

# Cloud and Edge Computing

Romain Moulin, Aude Jean-Baptiste, 5ISS - A1

**NB** : You can find all the figures and codes presented on this report on [this github repository](https://github.com/Aude510/cloud-and-edge-computing)<sup>1</sup>.

## Theoretical part

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

- The figure presents the difference between containers and VM. With VM, each VM has its own Operating System, while containers share the host OS and are therefore less isolated.

Criteria	VM	Container
<i>Virtualization cost</i>	Higher cost (hypervisor, fixed hardware resources allocation)	Lower cost (only the Docker engine - or equivalent - and container overhead)
<i>Usage of CPU, memory and network for a given application</i>	From the server POV, the VM resources are reserved and cannot be used by other processes or VM. All the VM resources not used by the application are therefore lost. Moreover, we consider here a type 2 hypervisor. It means we technically use 2 OS to run one application.	From the server POV, the application will use only what it needs on a given moment (with the container overhead). It is heavier than a single process but lighter than having one VM for a single application. It allows more flexibility with the host resources.
<i>Security</i>	The VMs do not share any OS part. They all have their own OS and libraries and from their point of view, they all have access to their "own" hardware thanks to the hypervisor virtualization. Each process inside a VM is therefore completely isolated from the processes in other VMs.	Containers are less secure than VMs. Due to the fact that the containers are not fully isolated from the host OS, they represent potential security breaches like privileges escalation. In addition to that, depending on the containerisation technique, containers can share libraries which is another potential risk as a container can poison a shared library.
<i>Performances</i>	For the VMs, the presence of a hypervisor introduces some overhead and with	A container is very short to launch as no boot is necessary. From an end

---

<sup>1</sup> <https://github.com/Aude510/cloud-and-edge-computing>

	that, some additional delay which can be annoying for the end user. In addition to that, the use of a level 2 hypervisor creates even more delay as we have to interact with 2 different OS to access the ressources. VMs are also very long to launch as it has to go through the whole booting process.	user POV, the response time is very short as the delay introduced by the containerisation technique is shorter than a VM.
<i>Tooling for the continuous integration support</i>	The support for DevOps in VM is the same as for your OS usual distribution.	Containers are usually easier to use for DevOps. For example with Docker, you can just push a new image to docker hub for users to use when you update your application. It allows you to offer your users a controlled and fixed environment.

- comparison based on different perspectives :

<b>Perspective</b>	<b>VMs</b>	<b>Containers</b>
<i>Developers</i>	With VMs, you can use Virtual Appliance to have a template with some of the libraries you need, however containers are more flexible and lighter for that. Apart from that, VMs do not present a particular advantage compared to development on a machine with a normal OS.	Developers are more concerned about the performances of an application for the end users which make containers the ideal environment to develop their applications. In addition to that, containers are light and easy to export which allows the developer to run the application with the same environment on different infrastructures.
<i>Infrastructure Administrators</i>	Infrastructure administrators are more concerned about security. That way, VMs seem like the best possibility as it is fully isolated and has less security threats than containers. In addition to that, VMs allow better management of the ressources as they are	Containers are very easy to create and less isolated than VMs. They represent a larger risk of security breach and are therefore less beneficial than VMs from an infrastructure administrator point of vue.

	statically allocated to each VM which is better for infrastructures administrators.	
--	---	--

## 2. Similarities and differences between the existing CT types

Criteria	LXC	Docker
<i>Application isolation and resources, from a multi-tenancy point of view</i>	all the containers share the liblxc library	Containers are all managed by the docker engine but they have their own libraries
<i>Containerization level (e.g. operating system, application)</i>	Lower level of isolation because shared libraries : application isolation level.	Higher level of isolation because no shared libraries : OS isolation level.
<i>Tooling (e.g. API, continuous integration, or service composition)</i>	For LXC, some <a href="#">github projects</a> exist but the tools are less developed and the documentation poor.	Docker is widely used and provides a good continuous integration support and clear <a href="#">documentation</a>
<i>portability</i>	LXC containers are tied to the system configuration so it can be complicated to migrate them on another machine	With Docker, you just need to install the Docker engine to execute any docker container on any machine so portability is much more consistent.

Source : [Conteneurs Linux et Conteneurs Docker : quelle\(s\) différence\(s\) ? | LeMagIT](#)  
[LXC vs Docker: Which Container Platform Is Right for You? - Earthly Blog](#)

	LXC	Docker
<i>Advantage</i>	Lighter and easier to implement (no software installation required)	Portability, more isolation, better documentation and CI/CD support, wider community
<i>Disadvantage</i>	Less documentation, CI/CD support and portability	Heavier than LXC with the installation of the Docker Engine and due to the isolation of the OS

### 3. Similarities and differences between Type 1 & Type 2 of hypervisors' architectures

	Type 1 (Openstack)	Type 2 (VirtualBox)
<i>Virtualization cost</i>	no host OS : the hypervisor is installed on the bare metal and all Virtual Machines are on the same level, all controlled by the hypervisor. The overhead is lighter than the Type 2 hypervisor.	the hypervisor is installed on the host OS so every virtual machine is above the host OS and the hypervisor : the overhead is higher.
<i>User Experience</i>	Faster to launch and to use because the hypervisor has access to the bare hardware.	More delay due to the number of OS it has to interact with
<i>Administrator point of vue</i>	You need a separate machine to administer your VMs.	You can install your hypervisor on your computer and administrate everything from here.
<i>Usecase</i>	designed for industrial cloud use : virtualization of servers for Infrastructure as a Service (IaaS) or private company cloud.	personal / experimental use
<i>Security</i>	If a VM is compromised, no other VM can be affected as they are fully isolated. In addition it is much harder for an attacker to gain access to the hypervisor manager than to access the OS of a machine.	Less secure because if you can access the host OS, you can access every VM.

Source : [What are hypervisors? | IBM](#)

VirtualBox is a type 2 hypervisor : it is a software that you can install on your machine (over your OS which will be the host OS) and you can then create VMs. OpenStack interacts with

type 1 hypervisor API to create VMs by allocating hardware resources (Computing, Network and Storage).

## Tasks related to objectives 4 and 5

### First part: Creating and configuring a VM

We create the VM "TP Cloud".

### Second part: Testing the VM connectivity

#### NAT mode :

*Host IP Address (ipconfig on Windows) : 10.1.5.90/16 ; gateway 10.1.0.254*

*VM IP Address (ifconfig on the VM) : 10.0.2.15/24*

*VirtualBox NAT IP address : 192.168.56.1*

All the virtual machines in the room have been attributed the same IP address. If we were to create more VMs on our host, they would all have the same IP.

#### Ping Result :

- VM to outside (ping 8.8.8.8 - DNS Google Server) : ping successful
- neighbor's host to VM (ping 10.0.2.15) : ping failed (timeout expired)
- host to VM (ping ) : ping failed (timeout expired)

The NAT allows the VM traffic to reach the internet, however the host does not have access to the VM. The IP attributed to the VM does not have a real existence and no route to it exists on the host.

The hypervisor cannot virtualize the network interface controller (NIC).

### Third part: Set up the "missing" connectivity

We forward the port 1234 of the host to the port 22 of the VM. This way, we can connect on the VM on SSH by connecting on SSH on 127.0.0.1 port 1234 on the host. We do not use the port 22 of the host to avoid conflicts if we would like to connect on SSH on the host.

### Fourth part: VM duplication

The copy made is a snapshot : everything that was done in the VM is also present in the copy.

### Fifth part: Docker containers provisioning

1. What is the Docker IP address? eth0 = 172.17.0.2/16
2. Ping an Internet resource from Docker => it works
3. Ping the VM from Docker => it works
4. Ping the Docker from the VM => it works
5. Elaborate on the obtained results

The VM creates an interface `docker0` on the same network as all the containers. This interface is the gateway for the containers to access the outside machines. The other way around, the VM uses this interface to reach the different containers.

*Do you still have nano installed on CT3?*

Yes, nano is installed on CT3 because it was instantiated with the snapshot of CT2 (which already has nano installed) as its image.

## **Expected work for objectives 6 and 7**

### **First part : CT creation and configuration on OpenStack**

DO NOT create a new volume when creating an instance.

The creation of the instance raises an error. This is due to the fact that by default, OpenStack tries to put our VM on the public network (INSA network). However, our student accounts do not have the privileges to do that. To solve that problem, we will create our own network, isolated from the INSA network. To allow the VMs to the internet, we add a router to act as a gateway between our private network and the public one.

To link a router to 2 interfaces, create a router with the outside interface associated with the public network. Then modify the router to add a new interface and associate the IP of the gateway if you specified one during the creation of the network.

After setting the network, we create a new instance. After the creation time, it becomes available and running. We can see the Image that we used for the VM, the IP address associated with it as well as the flavor and the status. It also says for how long the VM has been running.

### **Second part: Connectivity test**

*What do you think about this address?*

We choose the network `10.0.0.0/24` as our private network, with the gateway IP address `10.0.0.254`. Our VM has the address `10.0.0.109`, which is in this network. It is not the first address available, which is coherent because we use DHCP.

*Ping Result :*

- VM to desktop : => it works
- desktop to VM : => it initially does not work as the VM is on a private network so it is not reachable from the desktop POV. Then, we created and associated a floating IP to the VM which makes it available from the desktop. The floating IP is an IP from the INSA public network which is associated with the VM on the private network.

*SSH :*

We can connect in SSH to the Vm with `ssh user@floating_ip`. In our case the floating IP is `192.168.37.163`. We notice that the connection with SSH is much more enjoyable as it reacts

faster and is smoother than the console, which is logical because it is lighter.

### **Third part: Snapshot, restore and resize a VM**

If we try to resize a running instance, OpenStack raises an error. The same error is raised if the instance is shut down. This is due to a permission problem with the CSN, as resizing is very resource-consuming, but we should be able to resize a VM with OpenStack, to add or remove resources from it, even if it is running.

The difference between the snapshot and the original image is just all the changes we made between the launch of the instance and the creation of the snapshot. It copies the full state of the machine at the time it is made. In our case, we just changed the keyboard's language.

## **Expected work for objectives 8 and 9**

### **Part two : Web 2-tier application topology and specification**

**NB :** The CalculatorService listens on port 80. This port is one of the privileged ports : it needs root access to be binded, so we launch the microservice in *sudo*.

### **Part three: Deploy the Calculator application on OpenStack**

To install sync-request, run

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

To install a previous version of NodeJS. Do not forget to install npm, Then execute  
*\$ npm install sync-request*

We change the Calculator Service port to 50000 and the IPs of the operations' (+,-,\*,/) services in the Calculator service source code.

To be able to access the service from outside the OpenStack private network we created on the previous lab, we need open ports on the public network (the range 50000 - 50050 has been opened by the CSN). We then need to open the port 50000 in the Calculator Service VM in output and input. By default in OpenStack rules, every port is opened in output. We add a rule to open the port 50000 in input.

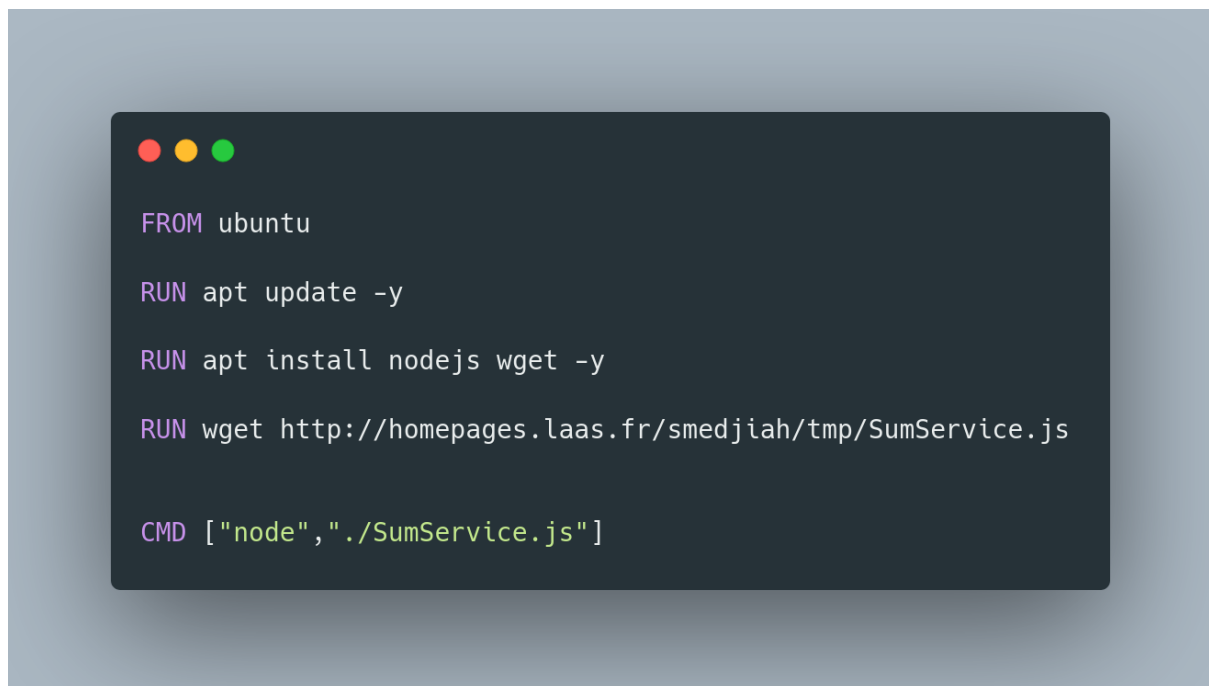
If we modify the source code of the service while it is running, nothing changes. However if we restart it the modifications are taken into account. This allows the service owner to deploy new features for its service by limiting the time where the service is down to the time to restart the service. The service is still running in its old version while we are making the changes.

#### Part four: Automate/Orchestrate the application deployment (optional)

We choose to make 5 dockerfile with the appropriate commands to be run on each container. We orchestrate the deployment with a docker compose file in which we also create a docker network to assign static IP addresses to each container.

**NB :** we get the source code of each service via a *wget* for every container but the calculator service. Indeed, this service needs to be modified with the IP addresses of the four other services. To address this problem, we get the file on the VM, modify it with the static IP addresses of the other containers, and put the modified file in the calculator container via a volume.


Figure 1 is the dockerfile for the sum service. The dockerfiles for the subtraction, division and multiplication are the same except the *wget* file. We could have chosen a more clever implementation with one docker file for the generic commands, and passing the file to get as an argument.



*Figure 1 : Sum service dockerfile*

Figure 2 is the dockerfile for the calculator service.





```
FROM ubuntu

WORKDIR /usr/app

RUN apt update -y

RUN apt install nodejs npm -y

RUN npm install sync-request

CMD ["node", "./CalculatorService.js"]
```

*Figure 2 : Calculator service dockerfile*

You can find in figure 3 the docker-compose file we used :

```

services:
  calculator-service:
    container_name: "calculator-service"
    build: "/home/osboxes/Desktop/calculator_service/"
    depends_on:
      - sum-service
      - sub-service
      - mul-service
      - div-service
    networks:
      public_net:
        ipv4_address: 192.168.1.6
    ports:
      - "50000:50000" # port binded to be accessed from outside the docker network
    volumes: # CalculatorService.js modified with the micro services ip addresses
      - ./CalculatorService.js:/usr/app/CalculatorService.js

  sum-service:
    container_name: "sum-service"
    build: "/home/osboxes/Desktop/sum_service/"
    networks:
      public_net:
        ipv4_address: 192.168.1.2
    expose:
      - 50001

  sub-service:
    container_name: "sub-service"
    build: "/home/osboxes/Desktop/sub_service/"
    networks:
      public_net:
        ipv4_address: 192.168.1.5
    expose:
      - 50002

  mul-service:
    container_name: "mul-service"
    build: "/home/osboxes/Desktop/mul_service/"
    networks:
      public_net:
        ipv4_address: 192.168.1.3
    expose:
      - 50003

  div-service:
    container_name: "div-service"
    build: "/home/osboxes/Desktop/div_service/"
    networks:
      public_net:
        ipv4_address: 192.168.1.4
    expose:
      - 50004

networks:
  public_net:
    driver: bridge
    name: "SC>>SI"
    ipam:
      config:
        - subnet: 192.168.1.0/24
    driver_opts:
      com.docker.network.bridge.name: "SC-interface" # VM interface on the docker network

```

Figure 3 : docker-compose file

## Expected work for objectives 10 and 11

### Part one: The client requirements and target network topology

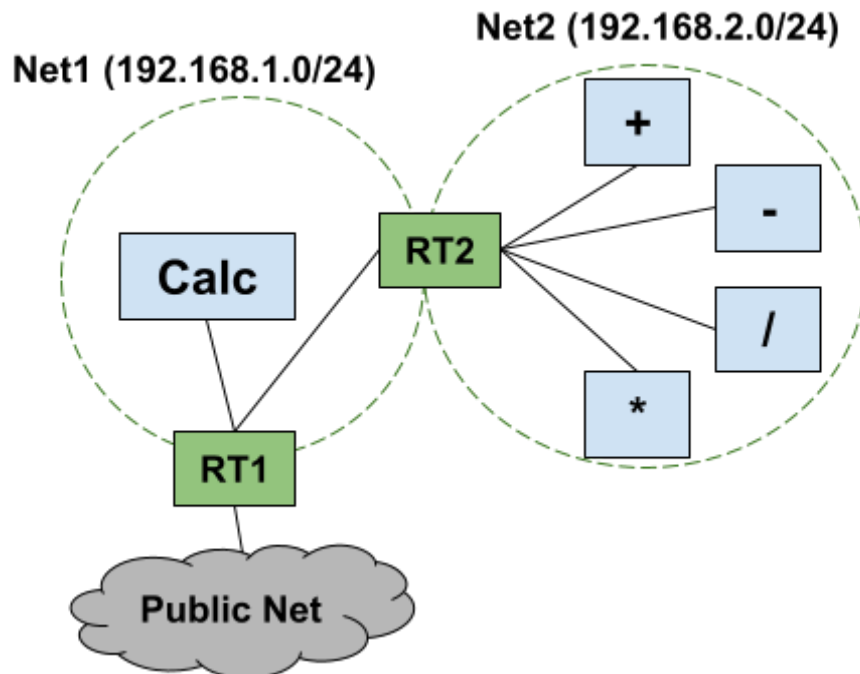


Figure 4 : Target Network Topology (cf. [this lab subject](#))

### Part two: Deployment of the target topology

We created a VM with the Ubuntu4CLV image. We installed on this VM all the necessary software for all the components of the service :

- nodejs
- npm
- syncrequest
- curl

We also got the 5 files of the five microservices. We changed in CalculatorService.js the IP addresses of the sum, subtraction, multiplication, division. We choose to allocate fixed IP addresses to these services in the rest of the deployment.

Then we added the command `node <service-name.js>` in the `.bashrc` to execute the service when you open a terminal on the VM . We made a snapshot for each service.

We are conscient that we could have made 5 different snapshots with only the requested software and files. This would have been better from a security point of vue. However in the context of this lab, we made the choice of installing everything on one VM and only changing the service name to execute because it was faster.

To deploy the requested architecture, we use the openstack client included in a bash script.

You can find this script in figure 5.

```

#!/bin/bash

##### env variables openstack #####

export OS_AUTH_URL=https://os-api-ext.insa-toulouse.fr:5000/v3
# With the addition of Keystone we have standardized on the term **project**
# as the entity that owns the resources.
export OS_PROJECT_ID=66bc14117fcd43d5a8fec9fc68ceae4a
export OS_PROJECT_NAME="5ISS-Virt-1-1"
export OS_USER_DOMAIN_NAME="insa"
if [ -z "$OS_USER_DOMAIN_NAME" ]; then unset OS_USER_DOMAIN_NAME; fi
export OS_PROJECT_DOMAIN_ID="ef8de9847762471f9f8cea12458550d2"
if [ -z "$OS_PROJECT_DOMAIN_ID" ]; then unset OS_PROJECT_DOMAIN_ID; fi
# unset v2.0 items in case set
unset OS_TENANT_ID
unset OS_TENANT_NAME
# In addition to the owning entity (tenant), OpenStack stores the entity
# performing the action as the **user**.
echo "Please enter your OpenStack login for project $OS_PROJECT_NAME"
read -r OS_USERNAME
export OS_USERNAME=$OS_USERNAME

# With Keystone you pass the keystone password.
echo "Please enter your OpenStack Password for project $OS_PROJECT_NAME as user $OS_USERNAME: "
read -sr OS_PASSWORD_INPUT
export OS_PASSWORD=$OS_PASSWORD_INPUT
# If your configuration has multiple regions, we set that information here.
# OS_REGION_NAME is optional and only valid in certain environments.
export OS_REGION_NAME="RegionINSA"
# Don't leave a blank variable, unset it if it was empty
if [ -z "$OS_REGION_NAME" ]; then unset OS_REGION_NAME; fi
export OS_INTERFACE=public
export OS_IDENTITY_API_VERSION=3
#####

echo "deployment of calculator service with openstack client"

echo "Creation of Net1 & Net2"
openstack network create Net1
openstack network create Net2

echo "Creation of subnet1 and subnet2"
openstack subnet create --network Net1 --subnet-range "192.168.1.0/24" subnet1
openstack subnet create --network Net2 --subnet-range "192.168.2.0/24" subnet2

echo "Creation of RT1 and RT2"
openstack router create RT1
openstack router create RT2

echo "Creation of the different ports"
openstack port create --network Net1 PortNet1
openstack port create --network Net2 PortNet2
openstack port create --network Net2 --fixed-ip subnet=subnet2,ip-address="192.168.2.1" PortSum
openstack port create --network Net2 --fixed-ip subnet=subnet2,ip-address="192.168.2.2" PortSub
openstack port create --network Net2 --fixed-ip subnet=subnet2,ip-address="192.168.2.3" PortMul
openstack port create --network Net2 --fixed-ip subnet=subnet2,ip-address="192.168.2.4" PortDiv
openstack port create --network Net1 PortCs

echo "Creation of the security group"
openstack security group create "Calculator Service"

echo "Creation of the security group rule"
openstack security group rule create --ingress --dst-port 50000 --protocol "tcp" "Calculator Service"

echo "Creation of the floating ip"
openstack floating ip create --port PortCs public

echo "Binding the routers interfaces"
openstack router set --external-gateway public RT1
openstack router add subnet RT2 subnet1

openstack router add port RT1 PortNet1
openstack router add port RT2 PortNet2

echo "Creating the VMS"
openstack server create --image "sum service" --flavor small2 --port PortSum "Sum Service VM"
openstack server create --image "sub service" --flavor small2 --port PortSub "Sub Service VM"
openstack server create --image "mul service" --flavor small2 --port PortMul "Mul Service VM"
openstack server create --image "div service" --flavor small2 --port PortDiv "Div Service VM"
openstack server create --image "calculator service" --flavor small2 --port PortCs --security-group
"Calculator Service" "Calculator Service VM"

```

Figure 5 : deployment script

Figure 6 presents our final topology.

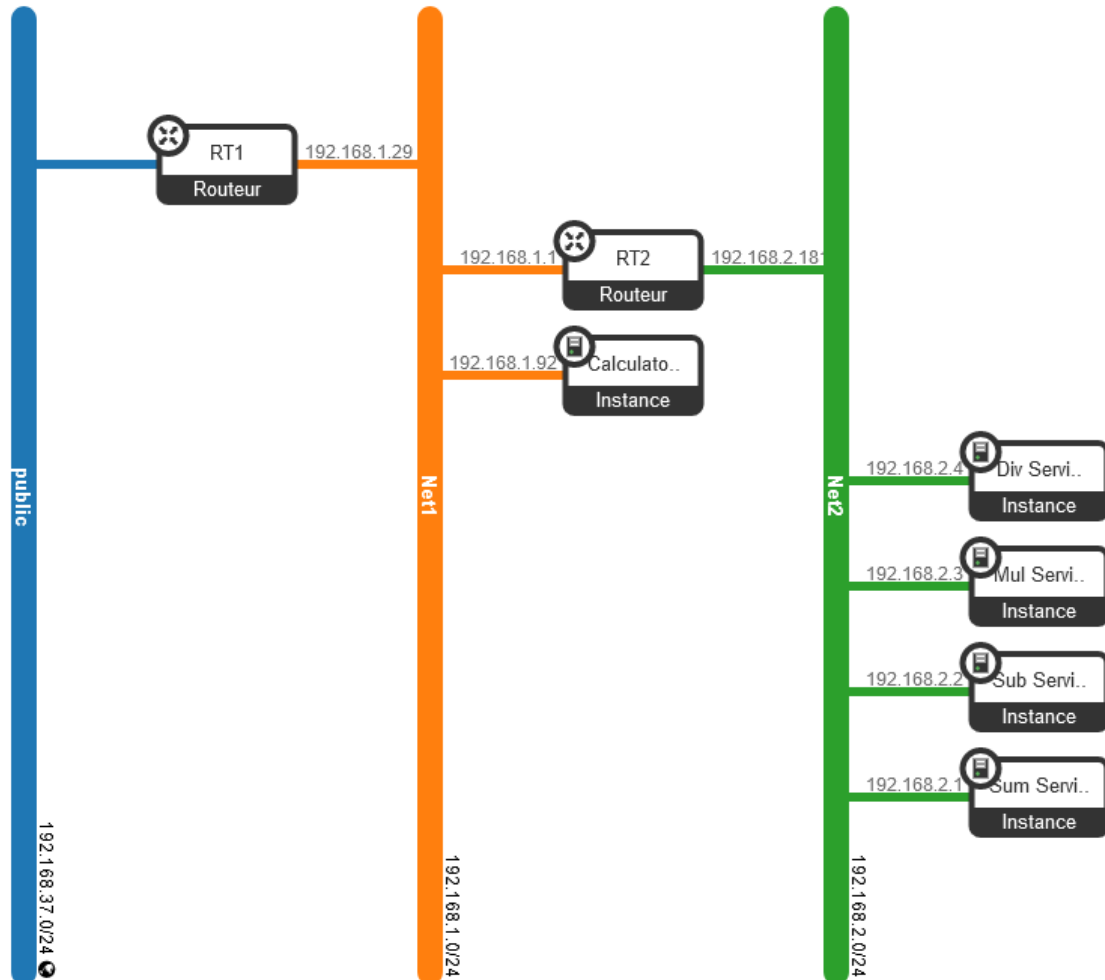


Figure 6 : screenshot of our final topology

### Part three: Configuration of the topology and execution of the services

The VM hosting the Calculator front end does not have the second network (192.168.2.0/24) in its routing table.

We need to add a route through this network to establish connectivity.

**NB :** We normally do not need to add a route to the VM hosting the sum, subtraction etc, because they have a default route towards RT2 that knows the network 1.

**NB 2 :** This is not needed here, but if we wanted to give the machines on Net2 an internet

connection, we would have to add a default route in RT2 towards RT1.

## Orchestrating services in hybrid cloud/edge environment

### Part 1: Cloud infrastructure setup

*Check the status of your cluster by running the following command:*

```
$ kubectl get nodes -o wide
```

*What do you notice?*

All the nodes are in “NotReady” status.

*After all the resources are deployed, run:*

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

*What do you notice?*

3 new pods have been created on the node worker1.

*copy one of them and try to curl it from your master node:*

```
$ curl http://endpoint_of_your_choice
```

*What do you notice?*

The endpoint displays {“hello”:”world”}.

*Then access inside the pod you just created with:*

```
$ kubectl exec -it testpod -- bash
```

*rerun the curl request from this pod.*

*What do you notice?*

We can still access the service from inside the test pod. Kubernetes guarantees that each pod can see each other by default.

*from your master node, run:*

```
$ kubectl apply -f ./NodePort
```

*after all the resources are deployed, run:*

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

*What do you notice?*

We see that 3 pods are created in the node worker1

*copy the name of the deployed service and run:*

```
$ kubectl describe services name_of_service
```

*you will notice a value attributed to the Nodeport marked: value/TCP*

*Does this value figure on your service\_yaml file?.*

Yes we can see in the service yam1 file the following line:

targetPort:5000

Kubernetes is deploying our NodePort service on port 31377 of the worker1 (server).

## Part 2: Services deployment in the hybrid cloud/edge

### Part 3: Service migration in the hybrid cloud/edge environment

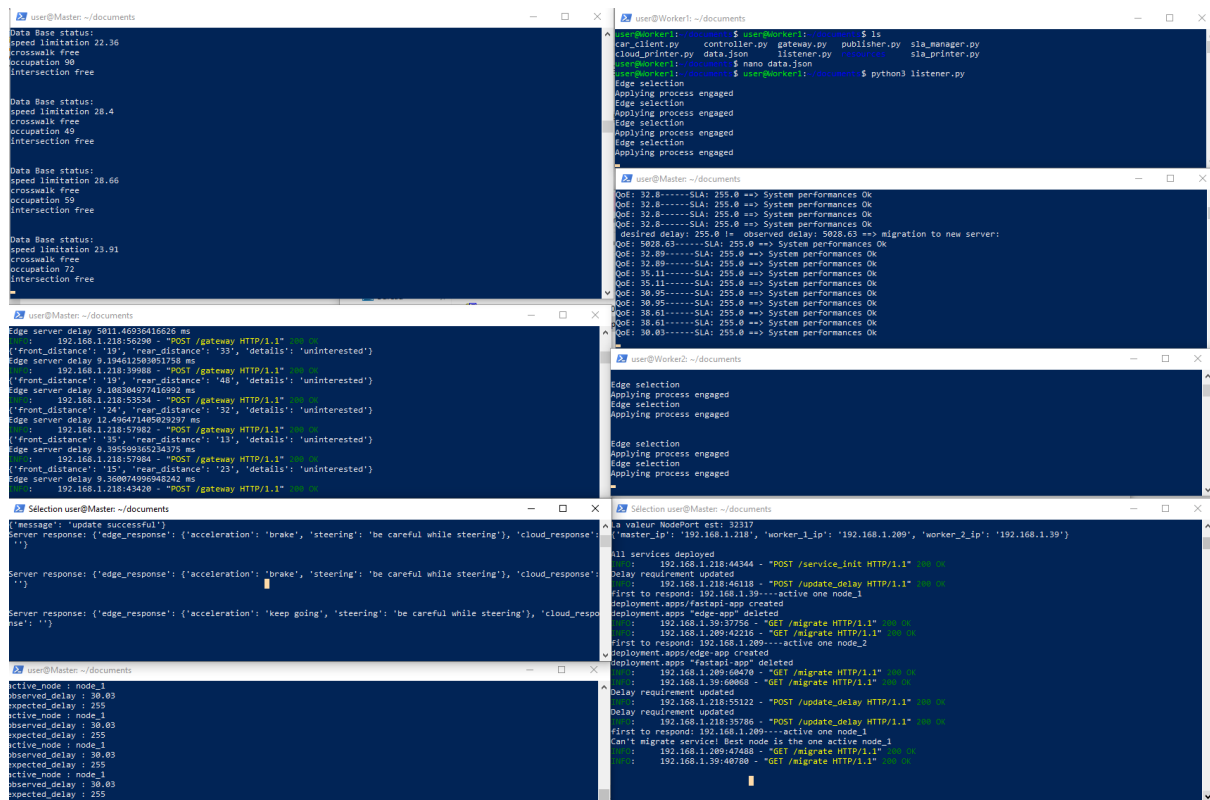


Figure 7 : Screenshot of our final result