INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Cloud and Edge Computing

João Gabriel BUTTOW ALBUQUERQUE
Fatine AZZABI

5 ISS

10/11/2025

Lab 1 : Introduction to Cloud Hypervisors

1) Similarities and differences between the main virtualisation hosts (VM and CT)

Application developer's point of view

Criterion	Virtual Machine (VM)	Container (CT)
Virtualization cost (CPU and memory)	The presence of a hypervisor introduces an additional virtualization layer between the hardware and the guest OS. This extra abstraction causes slight latency between user commands and their execution. Because each VM includes a full operating system, memory and CPU consumption are higher compared to containers.	Containers do not require a hypervisor and operate directly on the host operating system. As a result, their virtualization cost is much lower. They use hardware resources more efficiently since multiple containers can share the same OS kernel, which leads to lower overhead and better CPU efficiency.
Usage of CPU, memory, and network for a given application	The hypervisor statically allocates CPU, memory, and network resources to each VM. This ensures consistent and predictable performance, but it can lead to underutilization when resources are not fully used.	Containers dynamically share the host system's CPU, memory, and network resources. This "best-effort" model allows greater flexibility, but resource allocation is not guaranteed and may vary depending on the overall system load.
Security (access rights and resource sharing)	Virtual machines offer stronger isolation because each VM runs its own independent operating system. This separation minimizes interference between workloads and limits the impact of potential attacks.	Containers share the same OS kernel and system libraries. If a container is compromised, it can potentially affect other containers or the host. Additional security mechanisms (AppArmor, seccomp, namespaces) are

		required to ensure proper isolation.
Performance (response time)	The hypervisor introduces a small delay between requests and execution. Boot times and context switching are slower because each VM manages its own kernel and drivers.	Containers start almost instantly and have lower latency because they interact directly with the host kernel. This makes them faster for scaling, testing, and running microservices.
Tooling and continuous integration support (DevOps)	VMs are not natively integrated into DevOps or CI/CD workflows. They require heavier infrastructure management and manual updates. They are better suited for hosting full-scale application servers or long-lived services.	Containers are natively compatible with DevOps practices. Their lifecycle is automated, supporting continuous integration and deployment pipelines. Developers can easily package, test, and deploy specific applications using lightweight container images.
Backup and recovery	Virtual machines are backed up using snapshots that capture the full system state (including OS and disk). This provides reliable recovery but consumes large storage space.	Containers can be saved as lightweight images or committed snapshots. These images store only application layers and configurations, making backup and redeployment much faster and less resource-intensive.

Conclusion for both points of view

From a **developer's perspective**, containers are more suitable for rapid development, testing, and deployment.

They provide lightweight environments, faster startup, and seamless integration with continuous integration pipelines.

From a **system administrator's perspective**, virtual machines remain the best choice for hosting stable, long-term infrastructure and ensuring strong isolation between applications. They offer dedicated OS environments, predictable performance, and a higher level of security for production systems.

Overall, containers are better suited for developers, while virtual machines are better suited for administrators managing secure and persistent environments.

2) Similarities and differences between the existing CT types

This section compares container technologies based on their **isolation**, **containerization level**, and **tooling ecosystem**.

Criterion	**Linux Containers (LXC)**	**Docker (v1.10+)**	**Podman**	**Kata Containers**
Application isolation and resources	Provides OS-level system containers that can run full Linux environments. Less isolated since shared libraries and runtime are used.	Application-level isolation with single-process containers packaged with dependencies. More isolated and consistent for app deployment.	Similar to Docker but offers **rootless** containers, improving security and user-level separation.	Runs each container in a lightweight VM (micro-VM), providing hardware-assisted isolation. Strongest isolation among container runtimes.
Containerization level	**System-level:** each container behaves like a minimal virtual machine.	**Application-level:** runs only one service or microservice per container.	Same as Docker, OCI-compliant, daemonless operation.	**Application-level** virtualization but with a VM kernel per container for better isolation.

Tooling and ecosystem	Basic CLI tools (lxc-*), limited DevOps integration.	Rich CLI and APIs, strong CI/CD integration with Docker Compose and Kubernetes.	Compatible with Docker CLI, integrates with systemd and Kubernetes; secure alternative.	Integrated with Kubernetes as a runtime, less developer-oriented but designed for production-grade isolation.

Summary:

- **LXC** provides full Linux environments, suited for system-level workloads.
- **Docker** dominates for application deployment and CI/CD pipelines.
- **Podman** is a secure, daemonless alternative that supports rootless containers.
- **Kata Containers** bridge containers and VMs, ideal for multi-tenant or high-security use cases.

3) Similarities and differences between Type 1 and Type 2 hypervisors' architectures

Hypervisors are software layers that enable virtualization by allowing multiple virtual machines (VMs) to share the same physical hardware.

They are generally classified into two categories: **Type 1 (Bare-metal)** and **Type 2 (Hosted)**, depending on how they interact with the hardware and operating system.

Differences

Aspect	**Type 1 - Bare-metal (Native or Hardware-level)**	**Type 2 - Hosted (OS-level)**
Architecture	Works directly on the physical hardware without the need for a host operating system. The hypervisor	Runs on top of an existing host operating system, functioning as an application. It relies on

	manages hardware resources such as CPU, memory, and storage directly.	the host OS to access hardware resources.
Resource access	Direct access to hardware resources managed by the hypervisor, resulting in low latency and efficient resource allocation.	Indirect access to hardware, as operations must pass through the host operating system, which introduces additional latency.
Performance and efficiency	Higher performance and scalability because there is no intermediary OS layer. Commonly used in enterprise or production environments where performance and isolation are critical.	Lower performance due to the extra software layer. More suitable for development, testing, or desktop use.
Security	More secure, since it has a smaller attack surface and isolates VMs at the hardware level.	Slightly less secure because it depends on the host OS, which increases the potential attack surface.
Examples	VMware ESXi, Xen, KVM (used by OpenStack).	VirtualBox, VMware Workstation, Parallels Desktop.

Similarities

Both hypervisors allow the execution of multiple virtual machines **independently** on the same physical hardware.

They ensure **strong isolation** between virtual environments, preventing interference between applications or workloads.

Each type abstracts hardware resources and provides virtualization services to guest operating systems.

Examples and clarification

Oracle VirtualBox is a so-called hosted hypervisor, referred to as a **Type 2 hypervisor**, because it requires an existing operating system to be installed [].

It is widely used for testing, development, and educational environments due to its simplicity and user-friendly interface.

OpenStack doesn't have a hypervisor type, because it is **not a hypervisor**.
It is rather an open-source cloud computing platform that provides **Infrastructure as a Service (IaaS)**.
OpenStack allows the management of extensive compute, storage, and networking resources across a data center, accessible either through a web-based dashboard or the **OpenStack API**.
It orchestrates virtualization through underlying hypervisors such as **KVM**, which is an example of a Type 1 hypervisor [].

Practical part (Objectives 4 to 7)

1. Objectives 4 & 5 : Practical Work with VirtualBox and Docker

The purpose of these objectives was to put into practice the concepts of virtualization through **VirtualBox** (a Type 2 hypervisor) and **Docker** (container-based virtualization).
The goal was to compare and understand how network communication, resource management, and virtualization layering behave between a traditional virtual machine (VM) and containers.

Part 1 : VM Creation and Network Configuration (NAT Mode)

We used **Oracle VirtualBox**, which is a Type 2 (hosted) hypervisor, to create and configure an **Ubuntu virtual machine** operating in **NAT (Network Address Translation)** mode.
This configuration allows the VM to access the external network through the host, while keeping the VM unreachable from other physical or virtual devices on the network.

When the VM was launched, VirtualBox automatically assigned it a private IP address in the NAT subnet.
The addresses observed in our environment were as follows:

Host IP address : 10.1.5.85

VM IP address : 10.0.2.15

NAT gateway address : 192.168.56.1

To verify connectivity, we performed several ping tests.

From the VM to the Internet:

osboxes@osboxes:~$ ping 8.8.8.8

64 bytes from 8.8.8.8: icmp_seq=1 ttl=113 time=7.4 ms

64 bytes from 8.8.8.8: icmp_seq=2 ttl=113 time=8.1 ms

The VM successfully reached the Internet, confirming that outbound communication was working.

From the host to the VM:

ping 10.0.2.15

Destination host unreachable.

The VM was not reachable from the host, as expected.
NAT isolates the guest from incoming traffic, which means no direct access to the VM from the host machine or external network.

We could still observe that the VM could **reach other networked devices** and "see" the outside world,
but **other machines could not reach it**, which confirmed the default NAT behavior.

Part 2 : Enabling Host-to-VM Connectivity (Port Forwarding)

To enable external access to the VM, we configured a **port forwarding rule** within the VirtualBox NAT network.
This technique allows external connections targeting a specific port on the host to be redirected to a specific port on the VM (e.g., SSH port 22).

Parameter	**Value**
Protocol	TCP
Host IP	10.1.5.85
Host Port	1234
Guest IP	10.0.2.15
Guest Port	22

This setup means that connecting to the host machine's address on port **1234** would transparently forward the traffic to the VM's SSH port.

We then used **PuTTY** (on Windows) to connect to the VM remotely:

Host: 10.1.5.85

Port: 1234

Protocol: SSH

The SSH connection succeeded, proving that the port forwarding mechanism worked properly.
We were able to access the VM terminal and execute commands as if we were connected directly to it.

This configuration demonstrates that **VirtualBox NAT mode combined with port forwarding** allows external devices to reach specific services inside the VM, while maintaining the isolation benefits of NAT.

Part 3 : VM Duplication

Once connectivity was established, we tested **VM duplication** using VirtualBox's built-in "Clone" feature.
The process consisted of creating a **full clone** of the VM, preserving both the virtual disk and configuration.

After duplication:

- The new VM booted successfully with the same software and environment.
- A new MAC address was assigned automatically.
- The IP address changed, as expected, since NAT assigns addresses dynamically.

This demonstrated that **VMs are self-contained environments** that can be replicated easily, but at the cost of storage space and time (since each VM includes a full operating system image).

Part 4 : Docker Installation on the VM

Next, we installed **Docker Engine** on the Ubuntu VM to explore container-based virtualization.
Docker was installed and enabled as a system service:

osboxes@osboxes:~$ sudo apt update

osboxes@osboxes:~$ sudo apt install docker.io -y

osboxes@osboxes:~$ sudo systemctl enable docker

osboxes@osboxes:~$ sudo systemctl start docker

osboxes@osboxes:~$ sudo systemctl status docker

● docker.service - Docker Application Container Engine

Active: active (running)

Once installed, we verified Docker's version and pulled a base image:

$ docker --version

Docker version 24.0.5, build 12345

$ sudo docker pull ubuntu

The **Ubuntu image** was downloaded and stored locally, ready to be used for creating containers.

Part 5 : Container Provisioning and Network Connectivity

We then created our first container, named **CT1**, from the Ubuntu image:

$ sudo docker run --name ct1 -it ubuntu

Once inside the container, we installed net-tools and tested network connectivity:

From the container to the Internet:

root@ct1:/# ping 8.8.8.8

64 bytes from 8.8.8.8: icmp_seq=1 ttl=113 time=7.40 ms

64 bytes from 8.8.8.8: icmp_seq=2 ttl=113 time=8.15 ms

The container had Internet access via the VM's network bridge.

From the container to the VM:

root@ct1:/# ping 10.0.2.15

64 bytes from 10.0.2.15: icmp_seq=1 ttl=64 time=0.083 ms

64 bytes from 10.0.2.15: icmp_seq=2 ttl=64 time=0.067 ms

Bidirectional communication between VM and container was functional.

From the VM to the Docker bridge:

osboxes@osboxes:~$ ping 172.17.0.1

64 bytes from 172.17.0.1: icmp_seq=1 ttl=64 time=0.051 ms

The VM could reach the Docker network bridge (docker0), confirming that the bridge interface was up.

These results prove that the Docker containers are fully integrated into the host VM's network stack,
and that **Docker's virtual bridge** provides an isolated yet reachable subnet for all containers.

Part 6 : Image Creation and Snapshot Persistence

We created a second container (**CT2**) to test image persistence. Inside CT2, we installed the nano text editor:

root@ct2:/# apt update -y

root@ct2:/# apt install nano -y

We then committed this modified container to a new image named containers:seancetp:

$ sudo docker commit <CT2_ID> containers:seancetp

Listing all available images confirmed that the new image was successfully created:

osboxes@osboxes:~$ sudo docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

containers seancetp 74f6bace77a6 2 minutes ago 134MB

ubuntu latest 6d79abd4c962 3 weeks ago 78.1MB

We launched a third container (**CT3**) from the new image:

$ sudo docker run --name ct3 -it containers:seancetp

root@0b156b00075a:/# nano --version

GNU nano, version 7.2

Nano remained installed, which confirms that Docker correctly saved the container's state as a reusable image layer.
This approach allows rapid redeployment of customized environments.

Finally, we automated this process using a **Dockerfile**, which defines how to build an image from scratch:

```
FROM ubuntu

RUN apt update -y

RUN apt install -y nano

CMD ["/bin/bash"]
```

Building the image:

```
$ sudo docker build -t containers:seancetp -f myDocker.dockerfile .
```

This image now serves as a **template** that can instantly spawn identical containers.

Part 7 : Summary and Analysis

Through these objectives, we demonstrated how two virtualization technologies can coexist and complement each other:

Aspect	**Virtual Machine (VirtualBox)**	**Container (Docker)**
Virtualization level	Hardware/OS level	Application/process level
Performance	Heavier, slower startup	Lightweight, instant startup
Networking	NAT mode (host isolated)	Shared host kernel bridge (docker0)
Accessibility	Needs port forwarding	Directly connected within VM's network

Storage	Large images (full OS)	Compact layered images
Use case	Infrastructure virtualization	Application deployment

In summary:

- VirtualBox (Type 2) allowed us to **simulate a complete environment** with isolated network settings.
- Docker provided **lightweight, flexible virtualization** within that VM, reducing resource overhead.
- Both systems successfully communicated through NAT and Docker bridge networks.
- The snapshot mechanism (in VMs) and the **Docker commit** feature serve similar purposes but operate at different levels (system vs application).

This experiment highlights the complementarity between **system-level virtualization (VirtualBox)** and **container-level virtualization (Docker)** in modern cloud architectures.

Objectives 6 & 7 : Container and VM Management on OpenStack

The aim of this section was to explore **virtual machine creation and network configuration** in a cloud environment using **OpenStack**, the IaaS platform deployed at INSA.
Through these objectives, we practiced managing instances, networks, routers, and security groups while understanding how OpenStack handles multi-tenant infrastructure.

Part 1 : VM Creation and Network Configuration

We accessed the **OpenStack Horizon dashboard** via our INSA credentials (INSA domain → Project group).
From the *Instances* panel, we launched a new VM using the following parameters:

Parameter	**Value**
Image	ubuntu4CLV

Flavor	small2
Boot Source	Image (no volume creation)
Network	Initially: INSA public network (192.168.37.0/24)

The first launch failed with a “network allocation” error.
The issue was caused by the **absence of a gateway (router)** connecting the public INSA network to the tenant’s private space.
OpenStack could not assign an external IP to the instance because it had no route to the public network.

Part 2 : Private Network and Router Configuration

We deleted the failed instance and created a new **private network** named private_reseau.
This network had an address range of 5.7.12.0/24.
We launched a new instance in this private network using the same image and flavor.

To provide Internet access, we created a **router** and linked it to:

- the **private network** (as an internal interface), and
- the **INSA public network** (as the external gateway).

This configuration established connectivity between both subnets.
In the *Network Topology* view, we could see the router bridging both networks.

Part 3 : Security Group Configuration

The default security group blocks inbound traffic by default.
We modified it to allow SSH and ICMP communication:

Rule	Direction	Protocol / Port	Purpose
SSH	Ingress / Egress	TCP / 22	Remote access via SSH
ICMP	Ingress / Egress	ICMP	Ping requests (network testing)

These rules were added for both incoming and outgoing directions to ensure bidirectional connectivity.

Part 4 : Connectivity Tests

We accessed the instance’s console from Horizon and changed the keyboard layout using:

$ setxkbmap us

Then we tested network connectivity.

Ping to Internet:

$ ping 8.8.8.8
64 bytes from 8.8.8.8: icmp_seq=1 ttl=113 time=7.4 ms

The Internet connection was successful.

Ping to local host (Windows machine):

$ ping 10.1.5.85
64 bytes from 10.1.5.85: icmp_seq=1 ttl=64 time=0.54 ms

The host was reachable.

Ping from host to VM (private IP):

C:\> ping 5.7.12.222
Request timed out.

It was an expected failure, since the private IP is not routable externally.

To make the VM externally accessible, we assigned a **floating IP**.

Part 5 : Floating IP Association

From *Network → Floating IPs*, we allocated a new IP from the INSA public pool (192.168.37.0/24):

Floating IP: 192.168.37.58

We associated this IP with our instance (vmtest).
The VM now had two addresses:

- Private: 5.7.12.222
- Public (floating): 192.168.37.58

Ping test:

C:\> ping 192.168.37.58
Reply from 192.168.37.58: bytes=32 time=8ms TTL=55

Reachable through the floating IP.

SSH connection:

C:\> ssh user@192.168.37.58
Welcome to Ubuntu 20.04 LTS (GNU/Linux 5.4.0-139-generic x86_64)

SSH connection successful.

The floating IP mechanism uses NAT at the router level, enabling public access without exposing the entire private subnet.

Part 6 : Snapshot, Restore, and Resize Tests

We attempted to **resize** the running VM via the *Instances* tab, but encountered the following error:

> "An error has occurred. Please try again later."

After shutting down the VM and retrying, the error persisted.
This behavior is consistent with INSA's OpenStack restrictions: resizing is disabled to avoid resource contention.

We then created a **snapshot** of the VM:

- The snapshot appeared as a **private image** in our project.
- Disk size: 20 GB (modifiable).
- The original base image (ubuntu4CLV) was a **public** read-only image of 0 GB (reference template).

We created a second VM (vmtest2) from this snapshot using the same flavor and network.
The new instance successfully replicated the configuration, software, and settings of the original VM, confirming the snapshot's functionality.
However, the instance status turned "Error" after launch, likely due to quota limits or backend restrictions.

Part 7 : Observations and Conclusion

This part allowed us to understand the basic logic of OpenStack networking:
private networks are isolated by default, and connectivity to the public Internet is achieved only through a router and a floating IP.
Security groups play a key role in authorizing SSH and ICMP traffic.
We also verified that snapshots preserve the entire VM state, while resizing operations are restricted on the INSA OpenStack platform.

Overall, this lab provided hands-on experience with **instance deployment and network configuration** in a cloud environment,
illustrating how OpenStack manages virtualization at the infrastructure level.

Objectives 8 & 9 : Web Two-Tier Application on OpenStack

The purpose of these objectives was to deploy a **two-tier web application** composed of several **Node.js microservices** on the OpenStack cloud platform.
This work allowed us to understand how cloud infrastructure supports distributed applications through virtual networking and service orchestration.

Part 1 : OpenStack Client Installation

OpenStack provides a **command-line client** that allows interaction with the cloud through REST APIs.
To begin, we installed the OpenStack client **inside our Ubuntu VM running on VirtualBox**.

```
$ sudo apt update
```

```
$ sudo apt install python3-openstackclient
```

After the installation, we downloaded the **OpenStack RC file** from the *Horizon Dashboard* (tab *API Access*).
This file contains the authentication credentials (username, project ID, domain, and token).
We placed it in our /home directory and sourced it to load the environment variables:

```
$ source REOC-5-openrc.sh
```

Once sourced, we verified that the authentication was successful by executing:

```
$ openstack
```

The command displayed the list of available modules, confirming that the client was correctly configured.
From this point, we could control our OpenStack resources directly from the terminal instead of relying only on the web interface.

Part 2 : Web Application Deployment (Topology and Specification)

We deployed a **two-tier calculator web application** on OpenStack.
The application consists of multiple **Node.js microservices**: four services performing basic arithmetic operations (addition, subtraction, multiplication, division), one **CalculatorService**

managing the logic and requests, and one **front-end** component serving as the interface for clients.

Each service was hosted in a separate VM with the following configuration:

Parameter	Value
Image	ubuntu4CLV
Flavor	small2
Network	private_reseau
Floating IP	Allocated for each VM

Setup and Configuration

For each VM, we connected via SSH:

$ ssh user@<floating_IP>

Then we installed the required packages and runtime environment:

$ sudo apt update

$ sudo apt install nodejs

$ sudo apt install npm

$ sudo apt install curl

$ npm install sync-request

After preparing the environment, we downloaded the source files for each microservice using wget or curl from the link provided by the instructor.

Running the Services

Once the services were downloaded, we started them manually using:

$ node <Service>.js

Each service listened on a specific TCP port between 50000 and 50050.
To enable external communication, we modified the CalculatorService.js file to change its listening port to **500049** instead of 80:

const PORT = process.env.PORT || 500049;

We also updated the private IP addresses of the arithmetic services within the code so that the CalculatorService could reach them internally over the private network.

To ensure communication, we added new rules to the **security group** allowing inbound and outbound TCP traffic for ports 50000–50050.

Part 3 : Application Testing

1. Internal Network Test

We first tested the application inside the private network by sending an HTTP POST request to the CalculatorService using its private IP address:

$ curl -d "(5+6)*2" -X POST http://<private_IP>:50000

The expected result was returned, proving that all the arithmetic services were reachable and that internal communication worked correctly.

2. External Network Test

We then tested external access using the CalculatorService's **floating IP**:

$ curl -d "(5+6)*2" -X POST http://192.168.37.22:500049

The application responded correctly, confirming that the floating IP and security rules were properly configured and that the service could be accessed from outside the cloud environment.

Part 4 : Live Code Modification

During testing, we modified one of the microservices (for instance, by adding a console.log statement) while it was running.
The service continued to receive requests, perform operations, and return responses, but the new code was not reflected until the process was restarted.
This behavior is consistent with Node.js's standard execution model, which does not reload code dynamically without a process manager.

Part 5 : Automation Attempt (Python Script)

We attempted to use a **Python script** to automate the deployment of our calculator application.
The goal was to create the instances, assign floating IPs, and configure network links automatically using the OpenStack API.
However, the execution of the script failed due to **authentication and permission errors**.
Despite correct credentials, the OpenStack environment at INSA restricts user-level access to some administrative API calls, preventing the automation from completing.

The issue was later confirmed by the instructor and was known to affect all student projects.

Conclusion

This part demonstrated how to deploy, configure, and interconnect several microservices in a cloud environment using OpenStack.
We successfully:

- Installed and configured the OpenStack client,
- Created multiple VMs with floating IPs,
- Deployed and tested Node.js microservices,
- Validated internal and external communication using HTTP requests.

Although the automation attempt using Python was unsuccessful due to API restrictions, the manual deployment was fully functional, illustrating the core concepts of cloud-based microservice orchestration.

Objectives 10 & 11 : Deployment of the Target Network Topology

The objective of this final part of the lab was to design and deploy a more complex **network topology** on OpenStack.

This topology included two private networks connected by routers, allowing distributed microservices to communicate between different subnets while maintaining external access through a floating IP.

We also automated part of the deployment using a **Python script** and a **JSON descriptor**.

Part 1 : Network Creation and Configuration

We began by creating a **second private network** in addition to the first one already used in the previous objectives.

Network	Address Range	Purpose
Private Network 1	5.7.12.0/24	Hosted the arithmetic services (Add, Sub, Mul, Div)
Private Network 2	5.8.12.0/24	Hosted the CalculatorService (main gateway)

A new **router** was configured to connect the two networks.

The gateway router (connected to the INSA public network) was detached from the first private network and attached to the second one to provide Internet access to the new subnet.

We then created another router to bridge both private networks.

To do so, we disconnected the **floating IP** from the CalculatorService VM, removed the old router interface, and connected the gateway to *Private Network 2*.

The CalculatorService VM was then terminated and recreated inside this new network.

After reconnecting the floating IP, Internet connectivity was restored.

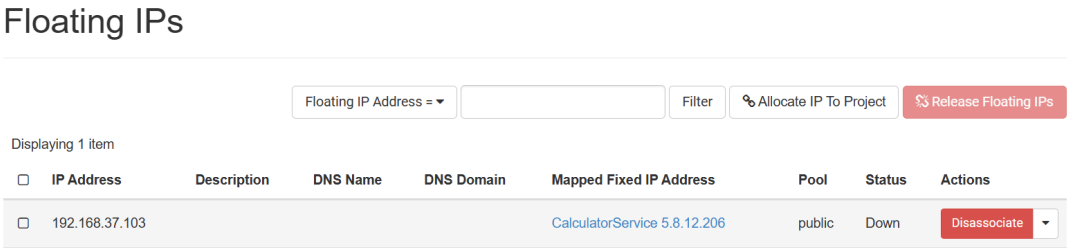
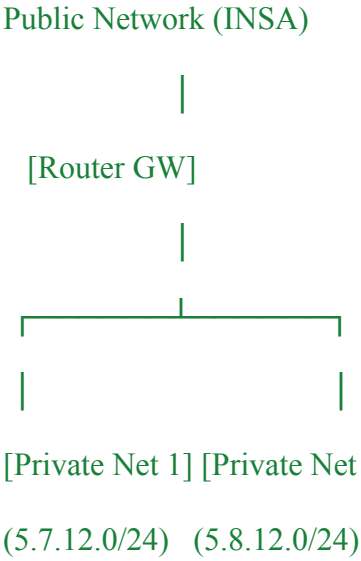


Figure 1 - Floating IP configuration for CalculatorService (192.168.37.103 assigned to 5.8.12.206)

The resulting simplified structure was as follows:



The **CalculatorService** was migrated to *Private Network 2*,
while the **operation services (Add, Sub, Mul, Div)** remained in *Private Network 1*.

Part 2 : Routing Configuration and Connectivity

At first, the VMs from the two private networks were unable to communicate.
When we tried to ping the CalculatorService from the SumService (or vice versa), we observed that the packets were being sent through the **default route** towards INSA's public network instead of directly between the private subnets.

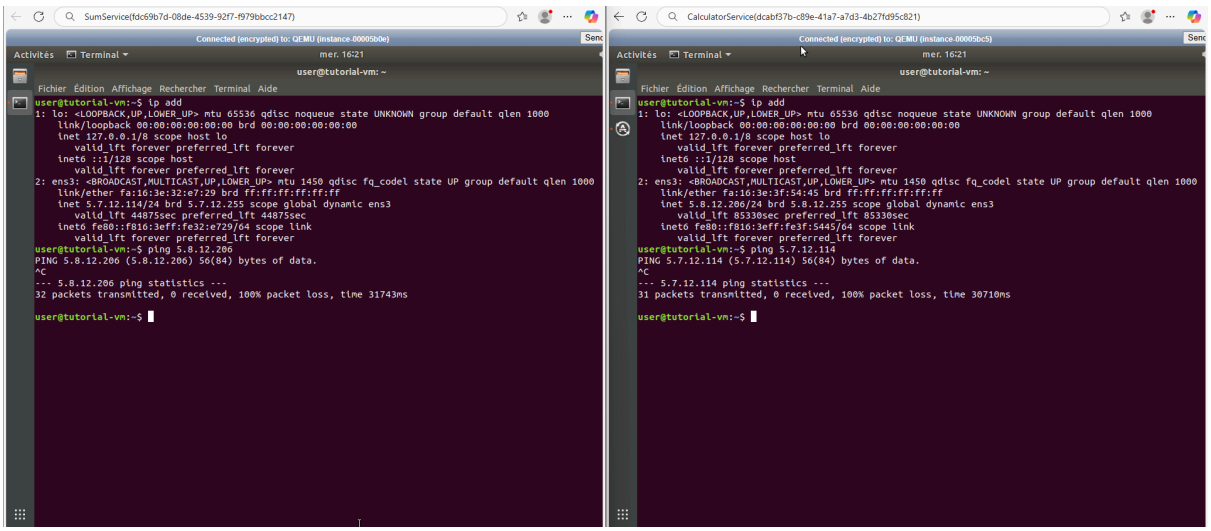


Figure 2 : Ping test failure between CalculatorService and SumService (no route between networks)

To fix this, we added a **manual static route** on the CalculatorService VM so that packets destined for Private Network 1 would go through the router's internal gateway.

Command executed:

$ sudo route add -net 5.7.12.0/24 gw 5.8.12.253

Parameter	Value
Destination network	5.7.12.0/24
Gateway	5.8.12.253 (router interface of Private Network 2)

After applying this rule, the routing table was updated, and the CalculatorService and SumService could successfully reach each other.
The ping test showed proper packet exchange between the two subnets.

Part 3 : Security Rules and Floating IP

To make the CalculatorService reachable from outside, we allocated a **floating IP (192.168.37.103)** and attached it to the VM through Horizon.
We also created new **security group rules** allowing inbound and outbound TCP traffic on port **50049**, corresponding to the HTTP service used by the Calculator.

Direction	Protocol	Port Range	Purpose
Ingress	TCP	50049	Allow HTTP requests to CalculatorService
Egress	TCP	50049	Allow outgoing responses

We confirmed network connectivity from our host machine using a ping command:

PS C:\Users\buttow-albuq> ping 192.168.37.103

Reply from 192.168.37.103: bytes=32 time=2ms TTL=62

Reply from 192.168.37.103: bytes=32 time=1ms TTL=62

Reply from 192.168.37.103: bytes=32 time=1ms TTL=62

The CalculatorService was now reachable from both inside and outside the cloud network.

Part 4 : Service Deployment and Execution

After the routing and connectivity setup, we redeployed the CalculatorService on the new VM within *Private Network 2*.
We installed the required runtime and dependencies again:

$ wget <link to CalculatorService.js>

$ sudo apt install nodejs npm curl

$ curl -sL https://deb.nodesource.com/setup_16.x | sudo -E bash -

$ sudo apt install nodejs

$ npm install sync-request

Before running the CalculatorService, all other microservices (Add, Sub, Mul, Div) were launched and set to listening mode in *Private Network 1*.

The CalculatorService was started with:

$ sudo node CalculatorService.js

In the source file, we changed the listening port to **50049** and verified the backend service addresses:

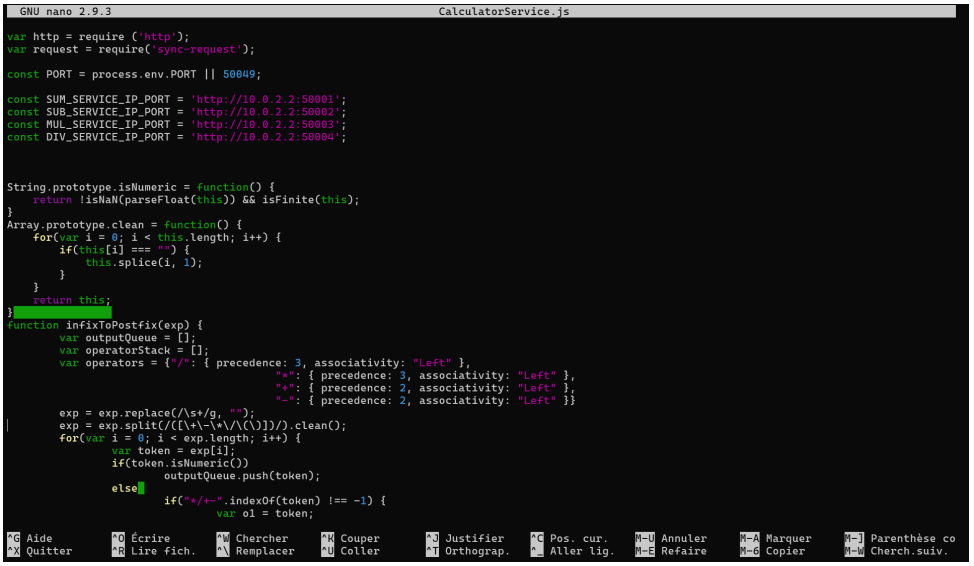


Figure 3 : Editing CalculatorService.js to configure port 50049 and backend service IPs

The service started correctly, listening on port 50049.

Part 5 : Functional Test

We first tested internal communication between the CalculatorService and one of the arithmetic services (e.g., SumService).
After adding the static route, both were able to communicate successfully.

We then tested external access using the floating IP of the CalculatorService:

```
$ curl -d "(5+6)*2" -X POST http://192.168.37.103:500049
```

The result was correctly returned, confirming that the application was functional across both networks and accessible externally.

Part 6 : Automation with Python and JSON Descriptor

To automate the deployment process, we used a **Python script** together with a **JSON descriptor file**.
The JSON file defined the topology, including the networks, routers, and virtual machines, while the Python script used the **OpenStack SDK** to deploy these resources automatically.

The source code used for this work was based on the reference repository:
https://github.com/Azefalo/Cloud-and-Edge-computing

The script was correctly written and logically structured.
It read the JSON descriptor, parsed the defined resources, and executed API calls to create the networks, routers, and instances described in the file.
However, **during execution, the script could not run successfully due to an authentication problem**.

The issue occurred when attempting to establish a connection with the OpenStack API using the provided credentials.
Although the authentication parameters in the RC file were correct, the platform returned an authorization error, preventing the Python client from initializing the deployment.
This prevented the script from compiling and executing the operations.

After investigating, the issue was confirmed to be related to **permission and authentication restrictions** in the INSA OpenStack environment.
This limitation was **acknowledged by the professor**, who confirmed that the Python code itself was correct but could not execute due to restricted API access rights on the student accounts.

Because of this, we were unable to observe the automated creation of the topology directly through the script, but the manual steps that we had already completed matched exactly what the automation was designed to reproduce.

Conclusion

This automation exercise demonstrated the principles of **Infrastructure-as-Code** applied to OpenStack cloud environments.

Although our Python script was technically correct, the execution failed due to **authentication restrictions on the INSA OpenStack platform**, which blocked the API connection.

The professor was informed and confirmed that this issue was environmental, not related to our implementation.

Despite this, the work allowed us to understand how cloud infrastructure can be defined and deployed programmatically using **Python and JSON descriptors**.

It also highlighted how authentication and permission management are crucial for automation in multi-tenant cloud environments.

Lab 2 : Orchestrating Services in a Hybrid Cloud/Edge Environment

This lab aimed to deploy a **Kubernetes cluster** on the OpenStack cloud environment and use it to orchestrate containerized services.
The goal was to understand the concepts of **container orchestration**, **pod networking**, and **service exposure** using *ClusterIP* and *NodePort* mechanisms.

Part 1 : Cluster Creation and Configuration

1. Network and Instance Setup

We first created a **private network** and connected it to the INSA public network using a router.
Then, we deployed **three virtual machines** on OpenStack:

Node	Role	Floating IP
Master	Control Plane	5.8.12.192
Worker1	Compute Node	5.8.12.193
Worker2	Compute Node	5.8.12.194

Each VM was assigned a **floating IP** and connected through SSH for configuration.

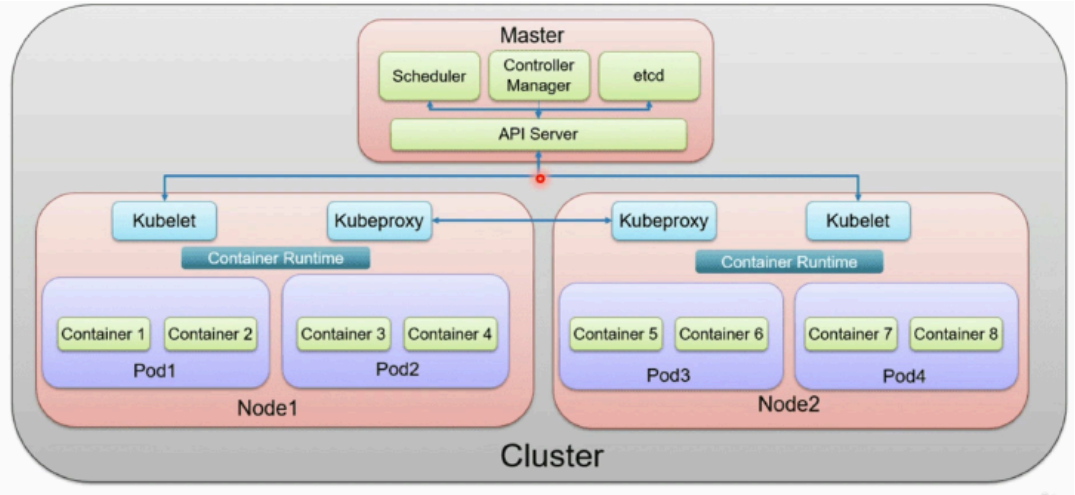


Figure 1 - Three-node topology (Master, Worker1, Worker2) deployed on OpenStack

2. Security Group Configuration

We added **ALL TCP rules** in *Ingress* and *Egress* to the security group to allow full communication between the nodes and Kubernetes components.

3. Node Preparation

We accessed each VM using SSH:

$ ssh user@<floating_IP>

For each node, we performed the following steps:

Created a new user account:

$ sudo adduser user

Disabled swap to ensure Kubernetes compatibility:

$ sudo swapoff -a

Changed the hostname to identify each node:

$ sudo hostnamectl set-hostname Master

$ sudo hostnamectl set-hostname Worker1

$ sudo hostnamectl set-hostname Worker2

Each node had a physical network interface ens3 (e.g., IP 5.8.12.192/24) used for inter-node communication.

We configured Docker to use the overlay2 storage driver and systemd as its cgroup driver:

```
$ sudo tee /etc/docker/daemon.json <<EOF
{
  "exec-opts": ["native.cgroupdriver=systemd"],
  "log-driver": "json-file",
  "log-opts": {"max-size": "100m"},
  "storage-driver": "overlay2"
}
EOF
$ sudo systemctl restart docker
```

We verified the configuration:

```
$ docker info | grep -i cgroup
```

Then, we enabled IP forwarding for pod networking:

```
$ sudo modprobe overlay
$ sudo modprobe br_netfilter
$ sudo tee /etc/sysctl.d/kubernetes.conf<<EOF
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
EOF
$ sudo sysctl --system
```

4. Kubelet Configuration

Before starting the cluster, we edited the kubelet configuration file on each node to bind it to its private IP address (ens3 interface):

- `$ sudo nano /usr/lib/systemd/system/kubelet.service.d/10-kubeadm.conf`

We added the following line:

- `--node-ip=<private_IP>`

Afterwards, we reloaded and restarted the service:

- $ sudo systemctl daemon-reload
- $ sudo systemctl restart kubelet

Part 2 : Cluster Initialization

1. Master Node Setup

On the Master node, we initialized the Kubernetes control plane:

- $ sudo kubeadm init --apiserver-advertise-address 5.8.12.192 --control-plane-endpoint 5.8.12.192

After initialization, we configured the kube client environment:

- $ mkdir -p $HOME/.kube
- $ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
- $ sudo chown $(id -u):$(id -g) $HOME/.kube/config

2. Worker Node Joining

On the Master node, we generated a token for the worker nodes:

$ kubeadm token create --print-join-command

We copied the join command and executed it as root on Worker1 and Worker2:

$ sudo kubeadm join 5.8.12.192:6443 --token <token> --discovery-token-ca-cert-hash <sha256>

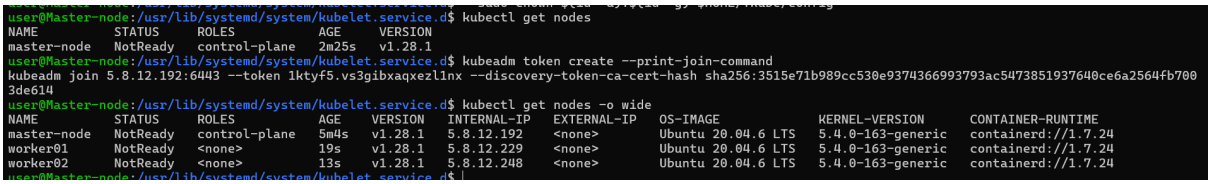


Figure 2 : Worker nodes joining the cluster successfully

3. Cluster Status

After all nodes joined, we checked the cluster status from the Master node:

- $ kubectl get nodes -o wide

Initially, the nodes appeared as **NotReady**.
This happened because **the pod network (CNI)** was not yet configured, meaning that Kubernetes could not assign IPs to pods across nodes.

To fix this, we deployed the **Calico CNI** to handle networking between pods:

```
$ kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml
```

Once Calico was applied, the nodes transitioned to the **Ready** state.

```
user@master-node:~$ kubectl get nodes -o wide
NAME          STATUS   ROLES           AGE   VERSION   INTERNAL-IP    EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION      CONTAINER-RUNTIME
master-node   Ready    control-plane   15m   v1.28.1   5.8.12.192     <none>        Ubuntu 20.04.6 LTS   5.4.0-163-generic   containerd://1.7.24
worker01      Ready    <none>          10m   v1.28.1   5.8.12.229     <none>        Ubuntu 20.04.6 LTS   5.4.0-163-generic   containerd://1.7.24
worker02      Ready    <none>          10m   v1.28.1   5.8.12.248     <none>        Ubuntu 20.04.6 LTS   5.4.0-163-generic   containerd://1.7.24
```

Figure 3 : Cluster nodes in Ready state after Calico deployment

Part 3 : Deploying ClusterIP Service

To explore service exposure inside the cluster, we deployed a simple “Hello World” web application using a **ClusterIP** service.

We cloned the service repository:

```
$ git clone https://github.com/ced-yxos/kube_service.git
```

Then, we deployed the ClusterIP service:

```
$ kubectl apply -f ./kube_service/ClusterIP
```

We listed all pods:

```
$ kubectl get pods -o wide
```

Initially, the containers were still being created, and some pods were pending scheduling on available nodes.
We observed that deployment 1 and deployment 2 shared the same name, which caused only one to appear in the pod list.
To fix this, we edited the YAML deployment files to give them distinct names:

```
$ nano app_deployment_1.yaml
```

```
$ nano app_deployment_2.yaml
```

After applying the modified deployments, we could see pods running on both Worker1 and Worker2 simultaneously.

```
user@Master-node:~$ kubectl get pods -o wide
NAME                              READY   STATUS    RESTARTS   AGE     IP              NODE       NOMINATED NODE   READINESS GATES
fastapi-app-866f4876d5-7zl5c      1/1     Running   0          5m21s   192.168.30.65   worker02   <none>           <none>
fastapi-app-866f4876d5-ln4mb      1/1     Running   0          4m35s   192.168.30.66   worker02   <none>           <none>
fastapi-app1-76f6674775-ll69k     1/1     Running   0          23s     192.168.5.9     worker01   <none>           <none>
fastapi-app1-76f6674775-skkp4     1/1     Running   0          23s     192.168.5.8     worker01   <none>           <none>
fastapi-app1-76f6674775-x92jf     1/1     Running   0          23s     192.168.5.7     worker01   <none>           <none>
fastapi-app2-866f4876d5-99s9n     1/1     Running   0          23s     192.168.30.68   worker02   <none>           <none>
fastapi-app2-866f4876d5-wvwrn     1/1     Running   0          23s     192.168.30.67   worker02   <none>           <none>
user@Master-node:~$ cat ./kube_service/ClusterIP/app_deployment_2.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fastapi-app2

spec:
  replicas: 2
  selector:
    matchLabels:
      app: fastapi-app
  template:
    metadata:
      labels:
        app: fastapi-app

    spec:
      imagePullSecrets:
        - name: repo-key
      containers:
        - name: fastapi-app-container
          image: yxos/fastapi-app
          imagePullPolicy: Always
          ports:
            - containerPort: 5001
              protocol: TCP
      nodeSelector:
          PoP: space_2
user@Master-node:~$ cat ./kube_service/ClusterIP/app_deployment_1.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fastapi-app1

spec:
  replicas: 3
  selector:
    matchLabels:
      app: fastapi-app
  template:
    metadata:
      labels:
        app: fastapi-app

    spec:
      imagePullSecrets:
        - name: repo-key
      containers:
        - name: fastapi-app-container
          image: yxos/fastapi-app
          imagePullPolicy: Always
          ports:
            - containerPort: 5000
              protocol: TCP
      nodeSelector:
          PoP: space_1
```

Figure 4 : Pods deployed across Worker1 and Worker2 (ClusterIP service)

Verification

We verified pod connectivity by executing commands inside one of the running pods:

$ kubectl exec --stdin --tty <Pod_name> -- /bin/bash

$ curl http://<ClusterIP>:<port>

The web service returned the expected “Hello World” response, confirming that inter-pod communication was functional.

Part 4 : Observations and Discussion

At this stage, the Kubernetes cluster was fully operational with:

- One master and two worker nodes successfully connected,
- Calico handling the pod network,
- A ClusterIP service providing internal communication between pods.

This part of the lab illustrated how **Kubernetes manages pods and services internally**, and how containerized applications can be distributed across multiple nodes transparently.

We also observed how modifying the deployment descriptors affects the scheduling and service visibility.
Once deployed, Kubernetes allowed us to **start, scale, and access containerized services** easily within the cluster.

Conclusion

This lab helped us understand the **fundamentals of Kubernetes orchestration** within an OpenStack environment.
We successfully:

- Built and configured a three-node cluster (1 Master, 2 Workers),
- Solved networking and CNI configuration issues,
- Deployed a simple ClusterIP web service,
- Verified internal communication between pods.

Through this setup, we gained a practical understanding of **container orchestration, pod networking, and service exposure mechanisms** within a hybrid cloud infrastructure.