OpenSDN - Overview of architecture and features

Dalibor Kafka
Faculty of Management Science and Informatics
University of Zilina
Zilina, Slovakia
dalibor.kafka@fri.uniza.sk

Pavel Segec
Faculty of Management Science and Informatics
University of Zilina
Zilina, Slovakia
pavel.segec@fri.uniza.sk

Marek Cunderlik
Faculty of Management Science and Informatics
University of Zilina
Zilina, Slovakia
cunderlik6@fri.uniza.sk

Marek Moravcik
Faculty of Management Science and Informatics
University of Zilina
Zilina, Slovakia
marek.moravcik@fri.uniza.sk

2025 International Conference on Emerging eLearning Technologies and Applications (ICETA) | 979-8-3315-7426-0/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICETA67772.2025.11280250

***Abstract*—OpenSDN, a fork of Tungsten Fabric, is an open-source, platform-agnostic software-defined networking (SDN) solution designed to provide scalable virtual networking across hypervisors, containers, and bare-metal servers. This paper presents an overview of OpenSDN's architecture, key features, and its integration with orchestrators such as OpenStack. We analyze the controller–vRouter model, overlay networking mechanisms, and the role of protocols including XMPP, BGP, and EVPN-VXLAN in enabling multi-tenant, secure, and efficient cloud networking. A practical case study is discussed, where OpenSDN was deployed in a single-node OpenStack environment at our university department to evaluate its viability as a private cloud solution. The paper outlines the deployment process, challenges encountered due to limited documentation and evolving support, and post-deployment configurations required to enable external connectivity. Experimental results with BGP EVPN VXLAN integration, using FRR as a routing solution, demonstrate how OpenSDN can extend cloud services beyond isolated environments. The findings highlight both the strengths and the practical hurdles of adopting OpenSDN in academic and enterprise cloud infrastructures.**

***Index Terms*—Cloud Computing, OpenSDN, SDN, Software Defined Networking, Orchestration, Hypervisor, OpenStack**

I. Introduction

OpenSDN was originally being developed under the name Contrail and OpenContrail by Juniper Networks and was later renamed to Tungsten Fabric. OpenSDN is a direct fork of Tungsten Fabric and the process of renaming it to OpenSDN hasn't yet been completed in all of its related literature and thus the name Tungsten Fabric still appears in many places in the documentation, so there might be sources within this paper that still refer to it as such. OpenSDN is a platform agnostic SDN solution which supports a wide variety of hypervisors and container solutions and even bare metal servers to provide a scalable virtual networking platform.

Originally our university department had a robust cloud-based infrastructure running on Openstack by itself, meaning there weren't any additional plugins concerning its networking features. As a part of this project we decided to implement OpenSDN together with Openstack as its orchestrator in a simplified environment running on a single baremetal server, otherwise known as single-node deployment. It was intended to test the viability of a truly SDN based private cloud solution and the features it provides for the orchestrator.

This paper will dive deeper into some of the important architecture details of OpenSDN that are relevant to our use-case, as well as document the process and method of deployment at our department, any challenges that were encountered during the deployment as well as important configuration that was necessary for the cloud solution to be viable for general use.

II. Related works

Research in Software-Defined Networking (SDN) has produced several open-source and commercial solutions aiming to enhance network programmability, flexibility, and orchestration. Early frameworks such as OpenDaylight [1] and ONOS [2] introduced modular, controller-centric architectures that allowed centralized management and dynamic policy enforcement across diverse networking environments. Projects like OpenContrail, later evolved into Tungsten Fabric [3], extended SDN capabilities into cloud orchestration platforms such as OpenStack and Kubernetes, providing integrated virtual networking and policy automation. Further advancements include Neutron ML2 [4], which supports pluggable network drivers within OpenStack, and Project Calico [5], which offers scalable, layer-3-based virtual networking and microsegmentation for containerized workloads.

In addition to these practical frameworks, several academic studies have explored SDN architectures and their integration

979-8-3315-7426-0/25/$31.00 ©2025 IEEE
Authorized licensed use limited to: University of Zilina. Downloaded on December 22,2025 at 10:35:22 UTC from IEEE Xplore. Restrictions apply.

with cloud environments. Kreutz et al. [6] provided a comprehensive survey of SDN research, defining its architectural layers, control abstractions, and applications in cloud data centers. Their findings underscore the importance of open platforms such as OpenSDN, which implement these principles in real-world cloud orchestration systems. Compared to prior research and frameworks, OpenSDN continues the evolution of Tungsten Fabric by refining its controller–vRouter model and overlay networking mechanisms, with enhanced support for BGP EVPN VXLAN integration. This combination delivers high scalability and interoperability across hybrid cloud environments, distinguishing OpenSDN as a viable open-source alternative for both research and production deployments.

III. Architectural and Functional Analysis of OpenSDN

The following analysis focuses on the internal structure and behavior of OpenSDN within cloud environments. It explores the platform's essential features, the roles of its primary components, and the mechanisms by which it interfaces with orchestrators and external systems.

A. Key features

OpenSDN implements virtual networking for cloud environments using orchestrators such as Openstack and Kubernetes [6]. It uses overlay networks between virtual routers located on each host. It includes enhanced features compared to native implementations of networking within these orchestrators [7]. These features include:

- Scalable and multi-tenant network architecture
- Centralized management of IP addresses across multiple tenants
- Use of DHCP and ARP proxies to prevent unnecessary network flooding
- Optimized edge replication
- Tenant-specific local DNS resolution
- Distributed firewall with customizable access control lists
- Security policies based on application requirements
- Load balancing distributed across multiple hosts
- Network address translation, floating IPs + SNAT
- Service chaining for virtual network functions integration
- Dual stack IPv4 and IPv6
- BGP peering with GW routers
- BGP as a Service (BGPaaS)

These characteristics reflect the general design goals of modern SDN platforms aimed at scalability, automation, and multi-tenant isolation [6]

B. Working with an orchestrator

OpenSDN consists of two major software components:

- Controller – a set of services that maintains a model of networks and network policies, the defacto centralized control plane of the entire virtual network
- vRouter – installed in each host (VMs or containers), it performs packet forwarding and enforces network and security policies.

Fig. 1 illustrates a typical deployment of a private OpenStack environment integrated with OpenSDN. It shows how the orchestrator, controller, and vRouters interact to provide virtual networking and policy enforcement across the cloud infrastructure.

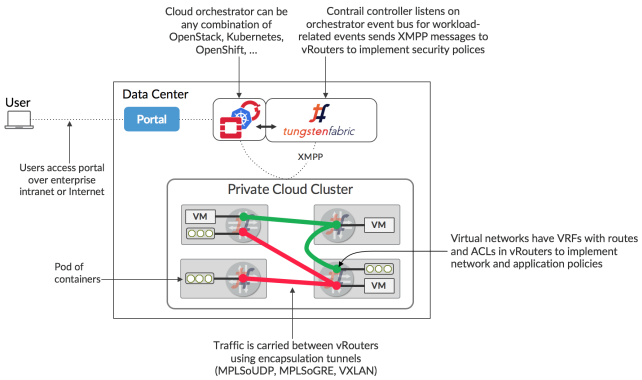


Fig. 1. Typical deployment of OpenSDN [7]

The OpenSDN controller integrates with an orchestrator through a software plugin that implements the orchestrator's networking services [8]. For example, the OpenSDN plugin for OpenStack supports the Neutron API, while the kube-network-manager and CNI (Container Network Interface) components utilize the Kubernetes k8s API to monitor network-related events [7].

On compute hosts, the OpenSDN vRouter replaces traditional networking components like Linux bridges, IP tables, or Open vSwitch [6]. The controller configures these vRouters to enforce desired networking and security policies. When packets need to travel between VMs on different hosts, they are encapsulated using MPLS over UDP/GRE or VXLAN. The IP address of the host running the destination VM is used as the destination in the outer header of the encapsulated packet. The controller installs the necessary routes in each vRouter's VRF (Virtual Routing and Forwarding) table to implement network policies. For instance, VMs within the same network can communicate freely by default, while communication across networks requires explicit policies to allow it. The communication between the controller and vRouters is handled using XMPP, a widely-used and flexible messaging protocol [7].

A crucial aspect of cloud automation is its ability to abstract the complexity of resource provisioning. Users can request resources for their applications without needing to know the technical details or the exact location of those resources. Typically, this is done through a portal that offers a catalog of service options. Users can select from these offerings, which are then translated into API calls to underlying systems,

979-8-3315-7426-0/25/$31.00 ©2025 IEEE

Authorized licensed use limited to: University of Zilina. Downloaded on December 22,2025 at 10:35:22 UTC from IEEE Xplore. Restrictions apply.

such as the cloud orchestrator, to provision virtual machines or containers with the necessary CPU, memory, and disk resources. These service offerings can range from a simple virtual machine with predefined resource allocations to an entire application stack comprising multiple pre-configured software components [7].

C. Interaction with an orchestrator

The diagram Fig. 2 illustrates how an orchestrator manages hypervisors and virtual machines (VMs), with a similar flow applicable to container orchestrators like Kubernetes. In this setup, each interface of a workload on a host connects to a Virtual Routing and Forwarding (VRF) instance, which contains Layer 2 and Layer 3 forwarding tables for the network associated with that interface's IP address. The vRouter performs Integrated Bridging and Routing (IRB), similar to physical routers, but only maintains VRFs for networks with interfaces on that host [9], including the Fabric VRF linked to the host's physical interface. VRFs enable overlapping IP and MAC addresses across different virtual networks, provided no network policies allow traffic between them. OpenSDN uses encapsulation tunneling to transport packets between VMs on different hosts, with encapsulation and decapsulation occurring between the Fabric VRF and the VM VRFs [7] [9].

When a new VM is created, the orchestrator's plugin detects the event and notifies the controller. The controller then instructs the vRouter agent to configure routes in the VRFs for the virtual networks. The agent sets up these routes in the forwarder. The process for configuring networking for a new VM with a single interface involves the following steps:

- Network Definition: Networks and policies are defined via the orchestrator or OpenSDN using a UI, CLI, or REST API. A network is essentially a pool of IP addresses for VM interfaces.
- VM Launch Request: A user requests the orchestrator to launch a VM, specifying the network for its interface.
- Host Selection: The orchestrator selects a host, and the compute agent on that host fetches the VM image and starts the VM.
- Networking Setup: The OpenSDN plugin receives instructions from the orchestrator to set up networking for the VM's interface. These instructions are converted into REST calls and sent to the OpenSDN controller.
- VRF Configuration: The controller requests the vRouter agent to connect the VM's interface to the specified virtual network. The agent instructs the vRouter forwarder to connect the interface to the VRF, creating the VRF if necessary.
- VM Startup and DHCP: The compute agent starts the VM, which typically requests an IP address via DHCP. The vRouter proxies the DHCP request, providing the IP, default gateway, and DNS server addresses.
- Route Installation: Once the interface is active, the vRouter installs routes to the VM's IP and MAC addresses, with the next hop being the VM's virtual interface.
- Label Assignment: The vRouter assigns a label to the interface and installs a label route in the MPLS table. It then sends an XMPP message to the controller with a route to the new VM, specifying the encapsulation protocol and label.
- Route Distribution: The controller distributes the route to other vRouters with VMs in the same or other networks, based on network policies.
- Policy Implementation: The controller sends routes for other VMs to the new VM's vRouter, ensuring all vRouters in the datacenter have updated routes that reflect the new VM and adhere to network policies [7].

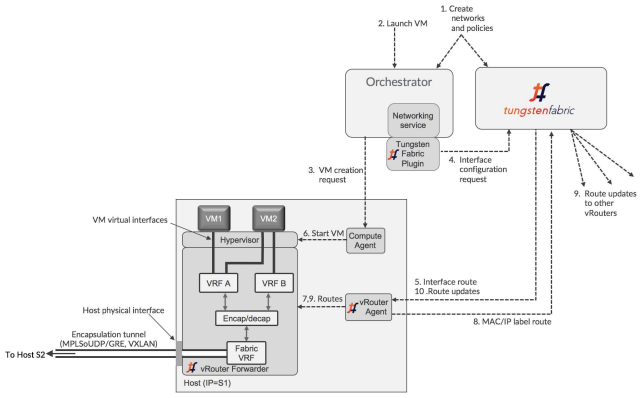


Fig. 2. Interaction with an orchestrator [7]

D. External communication

To connect the cloud infrastructure to external networks and allow communication with internet, it is necessary to set up a tunnel from VMs to a terminating device [10]. OpenSDN currently supports MPLSoUDP, MPLSoGRE and VXLAN tunneling technologies. It is also necessary to set up BGP EVPN advertisements between the terminating device and the cloud's control node. In the case of this paper we will explain specifically BGP EVPN VXLAN.

EVPN, functioning as the control plane, leverages BGP to share MAC address and IP prefix information between network devices [10], which eliminates the need for traditional network flooding methods, thus enhancing scalability and efficiency [11]. VXLAN, acting as the data plane, encapsulates Ethernet frames within IP packets, allowing Layer 2 connectivity to extend across Layer 3 routed networks. It achieves this by using VXLAN Network Identifiers (VNIs) to distinguish separate virtualized networks [12][13].

In practice, EVPN-enabled devices use BGP to advertise endpoint reachability information, such as MAC addresses and IP addresses, to other devices within the network. These devices establish VXLAN tunnels over the underlying IP infrastructure based on this information. Traffic between endpoints is then encapsulated into VXLAN packets and forwarded through

979-8-3315-7426-0/25/$31.00 ©2025 IEEE

Authorized licensed use limited to: University of Zilina. Downloaded on December 22,2025 at 10:35:22 UTC from IEEE Xplore. Restrictions apply.

these tunnels. This approach provides network scalability, multipath redundancy, workload mobility, and support for multi-tenant environments, making it ideal for modern data centers and cloud deployments [10][9][7].

BGP EVPN VXLAN uses several types of route advertisements to exchange information between network devices. The primary EVPN route types include Type 1 (Ethernet Auto-discovery), Type 2 (MAC/IP advertisement), Type 3 (Inclusive Multicast Ethernet Tag), Type 4 (Ethernet Segment Route), and Type 5 (IP Prefix Route). For the case of OpenSDN, the most important are Type 5 routes, which allow the distribution of IP prefix information across VXLAN networks, enabling Layer 3 routing between different network segments without relying on traditional routing protocols [14].

IV. OpenSDN at our department

Our use case of OpenSDN uses Openstack as its orchestrator, since the goal of the project is to create a cloud environment similar to the one already active at our department, which uses only Openstack without any advanced SDN functionalities, and deploy it , albeit at a much smaller scale in order to test the functionality and enhancements that OpenSDN brings to the Openstack cloud.

This section will be divided into a number of subsections, the first explaining our method of deployment, the second will detail some key challenges that were encountered during deployment, and the third will explain how the cloud operates in practice.

A. Deployment

Currently at our department, OpenSDN is deployed on a single bare metal server. We decided to first deploy the cloud in a single-node environment as it would be the simplest deployment for testing the functionality of the cloud while still leaving room for advancement later in the form of a multi-node deployment. Originally the baremetal server was connected to a FortiGate router as its gateway, however this later changed in order to allow external traffic. Reasons why will be explained in a later chapter. The baremental server is running on Ubuntu 22.04, as that is one of the operating system versions officially supported by the OpenSDN development team, the second being Ubuntu 20.04.

Our deployment uses Openstack as the orchestator for OpenSDN, as was necessary in order to create a simplified copy of the existing department cloud. The version of Openstack that is running on the server is Yoga, as the versions need to be specifically supported by OpenSDN and thus they do not always match the newest currently released version of Openstack. Since the deployment there have been new versions added for OpenSDN support, however redeployment was not considered at this time.

The OpenSDN cloud together with Openstack was deployed using tf-devstack package that can be found on the OpenSDN GitHub page. This install package makes use of kolla-ansible to install the necessary Openstack components and then installs the OpenSDN together with its' plugin for Neutron, which provides it with many advanced features it by itself does not supply [7].

B. Installation process

After cloning the repository, shell variables determine the orchestrator and Openstack version. In our scenario, we set these using the following commands: 'export ORCHESTRATOR=openstack' and 'export OPENSTACK_VERSION=yoga'. Subsequently, the 'run.sh' script located within the 'tf-devstack/ansible' directory is executed.

Initially, the script loads all shared libraries and functions defined in separate files ('common.sh', 'functions.sh', 'stages.sh', 'collect_logs.sh', and a custom 'functions.sh'). It automatically detects its own path and sets environmental variables such as passwords, repository paths, and service timeout durations.

The optional phase, named 'build', triggers a helper script that prepares the development environment. During this stage, dependencies like Docker, Python tools, and other required utilities for local development and deployment testing are installed.

The next phase, 'machines', manages provisioning and basic node configuration. The script identifies the distribution and installs necessary packages (Python, pip, network utilities, etc.) and executes an Ansible playbook to ready the nodes for subsequent software deployment.

Subsequently, the script proceeds with the 'openstack' phase. This stage installs Openstack using Kolla-Ansible. Depending on the selected mode, it deploys either a "vanilla" Openstack directly from the official Kolla-Ansible repository without OpenSDN integration, or it deploys a customized Openstack version designed specifically for OpenSDN purposes, available in the 'tf-kolla-ansible' repository on the OpenSDN GitHub page. We have used this second method.

The 'tf' phase is dedicated to deploying and configuring OpenSDN. The script initially checks whether the version of container images has changed, regenerating the inventory if necessary, and executes relevant Ansible scenarios to install OpenSDN, its Neutron plugin for Openstack communication, and the user interface. At the end of this phase, it outputs URLs for accessing the OpenSDN and Openstack graphical interfaces.

Following the deployment, the 'wait' phase is activated, where the script regularly monitors the status of running services to ensure all components have successfully started.

Finally, in the 'logs' phase, the script gathers configuration files and logs from all nodes, consolidating them into a centralized logging directory for debugging or auditing purposes.

979-8-3315-7426-0/25/$31.00 ©2025 IEEE

Authorized licensed use limited to: University of Zilina. Downloaded on December 22,2025 at 10:35:22 UTC from IEEE Xplore. Restrictions apply.

Upon successful installation, console outputs provide critical installation details, particularly emphasizing the IP address and port for accessing the OpenSDN user interface.

C. Challenges encountered during deployment

There were a few challenges encountered while deploying the OpenSDN cloud solution. While the architecture of OpenSDN itself has quite detailed documentation, the deployment methods themselves have been developed since the technical documentation was written and there have not been any updates to it since. This is mainly due to the fact that the fundamental architecture has not changed since the project was still being developed under the Tungsten Fabric name, back when it had a lot more support. Nowadays OpenSDN is being developed by a handful of developers who forked the original Tungsten Fabric repository.

This has caused significant problems, first of which was the mention of CentOS operating system being the only supported OS for the tf-devstack deployment package. We only learned of this being untrue after reaching out the developers themselves, where we were told that the supported OS is in fact Ubuntu 22.04 and 20.04.

The installation also requires an Chrony NTP server to be installed on the node, otherwise it will fail.

Another problem, albeit a very minor one, was the installation failing on the RabbitMQ install stage, because Ubuntu Server images have two hostname lines in the hosts file, one with the localhost name and one with the hostname of the node. This causes issues with RabbitMQ installation and can be easily circumvented by commenting out the second line and leaving only the localhost entry as seen on the figure below.

The last and the biggest challenge was encountered post-installation, and it was specifically that the VMs created within Openstack could not reach the internet even with a public IP. This was a problem that required another router to be placed between the FortiGate gateway and the node the cloud was deployed on. The details of how this problem was solved will be explained in the next chapter.

V. Post-deployment configuration

This section will explain the necessary post-deployment configuration of the network within the cloud as well as surrounding the cloud, specifically in order to enable connections outside the cloud, as that was one of the biggest challenges and deserves its own section to explain it thoroughly.

As previously mentioned in the architecture section of this paper, in order for the VMs to be able to communicate with networks outside the cloud itself, they require a BGP peer that will advertise a default route for the VMs using EVPN type 5 routes. It also requires either a VXLAN, MPLSoGRE or MPLSoUDP tunnel between the VM and the router carrying out the advertisement. The cloud itself will also advertise its own host routes (for each VM) to the BGP router in order to carry traffic back to the VM.

Since our department does not currently own any physical devices capable of EVPN, and both MPLSoGRE and MPLSoUDP are proprietary to JunOS devices, we decided to use a virtual machine with Ubuntu 22.04 OS, onto which we installed FRR.

The topology of the cloud together with the FRR router terminating the VXLAN tunnel is shown in the Fig. 3 below.

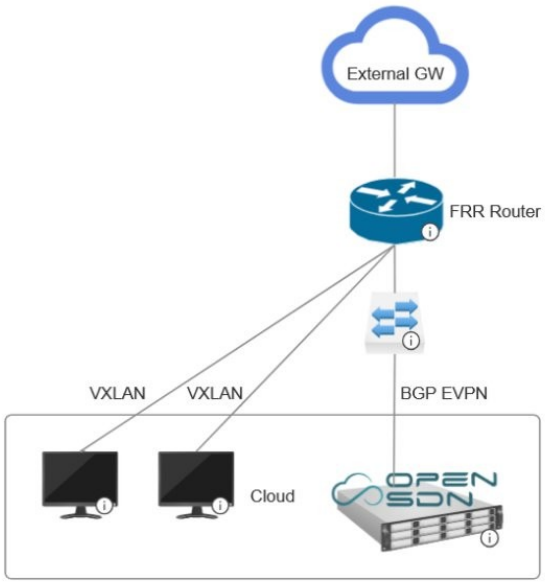


Fig. 3. OpenSDN topology diagram

FRR is an open-source networking protocol suite for Linux platforms, which enables the implementation of protocols such as BGP, OSPF, IS-IS, EVPN and more. It uses a daemon structure for the different protocols it implements, and as such a user can specifically pick and choose only the necessary components [15]. For the purposes of this project we only needed the BGPd daemon. The configuration of the peering was rather straightforward, as FRR uses VTYSH shell, which has syntax and structure very similar to Cisco IOS.

Before configuring the FRR options, it was necessary to properly configure the underlying interfaces on the router VM. The VM currently has two interfaces, one in the same subnet as the control node's interface, which directly connects to the control node and one in a different subnet, which functions as the default gateway for the EVPN to advertise. It was also necessary to configure a VRF, as the communication between the node and the terminating router runs in a non-default VRF, as well as the VXLAN interface itself, which requires its own bridge interface as is standard with VXLAN configuration [12]. Finally, the VXLAN needs a VTEP to be assigned to it, which is conventionally done with a loopback interface on the device as it is always up. The same was done in our case, however the loopback needs a /32 IP address to be added for it to be able to communicate with other devices, which was

979-8-3315-7426-0/25/$31.00 ©2025 IEEE Authorized licensed use limited to: University of Zilina. Downloaded on December 22,2025 at 10:35:22 UTC from IEEE Xplore. Restrictions apply.

configured in this case as well. The entire configuration was performed using Netplan, for its simple YAML based structure and ease of use.

The FRR configuration itself required a global BGP instance to be configured in the same autonomous system as the OpenSDN cloud and its neighbor set to the IP address of the vhost0 interface on the control node, as this is the VTEP for the cloud. It was also necessary to create a BGP instance for the VRF we created earlier. After that we needed to redistribute kernel routes in this instance otherwise the router would not advertise the static routes configured in the Netplan configuration [16]. After the configuration was completed it was also needed to set the peer to the FRR router's loopback IP address on the control node. This was done through the OpenSDN GUI, and can be seen in the Fig. 4.

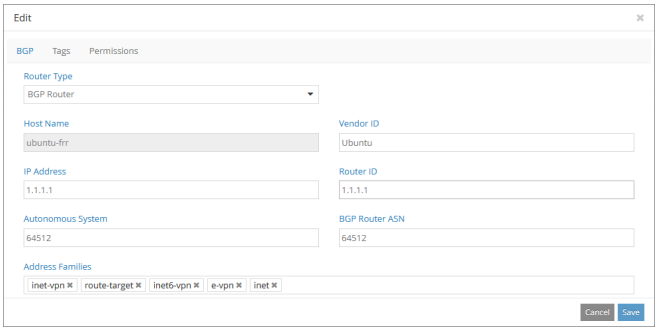


Fig. 4. BGP peer configuration in the OpenSDN GUI

Once all this was done, we could see that the routes were being advertised correctly in OpenSDN's GUI, specifically in the section routes on the control node. The host route advertised using XMPP is the VM's IP address, while the routes advertised using BGP are advertised by the FRR router, as can be seen in the Fig. 5.

In case we want to verify further, we can also check the type 5 routes advertised on the FRR router, which shows the same routes being advertised, as can be seen on Fig. 6.



Fig. 5. Advertised routes on the control node

Since the verification was successful, it would be expected that the VM should be able to ping the internet, however, this was still not the case at this point during the configuration process. Packet capture using TCPdump showed that the packets are arriving at the outgoing interface of the FRR router and therefore are being routed properly; however, the responses were not arriving back at that interface. This led us to believe that they were being routed directly to the IP address of the VM instead of being routed through the FRR router, which of course is not possible, as the packets need to go through the VXLAN tunnel to arrive back at the VM. In order to fix

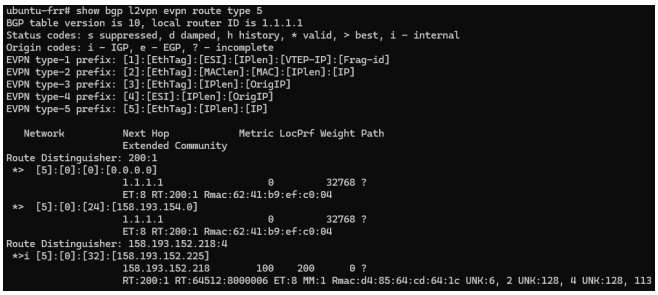


Fig. 6. BGP type 5 routes on the FRR router

this, it was simply sufficient to set a masquerade NAT on the outgoing interface of the FRR router for the IP subnet of the virtual machine interface. This way, once the packet leaves the outgoing interface towards the default gateway, it is translated to the IP of that interface, and therefore the response is also first routed to the FRR router and can then be sent back to the VM in the cloud through the VXLAN tunnel. Thanks to this, the VM is finally able to reach the internet, visible in the Fig. 7.

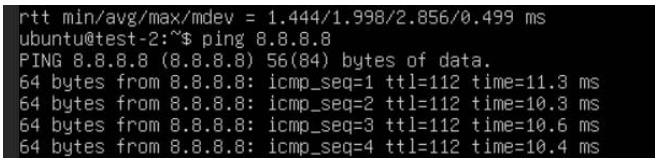


Fig. 7. Virtual machine internet ping

VI. Conclusion

This paper provided an overview of OpenSDN, an open-source Software Defined Networking (SDN) platform, and detailed its architecture, deployment methods, and features, with a particular focus on integration with the OpenStack orchestrator. It outlined the fundamental components such as the vRouter, controller, and various advanced networking capabilities that OpenSDN introduces to cloud environments.

The practical deployment at our university department highlighted the ease of implementing OpenSDN through available automation tools, despite initial challenges due to outdated documentation and minor installation issues. The key challenge was establishing connectivity between the cloud environment and external networks. This was successfully resolved by deploying an FRRouting (FRR) virtual router utilizing VXLAN and BGP EVPN to bridge cloud-internal networks with external gateways, demonstrating the interoperability and flexibility of OpenSDN solutions.

Ultimately, the integration of OpenSDN with OpenStack resulted in a robust and feature-rich private cloud solution. This deployment not only validated the viability of SDN-based networking but also enhanced existing infrastructure capabilities by leveraging OpenSDN's advanced networking

979-8-3315-7426-0/25/$31.00 ©2025 IEEE

Authorized licensed use limited to: University of Zilina. Downloaded on December 22,2025 at 10:35:22 UTC from IEEE Xplore. Restrictions apply.

features, paving the way for future expansions to multi-node deployments and further exploration of SDN technologies.

ACKNOWLEDGMENT

Funded by the EU NextGenerationEU through the Recovery and Resilience Plan for Slovakia under the project No. 09I05-03-V02-00010.

REFERENCES

[1] The Linux Foundation. *OpenDaylight Project Overview*. 2023. URL: https://www.opendaylight.org/.

[2] ON.Lab. *ONOS: Open Network Operating System*. 2022. URL: https://opennetworking.org/onos/.

[3] Juniper Networks. *OpenContrail / Tungsten Fabric Overview*. 2018. URL: https://tungstenfabric.github.io/.

[4] OpenStack Foundation. *Neutron Modular Layer 2 (ML2) Plugin Architecture*. 2024. URL: https://docs.openstack.org/neutron/latest/admin/ml2.html.

[5] Tigera Inc. *Project Calico Documentation*. 2024. URL: https://www.projectcalico.org/.

[6] Diego Kreutz et al. "Software-Defined Networking: A Comprehensive Survey". In: *Proceedings of the IEEE* 103.1 (2015), pp. 14–76. DOI: 10.1109/JPROC.2014.2371999. URL: https://doi.org/10.1109/JPROC.2014.2371999.

[7] LF Projects LLC. *Tungsten Fabric Architecture — ["Tungsten Fabric"]*. Mar. 2018. URL: https://tungstenfabric.github.io/website/Tungsten-Fabric-Architecture.html#key-features.

[8] OpenStack Foundation. *OpenStack Neutron Developer Guide*. 2023. URL: https://docs.openstack.org/neutron/latest/.

[9] Cisco Systems. *VXLAN BGP EVPN Configuration Guide*. 2022. URL: https://www.cisco.com/c/en/us/td/docs/solutions/CVD/SDN/VXLAN-BGP-EVPN.html.

[10] A. Sajassi et al. *A Network Virtualization Overlay Solution using EVPN*. Tech. rep. RFC 8365. IETF, 2018. URL: https://datatracker.ietf.org/doc/html/rfc8365.

[11] Juniper Networks Inc. *What is EVPN-VXLAN in an enterprise network? — Juniper Networks US*. URL: https://www.juniper.net/us/en/research-topics/what-is-evpn-vxlan.html.

[12] The kernel development community. *Virtual eXtensible Local Area Networking documentation*. URL: https://docs.kernel.org/networking/vxlan.html.

[13] Aadish Bahati. *Unpacking Control and Data Planes in VXLAN EVPN: A guide to scalable and secure networks*. URL: https://echostor.com/control-data-planes-vxlan-evpn/.

[14] Daniel Dib. *Introduction to VXLAN and EVPN*. URL: https://learningnetwork.cisco.com/s/blogs/a0D6e0000112Dn6EAE/introduction-to-vxlan-and-evpn.

[15] LF Projects LLC. *FRRouting Project*. URL: https://frrouting.org/.

[16] FRR. *EVPN - FRR latest documentation*. URL: https://docs.frrouting.org/en/latest/evpn.html.

979-8-3315-7426-0/25/$31.00 ©2025 IEEE Authorized licensed use limited to: University of Zilina. Downloaded on December 22,2025 at 10:35:22 UTC from IEEE Xplore. Restrictions apply.