0
          local-address: 'fe80::ca72:7eff:fe10:e2a3'
          remote-address: 'fe80::429b:21ff:fed8:83f0'
          remote-discriminator: 16397
          subscribed-protocols: BGP
          session-state: UP
          remote-session-state: UP
          last-state-transition: '2024-10-19T21:16:27.634Z (8 days ago)'
          failure-transitions: 0
          local-diagnostic-code: NO_DIAGNOSTIC
          remote-diagnostic-code: NO_DIAGNOSTIC
          remote-minimum-receive-interval: 250000
          remote-control-plane-independent: false
          active-transmit-interval: 250000
          active-receive-interval: 250000
          remote-multiplier: 3
          async:
            last-packet-transmitted: '2024-10-27T21:42:06.897Z (a second ago)'
            last-packet-received: '2024-10-27T21:42:06.817Z (a second ago)'
            transmitted-packets: 3497295
            received-packets: 3497461
```

```
            up-transitions: 1
```

Example 38. BFD information

In addition to viewing routes in BGP RIB-In using *show network-instance default protocols bgp routes [family] summary*, all routes advertised and received via BGP can also be confirmed using the commands *show network-instance default protocols bgp neighbor [neighbor] advertising-routes [family]* and *show network-instance default protocols bgp neighbor [neighbor] received-routes [family]*.

```
A:d5-leaf1# show network-instance default protocols bgp routes ipv4 summary | as yaml
---
header:
  - Header: default
    net-inst: default
    routes:
      - Status: u*>
      - Network: 192.0.2.1/32
      - Next Hop: 'fe80::22de:1eff:fea4:524%ethernet-1/30.0'
        LocPref: 100
        Path Val: ' i[65500, 65414]'
      - Status: u*>
      - Network: 192.0.2.1/32
      - Next Hop: 'fe80::429b:21ff:fed8:83f0%ethernet-1/29.0'
        LocPref: 100
        Path Val: ' i[65500, 65414]'
```

Example 39. BGP routes

7.1.2 Link aggregation

Interface state (and overall LAG state) can be viewed using *show lag [lag-interface] brief* or *show lag [lag-interface] lacp-state* for LACP-enabled LAGs.

```
A:d5-leaf1# show lag lag1 lacp-state  | as yaml
---
LacpHeader:
  - Lag Id: lag1
    LacpBrief:
      Interval: FAST
      Mode: ACTIVE
      System Id: '00:00:11:22:33:44'
      System Priority: 32768
    LacpState:
      - Members: ethernet-1/3
        Oper state: up
        Activity: ACTIVE
        Timeout: SHORT
        State: IN_SYNC/True/True/True
        System Id: '00:00:11:22:33:44'
        Oper key: 1
        Partner Id: '00:00:00:00:00:11'
        Partner Key: 32768
        Port No: 1
        Partner Port No: 1
```

Example 40. LAG state information

7.1.3 Ethernet segments

For Ethernet segments, Designated Forwarder (DF) and non-DF status can be determined on a per VRF basis.

```
A:d5-leaf1# show system network-instance ethernet-segments detail | as yaml
---
Ethernet-Segment:
  - Name: leaf1-leaf2-leaf3-leaf4-lag1
    Admin State: enable
    Oper State: up
    ESI: '00:00:00:11:22:33:44:00:00:00'
    Multi-homing: all-active
    Oper Multi-homing: all-active
    Interface: lag1
    Next-hop: N/A
    evi: N/A
    ES Activation Timer: 0
    DF Election: default
    Oper DF Election: default
    Last change: '2024-10-27T22:56:14.342Z'
    TimerInfo:
      - MAC-VRF: leaf1-leaf2-leaf3-leaf4-lag1
        Actv Timer Rem: 0
        DF: Yes
    NetworkInstance:
      - Network-instance: macvrf-v10
      - ES Peers: 192.0.2.1
      - Network-instance: macvrf-v10
      - ES Peers: 192.0.2.3
      - Network-instance: macvrf-v10
      - ES Peers: 192.0.2.4 (DF)
      - Network-instance: macvrf-v10
      - ES Peers: 192.0.2.6
      - Network-instance: macvrf-v50
      - ES Peers: 192.0.2.1
      - Network-instance: macvrf-v50
      - ES Peers: 192.0.2.3
      - Network-instance: macvrf-v50
      - ES Peers: 192.0.2.4 (DF)
      - Network-instance: macvrf-v50
      - ES Peers: 192.0.2.6
```

Example 41. Ethernet segment description

7.1.4 MAC VRFs and MAC address learning

The bridge table per MAC-VRF can be viewed using the commands given below.

```
A:d5-leaf1# show network-instance macvrf-v10 summary | as yaml
---
Network Instance:
  - Name: macvrf-v10
    Type: mac-vrf
    Admin state: enable
    Oper state: up
    Router id: N/A
    Description: macvrf-v10

A:d5-leaf1# show network-instance macvrf-v10 bridge-table mac-table all | as yaml
```

```
---
Network:
  - Name: macvrf-v10
    Mac table:
      - Address: '00:00:5E:00:01:01'
        Destination: irb-interface
        Dest Index: 0
        Type: irb-interface-anycast
        Active: true
        Aging: N/A
        Last Update: '2024-10-19T21:16:11.000Z'
      - Address: '00:11:01:00:00:01'
        Destination: ethernet-1/1.4096
        Dest Index: 11
        Type: learnt
        Active: true
        Aging: 271
        Last Update: '2024-10-25T01:49:52.000Z'
      - Address: '20:5E:97:B3:FA:FF'
        Destination: 'vxlan-interface:vxlan0.500 vtep:192.0.2.3 vni:10010'
        Dest Index: 7521570
        Type: evpn-static
        Active: true
        Aging: N/A
        Last Update: '2024-10-24T03:27:18.000Z'

*snip*

A:d5-leaf1# show tunnel-interface vxlan-interface bridge-table unicast-destinations destination |
as yaml
---
vxlan-tunnel:
  - Tunnel Interface: '*'
  - VxLAN Interface: '*'
Destinations:
  - VTEP Address: 192.0.2.1
  - Egress VNI: 10010
    Destination-index: 7521598
    Number MACs (Active/Failed): 1(1/0)
  - VTEP Address: 192.0.2.2
  - Egress VNI: 10010
    Destination-index: 7521569
    Number MACs (Active/Failed): 1(1/0)
  - VTEP Address: 192.0.2.3
  - Egress VNI: 10010
    Destination-index: 7521570
    Number MACs (Active/Failed): 1(1/0)

*snip*
```

Example 42. Bridge table per MAC VRF

7.1.5 Route validation in default network-instance and IP VRFs

The command below shows routes in the default network-instance. The default network-instance can be changed to user defined network instances based on VRFs in use.

```
A:d5-leaf1# show network-instance default route-table ipv4-unicast route 192.0.2.1 | as yaml
---
   instance:
  - Name: default
    ip route:
```

```
      - Prefix: 192.0.2.1/32
      - ID: 0
      - Route Type: bgp
      - Route Owner: bgp_mgr
      - Active: True
      - Origin Network Instance: default
        Metric: 0
        Pref: 170
        Next-hop (Type): 'fe80::22de:1eff:fea4:524 (direct)\nfe80::429b:21ff:fed8:83f0 (direct)'
        Next-hop Interface: ethernet-1/30.0 \nethernet-1/29.0
        Backup Next-hop (Type):
        Backup Next-hop Interface:
```

Example 43. Route validation

7.2 EDA validation

The validation steps shown below are Kubernetes CLI based EDA validations; the UI workflow in the subsequent sections will show the UI validations as well.

Note: All the resources within EDA and Kubernetes exist within a specific namespace; thus, while accessing the resources, either a -n <namespace name> or -A for all namespaces must be mentioned.

7.2.1 Onboarding validation

The output below shows the first phase validation in EDA fabric onboarding after the manifests have been applied. The expected state is that the nodes are in DHCP acknowledged and are ready to communicate via port 57400.

```
:~$ kubectl get targetnodes -A
NAMESPACE   NAME      NODESECURITYPROFILE   STATUS   DHCP           ADDRESS        PORT
eda         d3-leaf5  managed-tls           Ready    Acknowledged   192.168.70.8   57400
eda         d3-leaf6  managed-tls           Ready    Acknowledged   192.168.70.9   57400
eda         d4-leaf3  managed-tls           Ready    Acknowledged   192.168.70.6   57400
eda         d4-leaf4  managed-tls           Ready    Acknowledged   192.168.70.7   57400
eda         d5-leaf1  managed-tls           Ready    Acknowledged   192.168.70.4   57400
eda         d5-leaf2  managed-tls           Ready    Acknowledged   192.168.70.5   57400
eda         spine1    managed-tls           Ready    Acknowledged   192.168.70.2   57400
eda         spine2    managed-tls           Ready    Acknowledged   192.168.70.3   57400
```

Example 44. DHCP, gNMI port, and node status validation

Once the first phase is completed, EDA will deploy an NPP pod per node that will continue the onboarding process: NOS status check and sync and then orchestrate the fabric by configuring the nodes. The expected state here is ONBOARDED = true, NPP connected, and NODE synced with the correct SR Linux version.

```
:~$ kubectl get toponodes -A
NAMESPACE   NAME      PLATFORM       VERSION   OS    ONBOARDED   MODE     NPP         NODE
eda         d3-leaf5  7220 IXR-D3L   24.10.2   srl   true        normal   Connected   Synced
eda         d3-leaf6  7220 IXR-D3L   24.10.2   srl   true        normal   Connected   Synced
eda         d4-leaf3  7220 IXR-D4    24.10.2   srl   true        normal   Connected   Synced
eda         d4-leaf4  7220 IXR-D4    24.10.2   srl   true        normal   Connected   Synced
eda         d5-leaf1  7220 IXR-D5    24.10.2   srl   true        normal   Connected   Synced
eda         d5-leaf2  7220 IXR-D5    24.10.2   srl   true        normal   Connected   Synced
```

```
eda          spine1     7220 IXR-H4    24.10.2    srl    true          normal    Connected    Synced
eda          spine2     7220 IXR-H4    24.10.2    srl    true          normal    Connected    Synced
```

Example 45. OS version, ZTP onboarding status validation

Each of the resources cataloged above via the “kubectl get” command can be viewed in further detail via the “kubectl describe” command; this provides verbose information about probable failure causes as well.

```
:~$ kubectl describe targetnodes d5-leaf1 -n eda
Name:         d5-leaf1
Namespace:    eda
Labels:       eda.nokia.com/hostname=d5-leaf1
              eda.nokia.com/role=leaf
              eda.nokia.com/security-profile=managed
              eda.nokia.com/source=derived
Annotations:  <none>
API Version:  core.eda.nokia.com/v1
Kind:         TargetNode
Metadata:
  Creation Timestamp:  2025-01-07T08:58:54Z
  Generation:          2
  Resource Version:    8325623
  UID:                 e02108a2-1ccd-4fdd-981a-5b2eef587dbc
Spec:
  Address:  192.168.70.4
  dhcp4:
    Address:  192.168.70.4
    Options:
      Option:  3-Router
      Value:
        192.168.70.1
      Option:  51-IPAddressLeaseTime
      Value:
        604800
      Option:  1-SubnetMask
      Value:
        255.255.255.0
      Option:  67-BootfileName
      Value:
        http://100.116.161.50:9200/core/httpproxy/v1/asvr/eda/init-base/bootscript-d5-leaf1/d5-
leaf1-provision.py
  Operating System:  srl
  Platform:          7220 IXR-D5
  Port:              57400
  Serial Number:     NK220431218
  Version Match:     v24\.10\.2.*
  Version Path:      .system.information.version
Status:
  Bootstrap Status:         Ready
  Bootstrap Status Reason:  onboard success
  Dhcp Status:              Acknowledged
  Tls Status:
    Node Security Profile:  managed-tls
    Tls:
      Csr Params:
        Certificate Validity:  2160h0m0s
        City:                  Sunnyvale
        Country:               US
        Csr Suite:             CSRSUITE_X509_KEY_TYPE_RSA_2048_SIGNATURE_ALGORITHM_SHA_2_256
        Org:                   NI
        Org Unit:              EDA
```

```
        San:
          Dns:
            d5-leaf1
          Ips:
            192.168.70.4
        State:     California
      Issuer Ref:  eda-node-issuer
Events:            <none>
```

Example 46. Target node description

```
:~$ kubectl describe toponodes d5-leaf1 -n eda
Name:         d5-leaf1
Namespace:    eda
Labels:       eda.nokia.com/hostname=d5-leaf1
              eda.nokia.com/role=leaf
              eda.nokia.com/security-profile=managed
Annotations:  <none>
API Version:  core.eda.nokia.com/v1
Kind:         TopoNode
Metadata:
  Creation Timestamp:  2025-01-07T08:58:53Z
  Generation:          6
  Resource Version:    9033956
  UID:                 4cce0f9c-d710-4d58-af6a-b9ae00896e94
Spec:
  Node Profile:  real-srlinux-24.10.2
  Npp:
    Mode:              normal
  On Boarded:          true
  Operating System:    srl
  Platform:            7220 IXR-D5
  Serial Number:       NK220431218
  Version:             24.10.2
Status:
  Node - Details:      192.168.70.4:57400
  Node - State:        Synced
  Npp - Details:       10.244.0.42:50057
  Npp - State:         Connected
  Operating System:    srl
  Platform:            7220 IXR-D5
  Version:             24.10.2
Events:                <none>
```

Example 47. Toponode description

EDA has a unique ability to determine the operational state of various components of the fabric on the CLI, from interfaces to VLANs to VRFs. The network administrator can get the overall state of the fabric via CLI and GUI. The following examples demonstrate these validations.

```
:~$ kubectl get interfaces -A
NAMESPACE   NAME                           ENABLED   OPERATIONAL STATE   SPEED   LAST CHANGE
eda         d3-leaf5-ethernet-1-1          true      up                  100G    8d
eda         d3-leaf5-ethernet-1-10         true      up                  100G    8d
eda         d3-leaf5-ethernet-1-2          true      up                  100G    3d1h
eda         d3-leaf5-ethernet-1-5          false     down                100G    8d
eda         d3-leaf5-ethernet-1-9          true      up                  100G    8d
eda         d3-leaf6-ethernet-1-1          true      down                100G    3d1h
eda         d3-leaf6-ethernet-1-10         true      up                  100G    17d
eda         d3-leaf6-ethernet-1-5          true      up                  100G    17d
```

Example 48. Fabric wide interface(s) status

```
:~$ kubectl get vlans -A
NAMESPACE   NAME          BRIDGEDOMAIN   OPERDOWN SUBIF   TOTAL SUBIF   OPERATIONALSTATE
LASTCHANGE                AGE
eda         tagged-v20    macvrf-v20
42d
eda         tagged-v40    macvrf-v40     0                1             up                 2025-
02-12T05:08:01.000Z   42d
eda         tagged-v50    macvrf-v50     0                5             up                 2025-
02-20T11:10:22.000Z   42d
eda         tagged-v60    macvrf-v60     1                2             degraded           2025-
02-17T10:06:37.000Z   30d
eda         untagged-v10   macvrf-v10    1                8             degraded           2025-
02-12T05:27:58.000Z   42d
eda         untagged-v20   macvrf-v20    1                2             degraded           2025-
02-17T10:06:37.000Z   42d
eda         untagged-v30   macvrf-v30
42d
eda         untagged-v40   macvrf-v40
42d
eda         untagged-v50   macvrf-v50
42d
eda         untagged-v60   macvrf-v60
42d
```

Example 49. Fabric wide VLAN status

```
:~$ kubectl get bridgedomain -A
NAMESPACE   NAME         VNI     EVI   IMPORT TARGET   EXPORT TARGET   OPERDOWN SUBIF   TOTAL
SUBIF   OPERATIONALSTATE   LASTCHANGE
eda         macvrf-v10   10010   10    target:1:10     target:1:10     1                8
degraded           2025-02-12T05:27:58.000Z
eda         macvrf-v20   10020   20    target:1:20     target:1:20     1                2
degraded           2025-02-17T10:06:38.000Z
eda         macvrf-v30   10030   30    target:1:30     target:1:30     0                0
down               2025-02-06T05:22:37.000Z
eda         macvrf-v40   10040   40    target:1:40     target:1:40     0                1
up                 2025-02-12T05:08:01.000Z
eda         macvrf-v50   10050   50    target:1:50     target:1:50     0                5
up                 2025-02-20T11:10:22.000Z
eda         macvrf-v60   10060   60    target:1:60     target:1:60     1                2
degraded           2025-02-17T10:06:38.000Z
```

Example 50. Fabric wide bridge domain status

```
:~$ kubectl get irbinterfaces -A
NAMESPACE   NAME      MTU    OPERATIONALSTATE   LASTCHANGE
eda         irb-v10   9200   up                 2025-02-20T11:10:18.000Z
eda         irb-v20   9200   up                 2025-02-20T11:10:18.000Z
eda         irb-v30   9200   up                 2025-02-20T11:10:18.000Z
eda         irb-v40   9200   up                 2025-02-20T11:10:18.000Z
eda         irb-v50   9200   up                 2025-02-20T11:10:18.000Z
eda         irb-v60   9200   up                 2025-02-20T11:10:18.000Z
```

Example 51. Fabric wide IRB status

```

:~$ kubectl describe irbinterfaces irb-v10 -n eda
Name:         irb-v10
Namespace:    eda
Labels:       <none>
Annotations:  <none>
```

```
API Version:  services.eda.nokia.com/v1alpha1
Kind:         IRBInterface
Metadata:
  Creation Timestamp:  2025-01-08T12:15:31Z
  Generation:          7
  Resource Version:    9698627
  UID:                 38f1ab4f-ca15-4017-98ae-4983acbabb76
Spec:
  Arp Timeout:    250
  Bridge Domain:  macvrf-v10
  Evpn Route Advertisement Type:
    Arp Dynamic:  true
    Arp Static:   false
    Nd Dynamic:   false
    Nd Static:    false
  Host Route Populate:
    Dynamic:  false
    Evpn:     false
    Static:   false
  Ip Addresses:
    ipv4Address:
      Ip Prefix:  172.16.10.254/24
      Primary:    true
  Ip MTU:         9200
  l3ProxyARPND:
    Proxy ARP:        true
    Proxy ND:         false
  Learn Unsolicited:  BOTH
  Router:             vrf1
Status:
  Interfaces:
    Enabled:  true
    ipv4Addresses:
      Ip Prefix:        172.16.10.254/24
      Primary:          true
    Last Change:        2025-02-20T11:14:17.562Z
    Node:               d4-leaf3
    Node Interface:     irb0.4
    Operating System:   srl
    Operational State:  up
    Enabled:            true
    ipv4Addresses:
      Ip Prefix:        172.16.10.254/24
      Primary:          true
    Last Change:        2024-12-28T04:05:04.265Z
    Node:               d5-leaf2
    Node Interface:     irb0.4
    Operating System:   srl
    Operational State:  up
**snip**
```

Example 52. IRB description

```
:~$ kubectl get routers -A
NAMESPACE    NAME   VNI     EVI   IMPORT TARGET   EXPORT TARGET   OPERATIONALSTATE   LASTCHANGE
eda          vrf1   10500   500   target:1:500    target:1:500    up                 2025-02-
20T11:10:18.000Z
eda          vrf2   10501   501   target:1:501    target:1:501    up                 2025-02-
20T11:10:18.000Z
```

Example 53. VRF description and state

EDA also provides a tool called *edactl* that can be used to provide insight into the internal transactions and workflow results. The *edactl* tool can be used by accessing the *eda-toolbox* pod. See the following example for reference.

```
kubectl exec -it eda-toolbox-84c95bd8c6-ptbt4 -n eda-system – bash

root in on eda-toolbox-84c95bd8c6-ptbt4 /eda
➜  edactl transaction 580
input-crs:
    gvk: core.eda.nokia.com/v1, kind=Deviation name: spine2-
1260f6d2a707ccc7e2ed976c73113e47a3d5bfc4 action: CreateUpdate
    gvk: core.eda.nokia.com/v1, kind=Deviation name: spine2-
22ff7015d4bd52f6ba8d7d4633a00c5baef2e54b action: CreateUpdate
    gvk: core.eda.nokia.com/v1, kind=Deviation name: spine2-
856da911e3c57777a6d0dd127133005e1e1a97c2 action: CreateUpdate
    gvk: core.eda.nokia.com/v1, kind=Deviation name: spine2-
d6ec53ba28385d3daa558fa368350c58130aaeb6 action: CreateUpdate
intents-run:
nodes-with-config-changes:
general-errors:
commit-hash: 85dd1e47f78a93322745106a8e67782a1d1eb49c
execution-summary:
timestamp: 2025-02-04 06:45:03 +0000 UTC [2025-02-04T06:45:03Z] - 514h19m ago
result: OK
dry-run: false
```

Example 54. Transaction details by using edactl tool

The above tools and utilities are critical in both orchestrating and troubleshooting the fabric for before Day-0 and post Day-2 operations.

8 Automation and orchestration

8.1 Digital twin with Containerlab

Digital twins are an integral part of Day-0 through Day-2 operations, providing the operations and deployment teams with the opportunity to continuously validate the look and feel of any deployment. These virtual fabrics also grant the ability to learn and play with technologies and designs – in this case, a prescriptive 3-stage EVPN VXLAN fabric that has been validated and tuned to provide maximum efficiency and redundancy.

A digital twin of this NVD can be deployed using Containerlab and containerized SR Linux. The repository can be found here - https://github.com/nokia/nokia-validated-designs. This includes:

- An EDA-orchestrated deployment, which comprises of:
    - All manifest files required for an EDA-orchestrated digital twin of the 3-stage EVPN VXLAN NVD
    - A bash script (*deploy-3-stage-nvd.sh*) that deploys the end-to-end fabric

  - o A bash script (destroy-3-stage-nvd.sh) that destroys all resources created by the deployment script

  Note: For the EDA-orchestrated deployment, the deployment bash script assumes the respective Containerlab topology (*3-stage-nvd.clab.yaml*) is already deployed and healthy.

- A non-EDA deployment with all configuration pre-loaded for end-to-end fabric functionality.