Politecnico di Torino

Resource Usage Limitation

Matteo Frigo

Politecnico di Torino

19 July 2025

Contents

1 Introduction

The objective of this lab was created by the Cybersecurity course of Politecnico di Torino, which requires the complete creation of a laboratory activity based on the concepts of “domains of protection” or “multi-tenancy”, starting from the ResourceQuotas feature of Kubernetes and searching for other technologies for resource usage limitation.
In today’s world, it’s essential to have a system that lets us logically divide and isolate different teams, groups of users, or user work on the same machine, reducing the cost of hardware and management. In fact, this lab aims at understanding and practicing the instruments enabling multi-tenancy in the Kubernetes environment.
The focus of the project is resource usage limitation, where we don’t focus only on resource differentiation, but also on the constrainment and isolation of multiple teams inside our Kubernetes cluster. This will require instruments that let us logically divide the environments of different teams by using ResourceQuota, LimitRange, Access Control, and Network Policy. We will understand those tools, providing also some use case examples. All the tools we will see can be combined together inside a single cluster, fundamental to providing an advanced multi-tenancy environment for multiple customers.

2 Tools and Setup

Before analyzing the different tools used in this laboratory, we want to discuss the main goal of this project: managing and configuring a system capable of supporting multi-tenancy and multi-team environments, where each team typically deploys a small number of workloads that scale with the complexity of the service. What we aim to observe are the tools that allow us to share a cluster among different users, while still logically isolating each of them, making it appear as though they are the only ones using the Kubernetes cluster.
A Kubernetes cluster consists of a control plane and a data plane. The control plane consists of kubernetes software to manage the back end part of the cluster, we will see technology as namespaces, ResourceQuotas, LimitRange and Access Control. Then, we have the data plane which are composed of worker nodes where tenant workloads are executed as pods (more details in subsection 2.4).

2.1 Tools

For this project we used a set of high level tools, letting us configure and set up all the process needed. Here there is a brief set of components I used:

- Kubernetes.
- Docker.
- Github.
- Resource Quotas.
- YAML documents.

All the images used by me to handle and manage this project can be find in GitHub. Here there are all the needed command to install my repository [1]:

```
sudo apt update
sudo apt install git

% Check if it's correctly installed:
git --version
% Clone repository:
git clone https://github.com/FrigoMatteo/Resource-Limitation.git
```

Next, I have created a Docker repository where you can install the images I used for the different pods. The installation of them isn't necessary for the completation of this laboratory, since you can easily create your owns one, but for the good of knowleadge I am also sharing this. The images are handled by the docker hub, which is a repository where users can store, manage and share containers with other people. My Docker hub images can be pulled with [2]:

Listing 1: Pull any image from my repository

```
docker pull matteofrigo1618/repository_name:tag_name
```

Now we are going to deep dive on the tools necessary for this laboratory.

2.2 Docker

Docker [3] is a software platform which allows users to easily create lightweight, portable, and self-contained containers. Docker has also a desktop version, which provides a GUI dashboard letting us see and better manage the containers we have access to, as shown in Figure 1. Unfortunately, Docker desktop has limited functionality instead of CLI, but, in many times, it provides an easy and fast way to check up all your containers.

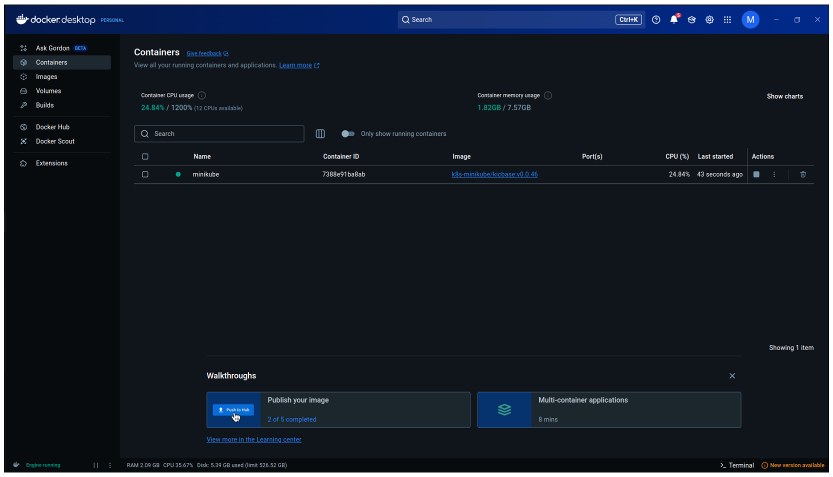
2.2.1 Relation docker-minikube

The purpose of using Docker is to give Minikube an engine where it can start its containers and, consequently, its pods. It may seem tricky, but in practice, Docker hosts a Minikube container, which then creates and manages other containers through its internal Docker engine. This doesn't stop here:

since we are talking about containers, we can use the images we create or pull from Docker Hub to run inside the Minikube container, which will handle them.
Docker images are read-only templates that contain the instructions needed to create a container. They are essentially snapshots of the libraries and dependencies used inside a Docker container, so you will have the same containers I used inside Minikube.

2.2.2 Docker setup

Here we provide the procedure to setup Docker. Here you can find original guideline to install it: https://docs.docker.com/engine/install/.



View all your running containers and applications.

Container CPU usage
24.84% / 1200% (12 CPUs available)

Container memory usage
1.82GB / 7.57GB

Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
minikube	7388e91ba8ab	k8s-minikube/kicbase:v0.0.46		24.84%	43 seconds ago	

Showing 1 item

RAM 2.09 GB CPU 35.67% Disk: 5.39 GB used (limit 526.52 GB)

Figure 1: Docker Desktop Dashboard

2.3 YAML documents

We use this kind of documents to describe and define a particular state we want to set in kubernetes. We will use these mechanisms to define resources for the kubernetes deployments, ResourceQuotas and others inside our cluster.

Listing 2: Example deployment of four pods, one container each

```yaml
apiVersion: apps/v1 # API version used by kubernetes
kind: Deployment # Kind of Resource we are creating
metadata:
  name: vite-deployment # Used to identify the deployment inside the cluster
  labels:
    app: vite
spec:
  replicas: 4 # Number of pods to generate
  selector:
    matchLabels: # Used to seelct the pods with specific label
      app: vite
  template:
    metadata:
      labels:
        app: vite # This label must correspond with the one of "selector"
    spec:
      containers:
      - name: vite # Container name
        # Image to use for the container (taken from Dockerhub)
        image: matteofrigo1618/vite_application:v1.0
        ports:
          # Each container will have <ip_container>:3000 for cluster's internal communication
        - containerPort: 3000
        resources:
          requests:
            # 250m => 250 millicpu
            cpu: "250m" # request 0.25 core
            memory: "128Mi" # request 128MB of memory
          limits:
            cpu: "500m" # maximum CPU it can take: 0.5 core
            memory: "256Mi" # maximum memory it can take: 256MB
```

To apply or delete yaml documents inside the kubernetes cluster, we use the following command:

Listing 3: Add or remove YAML document

```
% Apply
kubectl apply -f <file>.yaml
% Delete
kubectl delete -f <file>.yaml
```

2.4 Kubernetes

Here we define Kubernetes [4], which is an open-source container orchestration tool. It helps us orchestrate hundreds of containerized applications in different environments, including both physical and cloud machines. This is a very useful tool, since we don't want to manage all the containers ourselves — they can be complex, time-consuming, and prone to human error during configuration. Kubernetes helps us avoid such problems by handling the containers automatically. Working with Kubernetes provides the following advantages:

- Availability.Applications experience no downtime. Kubernetes supports blue-green deployments.
- Scalability. It can easily scale up the number of containers based on user requests, and conversely, scale down when demand is low.
- Disaster recovery. Kubernetes includes mechanisms to handle failures by restoring containers to their last known state, or automatically deploying new ones if necessary.

2.4.1 Kubernetes components

Here we introduce the components of Kubernetes. First, we define the cluster, which is made up of multiple nodes, each of which represents a single compute host (virtual or physical machine). Then, we have two types of nodes:

- The master node, which serves as the control plane. It manages the other nodes in the cluster. When we use kubectl to communicate with the cluster, we send requests to the master node, which then calls the specific node and operation.
- The worker nodes, which are used to deploy, run, and manage containerized applications.

As we will see in subsubsection 2.4.2, Minikube manages nodes in a different way. Next, we define the most important components we will use and interact with during this laboratory:

- Kubernetes Deployment – These objects are defined by YAML files that declare the desired state we want our application to follow. See an example in Listing 2.
- Container – The application that contains our code and runs inside a pod.
- Pod – The basic execution unit of a Kubernetes application, which may consist of one or multiple containers.
- Namespace – A way to logically group resources (pods, services, etc.) inside a cluster. For example, we can define a namespace for the front-end application, with specific limits and rules, different from those for the back-end application. In addition, many Kubernetes security policies are scoped to namespaces, making them fundamental to services.

- kubectl – This is used by developers to communicate directly with the master node to access or send information about the cluster.
- Kubelet – Each worker node includes this component, which communicates with the kube-apiserver on the control plane of the master node and receives commands from it.

2.4.2 Minikube

Minikube is mainly used for testing application in a local machine. I used this component for the sake of the project and the available resources. In fact, minikube is basically a local machine version of kubernetes. In fact, with minikube we can launch a cluster in a single node machine, by having the master and multiple worker node all together, something that it's not possible with the normal kubernetes.

Minikube setup:

Here I consider the installation of minikube for an Ubuntu operating system. In case of errors of fails I suggest you to follow the minikube guideline at https://minikube.sigs.k8s.io/docs/start/. We install the latest minikube stable release on x86-64 Linux.

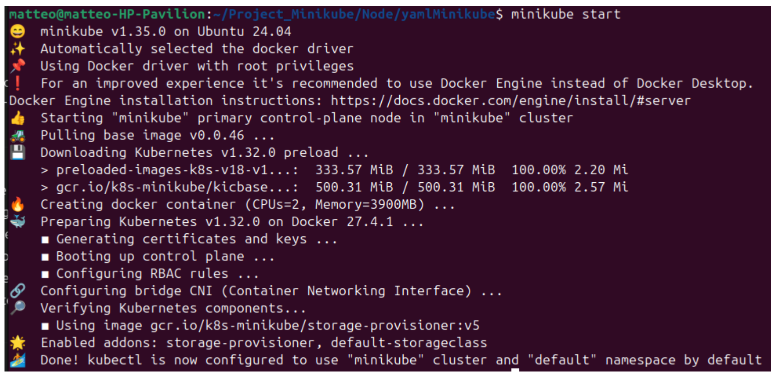
```
curl -LO https://github.com/kubernetes/minikube/releases/latest/download/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube && rm minikube-linux-amd64
```

After doing so, we can use the command to start the minikube cluster:

Listing 4: Start minikube command line

```
% We want to specify to use Docker as container engine
minikube start --driver docker
```

What you should see it's a screen like in Figure 2. It will take some times, since it needs to pull Minikube from the repository, but the next time you will open it, it will be faster.



```
matteo@matteo-HP-Pavilion:~/Project_Minikube/Node/yamlMinikube$ minikube start
😄  minikube v1.35.0 on Ubuntu 24.04
✨  Automatically selected the docker driver
📌  Using Docker driver with root privileges
❗  For an improved experience it's recommended to use Docker Engine instead of Docker Desktop.
Docker Engine installation instructions: https://docs.docker.com/engine/install/#server
👍  Starting "minikube" primary control-plane node in "minikube" cluster
🚜  Pulling base image v0.0.46 ...
💾  Downloading Kubernetes v1.32.0 preload ...
    > preloaded-images-k8s-v18-v1...:  333.57 MiB / 333.57 MiB  100.00% 2.20 Mi
    > gcr.io/k8s-minikube/kicbase...:  500.31 MiB / 500.31 MiB  100.00% 2.57 Mi
🔥  Creating docker container (CPUs=2, Memory=3900MB) ...
🐳  Preparing Kubernetes v1.32.0 on Docker 27.4.1 ...
    ▪ Generating certificates and keys ...
    ▪ Booting up control plane ...
    ▪ Configuring RBAC rules ...
🔗  Configuring bridge CNI (Container Networking Interface) ...
🔎  Verifying Kubernetes components...
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟  Enabled addons: storage-provisioner, default-storageclass
🏄  Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

Figure 2: Minikube start procedure

Now to check if everything is okay, you should be able to send the next line and see the result of the status of minikube:

```
$ minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

Make sure everything is running and configured, since these, as previously said, are important component necessary for our system. For more details about the commands and related information about Minikube, I redirect you to the official website: https://minikube.sigs.k8s.io/docs/handbook/.

2.5 ResourceQuota and LimitRange

Now, we are going to explain why ResourceQuota [5] and LimitRange are fundamental in a project involving Kubernetes. In the current configuration, any node or pod can use as many resources as it wants. This might be acceptable if we have an abundance of resources, but it becomes problematic in real systems with many pods and containers running simultaneously. This can lead to what is called the ”noisy neighbor” problem, where containers compete for and steal resources from each other. A possible solution is to use resource requests and limits for specific containers. An example can be seen in Listing 2, where we define the code to set resource limits for individual containers (not pods). We define:

- Request – This is the minimum amount of resources a container is guaranteed. Even if the container doesn't use all those resources, they are reserved and always available.
- Limit – This is the maximum amount of resources a container is allowed to use.

However, this mechanism alone is not enough. If a container or pod is left unregulated, one pod might consume all the available resources, depriving others that also need them. In this case, we are limiting the resources per container, but not globally per pod. If a team forgets to configure these limits, it can lead to chaos within the cluster. To ensure a stable and well-managed environment, we use LimitRange and ResourceQuota. These tools are particularly useful in environments where teams may forget to define resource limitations for their applications. That's why we introduce:

- ResourceQuota – This allows us to set resource usage limits per namespace, preventing any single namespace from consuming all the available resources. When distributing resources among teams, the system works on a first-come, first-served basis. See the setup in subsubsection 2.5.1.
- LimitRange – Unlike ResourceQuota, which sets quotas at the namespace level, LimitRange defines default resource limits for individual containers within a namespace. See its setup in subsubsection 2.5.2.

The cooperation between these two mechanisms provides control and isolation in environments with multiple teams or tenants, especially in shared or cloud-based infrastructures serving multiple customers. In addition, they help developers avoid errors, such as forgetting to set resource limits, which could cause future problems in the cluster.

2.5.1 ResourceQuotas setup

If we work with ResourceQuotas for CPU and memory resources, we enforce that every pod in that namespace sets limits for those resources. If a resource quota is enforced in a namespace for either CPU or memory, you, and any other clients, must specify either requests or limits for that resource for every new pod you submit, as specified in Listing 2. If you don't, the control plane may reject the admission of that pod. One way to enforce this **automatically and globally**, without relying solely on users to configure it manually, is by using LimitRange, which defines default requests and limits for each pod.
For example, in a cluster with a capacity of 16 GiB of RAM and 12 cores, if we need to distribute computational resources among different teams, we can describe this configuration using a YAML document, as shown in Listing 5. First, we need to create the namespace where our ResourceQuota rules will be applied:

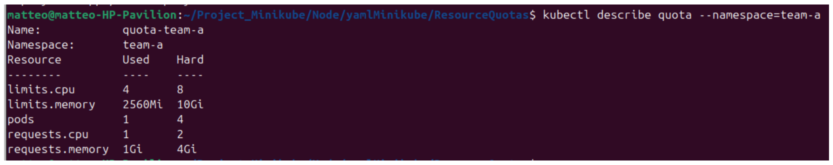
```
$ kubectl create namespace <name>
namespace/<name> created
```

Then, after succeding in creating the namespaces, we can create the YAML document to launch and apply the ResourceQuotas to the two namespaces:

Listing 5: Deployment namespace ResourceQuotas

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: quota-team
  namespace: <namespace name>
spec:
  hard: # Minimum amount of resources they will have
    pods: "<max pods number>" # [0, 1, 2, 3..]
    requests.cpu: "<cores number>" # [0, 1, 2, 3..] = [1000m, 2000m, ...]
    requests.memory: <memory number> # [100Mi, 200Mi,.., 1Gi, 2Gi..]
    # Maximum amount of resources they can take
    limits.cpu: "<core number>"
    limits.memory: <memory number>
```

Now, if we try to launch an example of deployment inside the namespace, we should see the applied ResourceQuota to the specific environment as shown in Figure 3.



```
matteo@matteo-HP-Pavilion:~/Project_Minikube/Node/yamlMinikube/ResourceQuotas$ kubectl describe quota --namespace=team-a
Name:            quota-team-a
Namespace:       team-a
Resource         Used    Hard
--------         ----    ----
limits.cpu       4       8
limits.memory    2560Mi  10Gi
pods             1       4
requests.cpu     1       2
requests.memory  1Gi     4Gi
```

Figure 3: Describe Resource Quotas

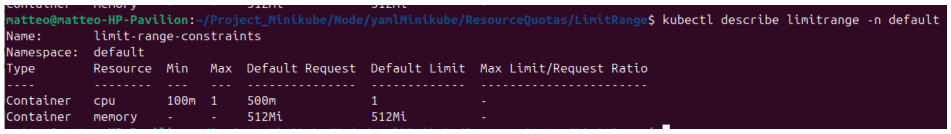
2.5.2 LimitRange setup

By default, containers run with unbounded compute resources in a Kubernetes cluster. We have seen that by using Kubernetes ResourceQuotas, we can restrict the consumption and creation of cluster resources within a specified namespace. However, within a namespace, a pod can still consume as much CPU and memory as allowed by the ResourceQuotas. So, what we want to add to the cluster is the ability to prevent a single object from monopolizing all available resources within a namespace.
This mechanism is the same as the one explained for setting limits and requests on individual pods, such as in Listing 2, but in this case, it will be applied globally to every pod or container within the namespace. The LimitRange object can enforce constraints related to maximum and minimum resource usage, and can also set default requests and limits for compute resources within a namespace for every container or pod.

Listing 6: LimitRange Setup

```
apiVersion: v1
kind: LimitRange
metadata:
  name: limit-range-constraints
  namespace: <namespace name>
spec:
  limits:
    # Default values if the container don't specify the limit and request:
  - default: # Defines default limits
      cpu: "<cores number>" # [0, 1, 2, 3..] = [1000m, 2000m, ...]
      memory: <memory number> # [100Mi, 200Mi,.., 1Gi, 2Gi..]
    defaultRequest: # Defines default requests
      cpu: "<cores number>"
    # Maximum and minimum values a container can take when initialized
    max:
      cpu: "<cores number>"
      memory: "<memory Mi>"
    min:
      cpu: "<cores number>"
      memory: "<memory Mi>"
    type: <type> # [Container, Pod]
```

We can see if it's operative by following the procedure in Figure 4:



```
matteo@matteo-HP-Pavilion:~/Project_Minikube/Node/yamlMinikube/ResourceQuotas/LimitRange$ kubectl describe limitrange -n default
Name:       limit-range-constraints
Namespace:  default
Type        Resource  Min   Max  Default Request  Default Limit  Max Limit/Request Ratio
----        --------  ---   ---  ---------------  -------------  -----------------------
Container   cpu       100m  1    500m             1              -
Container   memory    -     -    512Mi            512Mi          -
```

Figure 4: Describe Resource Quotas

2.5.3 Possible errors

I would pay close attention when creating pods or deployments with ResourceQuota and LimitRange limitations. In fact, when defining the container resources, we must be careful about how much we request.
For example, if we consider a setup like ResourceQuota with "request.cpu = 2000m", and we define a deployment, such as the one shown in Listing 7, that includes two pods each requesting cpu = 2000m, this will cause issues. The first pod will be created using the requested CPU, but the second one will fail to be created because all available CPU resources (according to the request quota) are already consumed. To allow both pods to be scheduled, we should instead set request.cpu = 1000m, so that both pods can run simultaneously.
Remember: we are now setting the minimum amount of resources a pod can use. Once initialized, both pods can still request additional resources—up to the limit specified in the "limits" section of the YAML document. I would also be careful when defining deployment constraints for all resource parameters. If you make a mistake, the system may still **return a success message**, even though the pods or containers were not initialized correctly.

Listing 7: Deployment web application with team-a namespace

```
apiVersion: apps/v1
kind: Deployment
**** Deployment configurations ****
spec:
  replicas: 2
    **** Deployment configurations ****
      containers: # ATTENTION IN SETTING CONSTRAINTS
        resources:
          requests:
            cpu: "2000m"
            memory: "1024Mi"
          limits:
            cpu: "4"
            memory: "2560Mi"
```

2.6 Access Control

Access control is one of the most important types of isolation for the control plane part of the kubernetes cluster, since it requires authorization for the operations we perform. If teams can access or modify each other's API resources, they can change or disable policies, thereby negating any protection those policies may offer. As a result, it is critical to ensure that each tenant has access only to the namespaces they need—and nothing more. This is known as the **Principle of Least Privilege**.
Role-Based Access Control (RBAC) is commonly used to enforce authorization in the Kubernetes control plane. In a multi-team environment, RBAC must be used to restrict tenants' access to the appropriate namespaces and to ensure that cluster-wide resources can only be accessed or modified by privileged users, such as cluster administrators.
This policy is mainly used to restrict access to kubernetes API if the tenants ask for specific pods/deployment to the administrator to perform read or write operations to the configurations of the kubernetes cluster. We can divide these components into three categories:

- Identity. They can be users, which are externally defined, or Service Accounts, that are entities managed by kubernetes for the pods.
- Role. The role defines the group of permission inside a specific namespace. We can also have RoleCluster, which are the set of rules for a cluster or group of namespaces.

- Role Binding. We assign a Role or a RoleCluster to one or multiple users/group.

2.6.1 Identities

We have two categories of identities: users and Service Accounts. A user is an external identity, which is not managed by the Kubernetes cluster, but it is still possible to use it and authenticate via certificates such as X.509. Different users have specific permissions to access or modify the entire cluster through tools like kubectl or the dashboard commands, as we have seen so far, with possible restrictions based on policies. However, for the purpose of multi-tenancy, users do not fit well within the structure. This is better managed by Service Accounts, which can represent either a user (human) or an application, providing an identity for processes running inside a pod. Using Service Accounts, we can monitor, scale up, or modify the resources available within our namespace. Moreover, unlike users, Service Accounts are managed (created, modified, or monitored) by Kubernetes itself, having their own objects that we can reference. We can create a Service Account using the following YAML code:

Listing 8: Service Account creation

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: <name Service Account>
  namespace: <name Namespace>
```

2.6.2 Role and ClusterRole

A Role or ClusterRole contains rules that represent a set of permissions. A Role always sets permissions within a particular namespace; when you create a Role, you have to specify the namespace it belongs in. If we want to define a role cluster-wide, we use a ClusterRole. We can create a Role or Cluster role with the following YAML commands:

Listing 9: Role or Cluster Role creation

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: <Role> # [Role, ClusterRole]
metadata:
  namespace: <name Namespace>
  name: <Name of the Role>
rules:
- apiGroups: [<Api Group>]
  resources: [<resources>] # [pods, services, nodes, deployments, etc..]
  verbs: [<constraints>] # [get, list, delete, watch, create, update, etc..]
```

Inside the YAML document presented, we also define the "apiGroups". These are used by kubernetes to group and organize the resources inside his API REST. We have different mods, the main one is empty quotation, which include the APIs for resources as pods, services configmap and others. So, based on the apiGroup we choose, we will have different resources available.

2.6.3 Role and Cluster Binding

A role binding grants the permissions defined in a role to one or more users or Service Account. A RoleBinding grants permissions within a specific namespace whereas a ClusterRoleBinding grants that access cluster-wide. Here there is how we define it:

Listing 10: RoleBinding creation

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: <name> # Name we want to give to binding
  namespace: <name Namespace>
```

```
subjects:
- kind: ServiceAccount # In case you want to bind a user => "User"
  name: <name of Service Account>
  namespace: <name Namespace>
roleRef:
  kind: <type of Role> # [Role, ClusterRole]
  name: <name of the Role>
  apiGroup: rbac.authorization.k8s.io
```

2.7 Network Policies

Inside a cluster, by default, each pod can talk to any other pod inside it. So a tenant, in this case, could communicate with resources of other tenants. For this reason we want to limit the communication with network rules inside the cluster. For the purpose of our laboratory, we will focus only on the isolation of the namespace, since deeper isolations of specific pods or resources are dependent based on the application we want to create/install inside our namespace.
For example, there can be rules where the Front-end pod of a web application can talk only to the back-end pod, not permitting direct communication with a database, but forcing it to communicate with the back-end for specific kind of operations.
In addition, this can also protect us against any possible infections of pods inside the kubernetes cluster, without permitting an infected pod to communicate and propagate to other resources of other namespaces. Now, I am going to show you only a namespace isolation between multiple tenants:

Listing 11: Network policy setup

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: <policy name you want>
  namespace: <name namespace>
spec:
  # podSelector => {} => affect all pods of the namespace
  # If we want to select specific pods => {podName1, podName2..}
  podSelector: {}
  policyTypes:
  - Ingress # Control traffic in ingress
  ingress:
  - from:
    - podSelector: {} # This imply we consent traffic only from our namespace
```

As we saw, we only limit traffic going inside our namespace, but the pods inside can still communicate with namespace's resources of the cluster or external cluster. In addition, this policy is applied only for internal cluster communications, so it acceptes any incoming traffic from outside the cluster.

3 Show Case Examples

Here we are going to introduce and show different tests we have done through the usage of the multiple instruments defined during the section 2 chapter. We will proceed gradually, by giving first an example of how you can start minikube, then moving on showing the checks you can perform for testing everything is working correctly.

3.1 Show case Minikube Project

Now, we are going to see the basic procedures to start a cluster. After downloading and setting all the tools needed in section 2, we will open our terminal and start our minikube, composed of only one node (both master and worker). You'll see an image as shown in Figure 2.
Now, we'll proced to launch the following code, which it will configure a web application, composed of four pods. These pods will be launched inside the default namespace:

Listing 12: Deployment web application

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: vite-deployment
  labels:
    app: vite
spec:
  replicas: 4
  selector:
    matchLabels:
      app: vite
  template:
    metadata:
      labels:
        app: vite
    spec:
      containers:
      - name: vite
        image: matteofrigo1618/vite_application:v1.0
        ports:
        - containerPort: 3000
```

Following, we will write the service yaml document. This code is useful to expose one or multiple pods to the external network of minikube, i.e. our local system. In addition, the service type will provide load balancing between all pods, managing the multiple requests from different users.

Listing 13: Service web application service

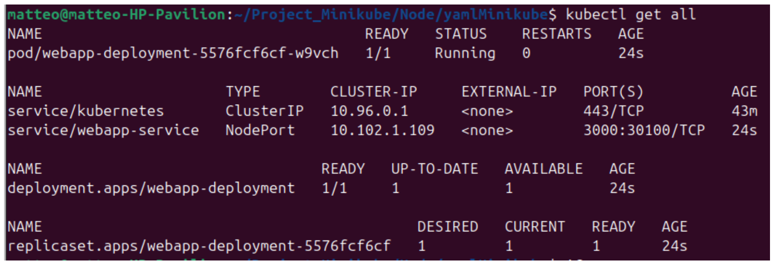
```yaml
apiVersion: v1
kind: Service
metadata:
  name: webapp-service # Name we want to assign the service
spec:
  type: NodePort # It's the type of port required to connect with the external
  selector:
    app: vite # we define at which pods it needs to connect to
  ports:
    - protocol: TCP
      port: 3000 # Internal exposed port inside cluster.
      targetPort: 3000
      nodePort: 30100 # Port to extern cluster. Must be between 300000 - 32767.
```

Next, by sending the following commands in CLI you will create the four pods and a service which enable us to connect to it.

Listing 14: Launch deployment and service

```
% For my project they were. Deployment:
kubectl apply -f vite_application.yaml
% Service:
kubectl apply -f vite-svc.yaml
```

Now, we have launched the two YAML documents, i.e. Service and Deployment, to create and make our application available to external. If everything worked fined, by passing the command ”kubectl get all”, you should see the result in Figure 5.



```
matteo@matteo-HP-Pavilion:~/Project_Minikube/Node/yamlMinikube$ kubectl get all
NAME                                    READY   STATUS    RESTARTS   AGE
pod/webapp-deployment-5576fcf6cf-w9vch  1/1     Running   0          24s

NAME                     TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
service/kubernetes       ClusterIP   10.96.0.1      <none>        443/TCP          43m
service/webapp-service   NodePort    10.102.1.109   <none>        3000:30100/TCP   24s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/webapp-deployment   1/1     1            1           24s

NAME                                           DESIRED   CURRENT   READY   AGE
replicaset.apps/webapp-deployment-5576fcf6cf   1         1         1       24s
```

Figure 5: Minikube get all components

To access the application we have created, we will use a tunnel to connect to Minikube. With the following command, you will be able to connect to the application:

Listing 15: Open tunnel to minikube

```
$ minikube service webapp-service --url
http://<localhostIP>:<PortNumber Tunnel>
! Because you are using a Docker driver on linux, the terminal needs to be open to run it.
```

If you have run it, you will see that the ”PortNumber Tunnel” isn’t 30100 as specified before, but a random value generated by Minikube. This is because we are creating a tunnel from our local system, to the minikube cluster. This command is **necessary** for the connection because we are using minikube with driver of Docker. It would be different the procedures to connect to the web application if we were using VirtualBox or VM-based drivers.

3.2 Show case Resource Quotas

Here we are going to show an example of implementation of a ResourceQuota constraints across two namespaces, which logically divide a clusters, isolating the different resources. This can be seen as a perfect example of how we can differentiate our cloud resources among the customers, based on what contract we have with them.

Listing 16: Deployment namespace ResourceQuotas

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: quota-tenant-a
  namespace: tenant-a
spec:
  hard:
    # Minimum amount of resources they will have
    requests.cpu: "2"
    requests.memory: 4Gi
    # Maximum amount of resources they can take
    pods: "4"
    limits.cpu: "8"
    limits.memory: 10Gi
---
apiVersion: v1
kind: ResourceQuota
metadata:
  name: quota-tenant-b
  namespace: tenant-b
spec:
  hard:
    # Minimum amount of resources they will have
    requests.cpu: "1"
    requests.memory: 10Mi
    # Maximum amount of resources they can take
    pods: "4"
    limits.cpu: "2"
    limits.memory: 50Mi
```

We apply the constraint to the two namespaces and then to the cluster. Remember, that first we need to create the two namespaces if you haven't already done it:

Listing 17: Apply Resource Quota feature

```
$ kubectl create namespace tenant-a
$ kubectl create namespace tenant-b
$ kubectl apply -f resource_quota_exceed_mem.yaml
```

As we can see for tenant-b, we allocated 50MB of memory as limit. Now, we want to purposly use more of that, to show what it's going to happen when we try to use more memory than we should. We want to test it because it can happen that a group team has certain limit of resources, but the **application requires** more of that. So we should block them and change contract or give more resources based on the relationship we have with the client, for example if we are in a company environment between departments. Here's the python code we will use to trigger this limit:

Listing 18: LimitRange Setup

```
import time
def allocate_memory(mb):
    size = mb*1024*1024
    return bytearray(size)

time.sleep(5)
print("We start by allocating 10 MB of memory")
mem_chunks = [allocate_memory(10)] # 10 MB
time.sleep(30)

print("After ten seconds we will try allocating 50MB")
try:
    mem_chunks.append(allocate_memory(50)) # Supera i 50 MB totali
    print("Process still alive")
except MemoryError:
    print(" MemoryError: we overflow the limit")

# Wait to observe the pod behaviour
time.sleep(60)
print("Finished")
```

After writing the code and creating the appropriate Docker image of the program, we shall configure the deployment. We can reuse the code of Listing 2, but modifying the following lines:

Listing 19: Deployment exceeding memory application

```
*** Configuration Listing 2 ***
  namespace:tenant-b
  labels:
    app: exceed_memory
spec:
  replicas: 1 # Number of pods to generate
  *** Configuration Listing 2 ***
      containers:
       image: <you-image-name-python>
       resources:
         requests:
           cpu: "500m"
           memory: "5Mi"
         limits:
           cpu: "1"
           memory: "50Mi"
```

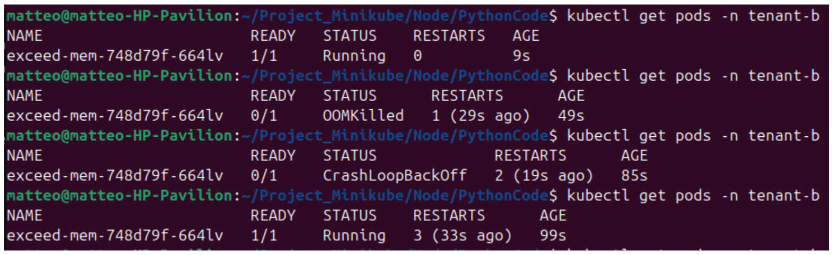
Obviously, if we try to configure more resources instead of the limit of what we configured in the ResourceQuota, the deployment will fail, without launching the requested pod and forcing us to configure the pod in a more coeherent way. You can see a scenario of this problem in subsubsection 2.5.3. After applying the deployment we can see the logs of the application by launching:

Listing 20: Logs Pod

```
$ kubectl get pods -n tenant-b
% Get the name of the pod to be used next:
$ kubectl logs <name-pod> -n tenant-b
```

What it's going to happen is figured in Figure 6. Where we can see the looping of three status for the specific pod:

- Running. Pod is running as it should be. It's before we try to configure more than the limit of memory for the pod.
- OOMKilled. The pod has gone over the limit of memory we set. The status tells that kubernetes killed that process and reload it from beginning.
- CrashLoopBackOff. The pod continually crash and kuberentes is waiting some time before restarting it.



```
matteo@matteo-HP-Pavilion:~/Project_Minikube/Node/PythonCode$ kubectl get pods -n tenant-b
NAME                        READY   STATUS    RESTARTS   AGE
exceed-mem-748d79f-664lv   1/1     Running   0          9s
matteo@matteo-HP-Pavilion:~/Project_Minikube/Node/PythonCode$ kubectl get pods -n tenant-b
NAME                        READY   STATUS      RESTARTS      AGE
exceed-mem-748d79f-664lv   0/1     OOMKilled   1 (29s ago)   49s
matteo@matteo-HP-Pavilion:~/Project_Minikube/Node/PythonCode$ kubectl get pods -n tenant-b
NAME                        READY   STATUS             RESTARTS      AGE
exceed-mem-748d79f-664lv   0/1     CrashLoopBackOff   2 (19s ago)   85s
matteo@matteo-HP-Pavilion:~/Project_Minikube/Node/PythonCode$ kubectl get pods -n tenant-b
NAME                        READY   STATUS    RESTARTS      AGE
exceed-mem-748d79f-664lv   1/1     Running   3 (33s ago)   99s
```

Figure 6: Pod exceeding memory

3.2.1 Addional constrainments

In addition to the constraints of the ResourceQuota. We can configure a namespace quotas with different classes of priority and resource quantity. In fact, based on the pod we launch, we can define different constraints inside the same namespace:

Listing 21: Class ResourceQuota setup

```
apiVersion: v1
kind: List
items:
- apiVersion: v1
  kind: ResourceQuota
  metadata:
    name: pod-high
    namespace: <namespace name>
  spec:
    hard:
      cpu: "6"
      memory: "16Gi"
      pods: "6"
    scopeSelector:
      matchExpressions:
      - operator: In
        scopeName: PriorityClass
        values: ["high"]

- apiVersion: v1
  kind: ResourceQuota
  metadata:
    name: pod-low
    namespace: <namespace name>
  spec:
    hard:
      cpu: "2"
      memory: "4Gi"
      pods: "2"
    scopeSelector:
      matchExpressions:
      - operator: In
        scopeName: PriorityClass
        values: ["low"]
```

Then, in the deployment of the pods, we can add our additional class constraint:

Listing 22: Set class - deployment

```
 *** Deployment configurations ***
 resources:
      requests:
        memory: "10Gi"
        cpu: "500m"
      limits:
        memory: "10Gi"
        cpu: "500m"
  priorityClassName: high # Value defined in the matchExpression.values
```

Obviously, we have just scratch the surface of additional constraints we can set to the ResourceQuota.

3.3 Show case combination ResourceQuotas and LimitRanges

Until now we have explore the ResourceQuota setting and what to insert inside the deployment of your containers. What if the team managing it, forgets to configure the limit and request values inside the deployment? What we can do is to combine LimitRange and ResourceQuota:

Listing 23: LimitRange setup with ResourceQuota

```
apiVersion: v1
kind: LimitRange
metadata:
  name: limit-range-constraints
  namespace: tenant-b
spec:
  limits: # We need to respect constraint of ResourceQuotas of before
  - default:
      cpu: 1000m
      memory: 50Mi
    defaultRequest:
      cpu: 500m
      memory: 5Mi
    type: Container
```

With the code of Listing 23, we consider it with the implementation of ResourceQuota of Listing 16, where we set default limits and request for all the pods inside a namespace. So, when deploying our application Listing 2, if we want we can remove the "resources" fields and launch it inside the namespace, since its set automatically by the limitRange. With this method, if a developer forget to set the resources used by the application, or for standardizing deployment, we can initialized them by default. In fact, after deploying any application deployment you will see the default request and limit constraints by sending the following line:

Listing 24: Show constraints deployment

```
$ kubectl describe quota -n <namespace>
$ kubectl describe limitrange -n <namespace>
```

3.4 Show case Access Control

Another measure we can apply to our cluster is access control and network policies, which we saw in subsection 2.6 and subsection 2.7. These functionalities let us impose strict rules inside our namespace for every tenant. We divide the two into:

- Network policies, as we also saw before in subsection 2.7, we defined a general rule where different pods inside a namespaces, cannot communicate with others, so isolating the network connections between namespaces. Here we are not going to implement it, since it's personal for specific cluster and the implementation of it is generally equal to any other definition of service or resources inside kubernetes.

- Role-based access control, where we impose rule the services apply. This can change based on what activity or access we want to give to the pods owned by a specific tenant. In subsubsection 3.4.1 we described an example of only observation role for the customer, making the administrator of the cluster the only one who can change the characteristics or setting of a namespace. This is a feature we can implement, in addition to many other based on the service, security and management we want to give to the customers. **By default**, the pods or containers cannot access any information related to the cluster, so implementing these rules let a tenant access additional resources about the cluster (based on what we permit it to do).

3.4.1 Role-Based Access Control

Below we implement the yaml code needed to create, manage and bind a rule to the service account. First lets create the namespace where a customer tenant can work, through the command Listing 17 with name "torino-tenant". Next, we create and apply the following code through the "kubectl apply" command:

Listing 25: ServiceAccount, rule and ruleBinding

```yaml
# we define a Service account
apiVersion: v1
kind: ServiceAccount
metadata:
  name: service-torino
  namespace: torino-tenant
---
# we define a Role we want to apply to our service account
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: read-pod
  namespace: torino-tenant
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list"]
---
# we define the bind between role and service account
```

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: bind-torino
  namespace: torino-tenant
subjects:
- kind: ServiceAccount
  name: service-torino
  namespace: torino-tenant
roleRef:
  kind: Role
  name: read-pod
  apiGroup: rbac.authorization.k8s.io
```

After creating the service account, role and bind these two together. We can implement the service account related to pod resources inside the namespace. In fact, what I image from my example is to create a pod resource which was able to access and look at the pods details of his own namespace. Below we can see the code used to achieve and create this pod. We used bitnami, which it's used to execute kubectl commands inside a pod, mainly used for debugging purposes.

Listing 26: Lauch pod with specific ServiceAccount

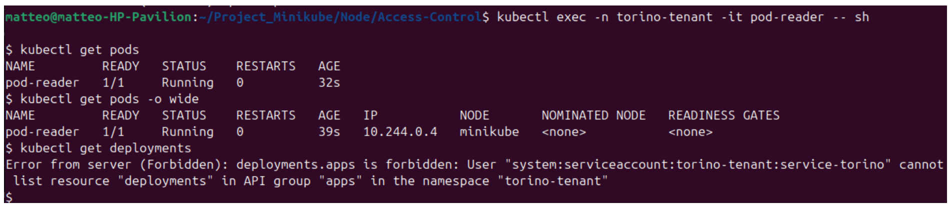
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-reader
  namespace: tenant-b
spec:
  serviceAccountName: pod-reader
  containers:
  - name: reader
    image: bitnami/kubectl:latest
    command: ["sleep", "3600"] # Used to keep the container alive
```

Then, after applying the specified code, we can see the result by writing the following code in the cmd:

Listing 27: Show constraints deployment

```
$ kubectl exec -n torino-tenant -it pod-reader -- sh
# -- Enter pod ssh --
$ kubectl get pods
$ kubectl get pods -o wide
$ kubectl get deployments
```

The result of the previous commands will be the possibility to access only the pods information. In fact, we can see the result also in the following image:



```
matteo@matteo-HP-Pavilion:~/Project_Minikube/Node/Access-Control$ kubectl exec -n torino-tenant -it pod-reader -- sh

$ kubectl get pods
NAME         READY   STATUS    RESTARTS   AGE
pod-reader   1/1     Running   0          32s
$ kubectl get pods -o wide
NAME         READY   STATUS    RESTARTS   AGE   IP           NODE       NOMINATED NODE   READINESS GATES
pod-reader   1/1     Running   0          39s   10.244.0.4   minikube   <none>           <none>
$ kubectl get deployments
Error from server (Forbidden): deployments.apps is forbidden: User "system:serviceaccount:torino-tenant:service-torino" cannot
 list resource "deployments" in API group "apps" in the namespace "torino-tenant"
$
```

Figure 7: Retrieve information from inside pod

4 Conclusion

The laboratory enabled us to create a complete and operational cluster with multiple instruments that permit us to operate a multi-tenancy network and working environment. The principal instruments we used are:

- Resource quota, defined by a ResourceQuota object, provides constraints that limit aggregate resource consumption per namespace. When you create a workload management object such as a Deployment that tries to use more resources than are available, the creation of it succeeds, but the Deployment may not be able to get all of the Pods it manages to exist.
- The LimitRange object can enforce constraints related to maximum and minimum resource usage, preventing a single object from monopolizing all available resources within a namespace. A LimitRange is a policy to constrain the resource allocations (limits and requests) that you can specify for each applicable object kind in a namespace.
- Role Based Access Control is a method of regulating access to computer or network resources based on the roles, which represent a set of permission of individual users within your organization.
- Network Policies are an application-centric construct which allow you to specify how a pod is allowed to communicate with various network entities over the network.

During the laboratory, we saw different models and example cases exploiting the techniques explained before, showing differences between them and how to implement them inside a Kubernetes cluster architecture. With the knowledge about these instruments, we can continue this laboratory by implementing them all together and performing correct performance and logical isolation of the tenants based on the customers' needs. All the code and images can be found in my GitHub and Docker Hub defined in the reference page.

References

[1] Matteo Frigo. Resource-limitation. `https://github.com/FrigoMatteo/Resource-Limitation.git`, 2025. Access: May 2025.

[2] Matteo Frigo. Resource-usage-limitation. `https://hub.docker.com/repositories/matteofrigo1618`, 2025. Access: May 2025.

[3] Inc Docker et al. Docker, 2025.

[4] T Kubernetes. Kubernetes. *Kubernetes. Retrieved May*, 24:2019, 2019.

[5] Inc Kubernetes et al. Kubernetes resourcequotas, 2025. Access: may 2025.