MUlti-Slice/Tenant/Service (MUSTS)

1.1. Scope of this document

This document is to describe the purpose of the demonstration and the architecture of the test infrastructure that was used by the NECOS consortium. This test infrastructure is not precluding that any other can be used to run de demonstration. The guide to install the software in the substrate resources is provided in the README file, in the same repository as the software.

1.2. Introduction

MUSTS is the main NECOS demonstration that creates 2 slices using one slice provider hosted by UFG, the marketplace hosted by UFPA, and four resource providers (UFSCar, 5TONIC, UoM and UNICAMP), as presented in **Figure 1** Slices requested by two different tenants are shown through their complete life cycles. Each tenant has specific resource and service constraints that must be supplied and guaranteed by NECOS. More specifically, one tenant runs a Touristic service while the other runs an Internet of Things (IoT) service, which are derived, respectively, from the Network Slicing for Touristic Content Distribution and Network Slicing for Metropolitan Integrated Monitoring scenarios [D2.2].

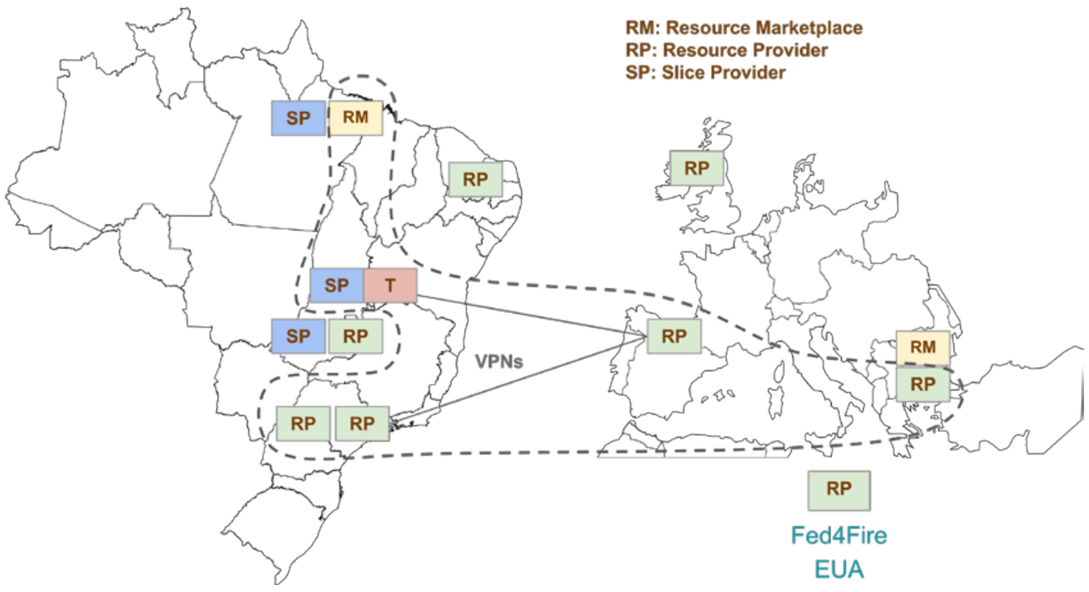


Figure 1. Instantiation of the MUSTS demo on the distributed experimental infrastructure.

1.3. License

All the source code developed within the NECOS project, and made available as OSS, is released under the Apache License – Version 2.0[1]

1.4. Objectives

This demonstration aims at exercising the following key features of NECOS: slice creation, slice decommission, slice monitoring, service deployment, service update, VIM heterogeneity, and elasticity upgrade (both vertical and horizontal). The features slice creation, slice decommission, slice monitoring and service deployment are features exercised in both slices, as they are essential for every slice. However, the feature VIM heterogeneity is exercised only in the Touristic service slice to showcase the capability of NECOS to support different VIM technologies, in our case, Docker and XEN. The elasticity feature is exercised in the IoT service slice, in which we demonstrate the NECOS capability of vertical and horizontal elasticity upgrades.

Touristic Service for End-to-End Slice

The touristic Content Delivery Network (CDN) service is a cloud network slice use that delivers touristic content to users based on their geographic location. The idea is that tourists visiting specific locations (e.g., cities, archaeological sites, museums, churches, or towers) are more likely to request content related to a geographical site.

Assumptions of the touristic CDN scenario include: i) a central Web server hosting all content (videos/web pages), and ii) three edge cloud nodes hosting a single video and a web site related to their geographic location. For instance, the core cloud node could be Brazil-Unicamp, an edge cloud node with Greek Touristic content could be Greece-UOM, an edge cloud node with Spanish content could be 5G-Tonic and another edge cloud with Brazilian content could be Brazil-UFSCar. Touristic content requests (video or web content) from a visitor sightseeing Spain are directed either to the local edge cloud server, in case that the requests are related to Spain (e.g., visiting hours for the Royal Palace of Madrid), or to the core server if his requests are irrelevant to his position (e.g., a YouTube video for healthy eating).

Our implementation approach regarding the aforementioned service aims at highlighting how the CDN services benefit by using the NECOS Platform. More importantly, we demonstrate that CDN technology brings services close to the end-users achieving efficiency not only in terms of performance (e.g., connection time, download time) but also in respect to the processing resources required. For example, in the touristic CDN service, applications running at the network edge are lightweight, and the edge infrastructure computing resources are much less powerful than the ones at the core. What is more, the NECOS Platform facilitates the deployment of such services.

[1] APACHE http://www.apache.org/licenses/LICENSE-2.0

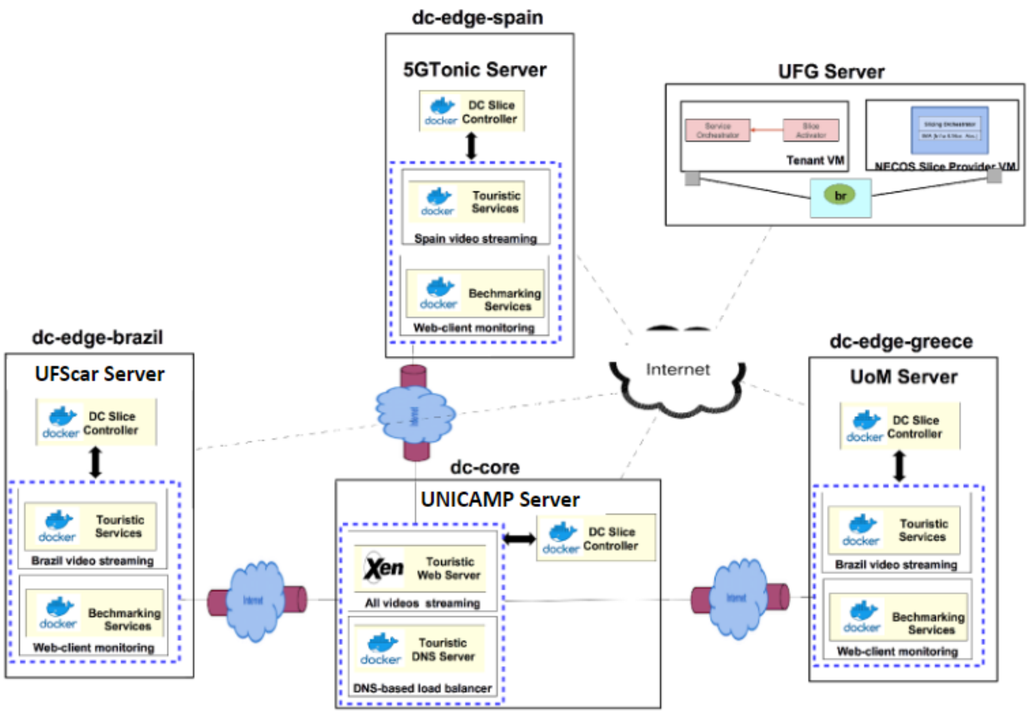


Figure 2. CDN Slice Overview.

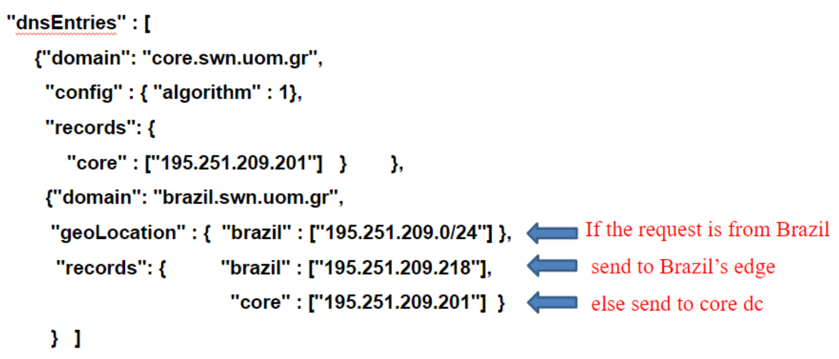
Figure 2 shows the connections among the components in the touristic CDN service. Boxes in blue represent the slice-parts holding the CDN service components. Each slice part in the touristic CDN deployment is composed of a single VM. Such figure also shows the different VIM technologies being used in a geographically distributed slice: the dc-core is using XEN as the VIM and the dc-edges are using Docker. The dc-core slice part accommodates the content services and the Domain Name System (DNS) load balancer is responsible to direct the requests to the appropriate DC slice part (core or edge). As previously mentioned, the requests are shared among the DC slice-parts according to the client's geographic location. The dc-edges host the content services and benchmarking tools (e.g., load testing tools), which are used to test the performance of touristic service.

Experimental setup

Core DC - the applications that are running in the core are the following: the DNS load balancer, an Apache web server which provides all the web pages, VLC video streaming servers, and the Grafana and Influx-db monitoring tools in the touristic Web Server. The inputs in the monitoring tools are the results from the benchmarking tools, which are running on edge cloud nodes.

Edge DC - the services at the edge dc slice parts have been containerized (e.g., Docker). For the web services, we use flask and for the video streaming services we use VLC. The benchmarking load testing tool used is jmeter.

DNS Load balancer - it has a major role in the touristic CDN deployment. The DNS server uses a JavaScript Object Notation (JSON) configuration file (shown in **Figure 3**) which has the current information about domain names and slice parts' IPs. In this section, we present only the part of the JSON file which is used for the touristic CDN scenario for an edge cloud slice (e.g., dc-edge-brazil).



```
"dnsEntries" : [
   {"domain": "core.swn.uom.gr",
    "config" : { "algorithm" : 1},
    "records": {
       "core" : ["195.251.209.201"]  }       },
    {"domain": "brazil.swn.uom.gr",
     "geoLocation" : {  "brazil" : ["195.251.209.0/24"] },   If the request is from Brazil
      "records": {         "brazil" : ["195.251.209.218"],    send to Brazil's edge
                             "core" : ["195.251.209.201"] }   else send to core dc
    }  ]
```

Figure 3. JSON file used for touristic CDN scenario.

The DNS can be reconfigured while it is running so we can easily apply any changes that happened after the deployment (for example the DC's IPs). So particularly, the DNS should know the slice parts' IPs (the IPs are defined in the JSON configuration file), while the core's IP should be added in the edge slice parts' *resolv.conf* file.

1.5. IoT Service

The IoT Demonstration aims to show that NECOS is suitable and also a facilitator in supporting the deployment, management and operation of real IoT solutions; being able to provisioning and monitoring resources, deploying services and scaling slice resources according to load changes. As mentioned above, beyond demonstrating the slice creation, slice decommission, slice monitoring and service deployment, this slice specifically showcases the service update and elasticity upgrade, both vertical and horizontal.

The chosen IoT scenario consists of a real-time cargo monitoring and tracking IoT solution, where a monitoring device with wireless communication and multiple sensors (temperature, humidity, light, and gps) is attached to a cargo container and rides with it all along its journey. This device periodically "wakes-up" and transmits precise monitored data to a centralized system, which allows customers to have visibility of their goods in movement and being notified when something happens with their cargo (door openings, extreme temperature shifts, etc.). This scenario is illustrated by **Figure 4**.

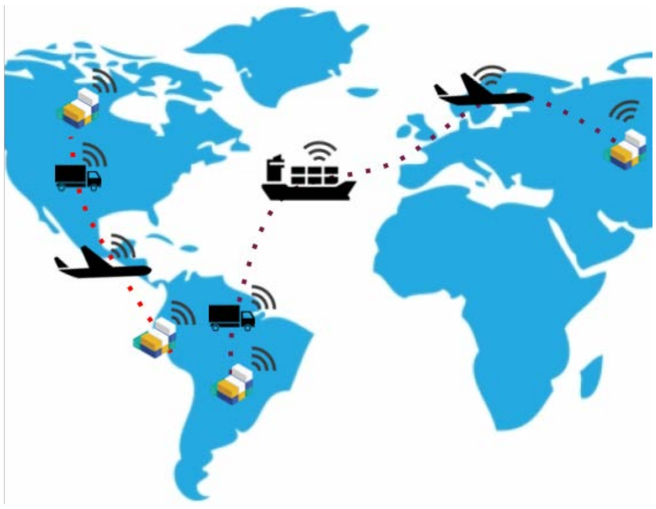


Figure 4. Real-time cargo monitoring and tracking.

The IoT demonstration was built using Dojot, an open source IoT Platform whose development is led by CPqD, and a load testing tool for Message Queuing Telemetry Transport (MQTT) IoT devices, which was customized for this demonstration. The software components were deployed, initially, in two slice parts as depicted in **Figure 5**.

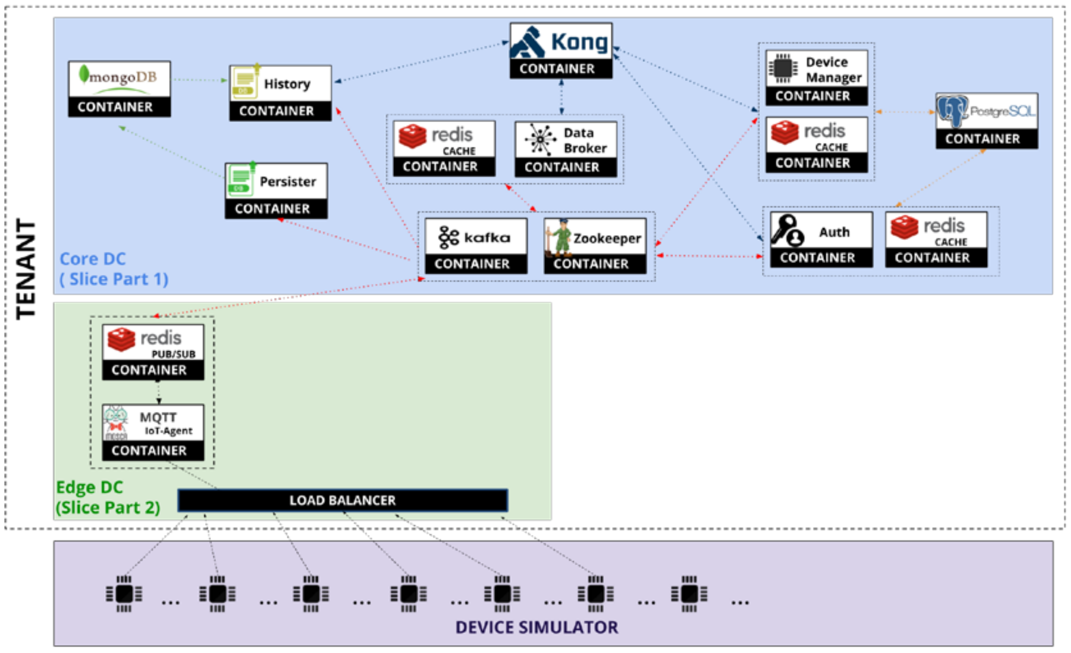


Figure 5. IoT Demonstration setup based on Dojot micro services.

The Core DC (slice part 1) runs the cloud Dojot micro services, which are responsible for: managing the IoT devices life cycle, storing telemetry data, and providing Representational State Transfer (REST) and socket.io interfaces to retrieve, respectively, historical and real time data about the devices. In this Demonstration the slice part 1 is a set of three VMs managed by the Kubernetes VIM (one VM hosting the master node and the other two representing the worker nodes). The Dojot micro services are deployed in the worker nodes.

The Edge DC (slice part 2) runs the edge Dojot micro service, called MQTT IoT-Agent, which is responsible for establishing connections with the IoT devices and transform the data to be transmitted to the cloud micro services. In this Demonstration the slice part 2 is a set of two VMs managed by the Kubernetes VIM (one VM hosting the master node and the other representing the worker node). The edge Dojot micro service is deployed in the worker node.

The device simulator runs outside the NECOS infrastructure and simulates connected IoT devices publishing telemetry data according to some input settings.

Figure 6 shows the IoT Slice overview. The Core DC is at Unicamp, Campinas, and the Edge DC is at 5TONIC, in Madrid. Note that for horizontal elasticity upgrade, a new slice part is created at UFSCar, Sorocaba.

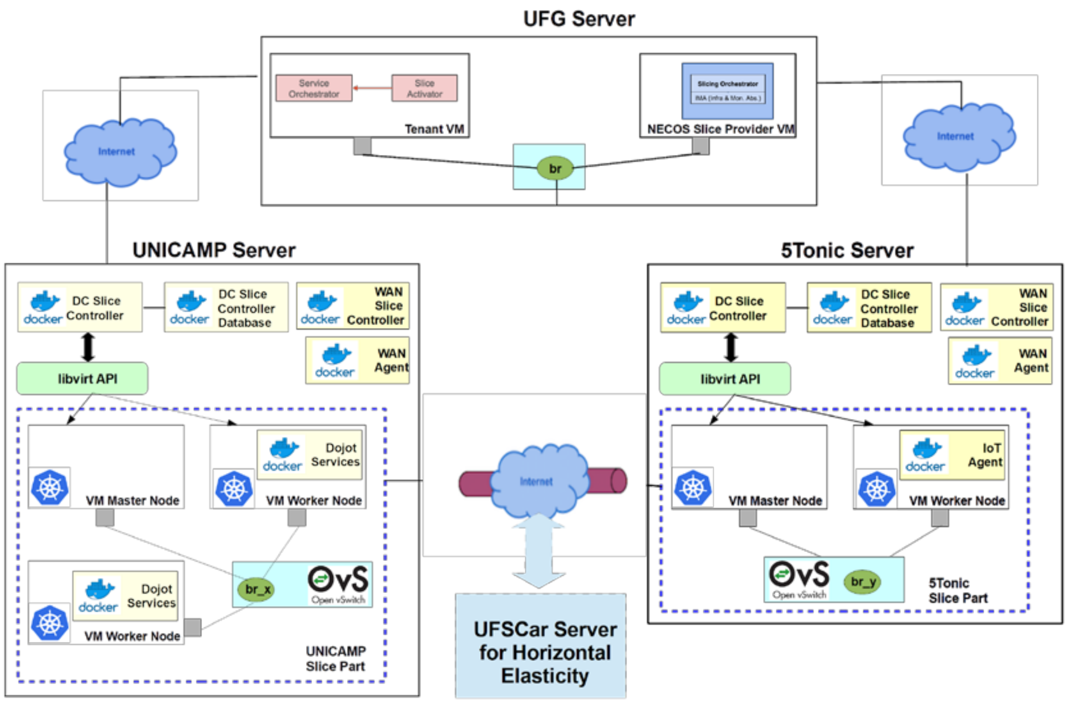


Figure 6. IoT Slice overview.

The major flow of the telemetry data inside the platform is described in **Figure 7** with each step identified by a red circle with a number inside. A detailed description of each step is given below:

1) The simulated device opens an MQTT connection through a TCP Load Balancer, publishes some telemetry data, and closes the connection;
2) The TCP Load Balancer redirects the connection/telemetry data to an instance of the MQTT IoT-Agent;
3) The MQTT IoT-Agent authorizes the device connection and sends the telemetry data to the REDIS DB, responsible for implementing the PUB/SUB of the MQTT Broker, and also sends it with some extra metadata to the Apache Kafka, responsible for redistributing it to the other dojot micro services;
4) The Persister micro service consumes the telemetry data from Kafka and stores it into the MongoDB;
5) The DataBroker consumes the telemetry data and provides it through socket.io to the registered clients.

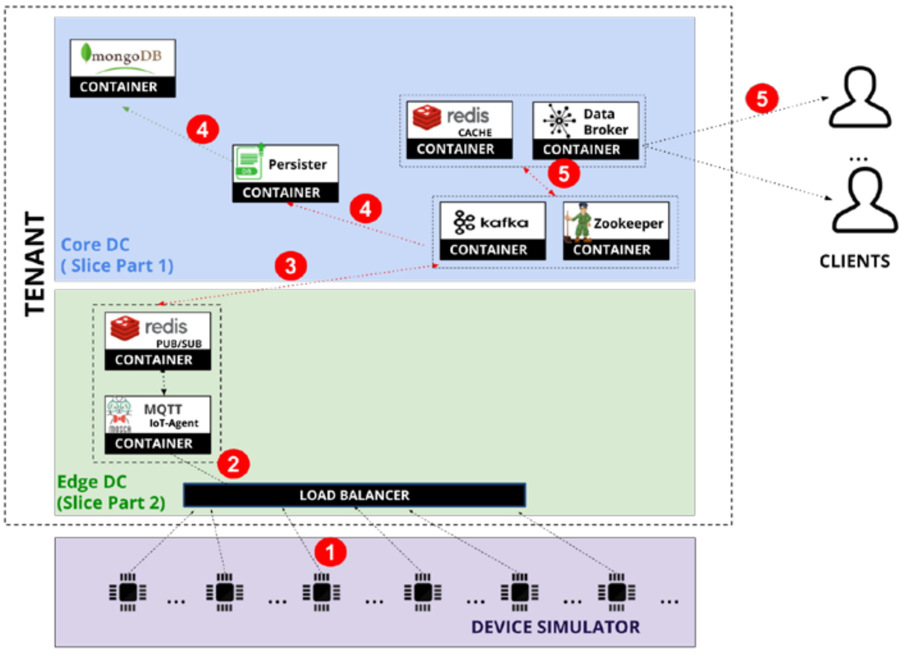


Figure 7. Dojot data flow.

An increasing number of cargo containers was simulated, publishing telemetry data (temperature, humidity, lightness, gps) with a given periodicity. When the machine hosting the IoT Agent in the edge was overloaded due to an increase in the number of requests to the IoT Agent, elasticity took place to avoid a degradation of the quality of the service (connections rejections, messages losses and long response time). First of all, NECOS would try to do a vertical elasticity upgrade, which means to add a new machine to host another IoT Agent in the same slice part. Specifically, in our case NECOS will make vertical elasticity in the Edge DC (slice part 2). Then, after making vertical elasticity, the service is redeployed so that the new machine receives the new IoT Agent and the requests coming from the sensors are now balanced between both IoT Agents.

A second experiment that was done is related to horizontal elasticity. In this case, NECOS tries to make vertical elasticity in the Edge, but there are no more resources in that slice part. Then, SRO triggers the horizontal elasticity upgrade which means to add a new slice part to the existent slice. Specifically, for this experiment, a new slice part at UFSCar is created. After the SRO receives the return about the creation of the new slice part, it triggers the service update (redeployment) so that the new slice part is used by the Dojot service. After the service redeployment, requests coming from the sensors are balanced between the IoT Agent at 5TONIC (the overloaded slice part) and UFSCar.

1.6. Workflow

Since this demonstration is the one which exercises the majority of the NECOS features and follows the working flows defined in D5.1 and D5.2, the working flow presented in **Figure 8** is depicted in a higher level of detail.

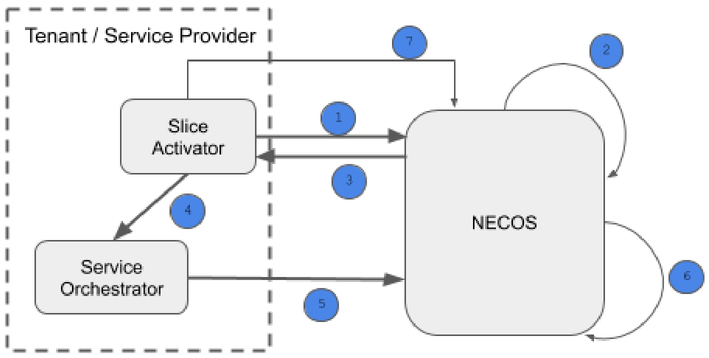


Figure 8. MUSTS demonstration workflow.

- Step 1: The tenant calls the NECOS system to create a slice. In this call, the tenant must inform the specification of the slice so that NECOS will be capable of building it;
- Step 2: This step, abstracted as a single step in this figure, follows the complete slice creation working flow presented in D5.2. In this step, actions such as looking for candidate slice parts in the Marketplace and start the monitoring of the slice are performed;
- Step 3: The NECOS system returns to Tenant all the details about the slice that was created, including information about every slice part, physical resources, location, etc.;
- Step 4: The Tenant calls its Service Orchestrator so that it can embed into the slice the service to be run. The Service Orchestrator can be a very smart engine or a very simple solution (even performed by hand by an operator) to decide where to put every service inside the slice;
- Step 5: Service Orchestrator triggers the deployment of the service. NECOS is responsible for parsing the YAML file received and dispatch specific commands in each slice part to deploy the correspondent service inside the slice;
- Step 6: This step represents the elasticity. Specifically, for this Demonstration, it refers to the vertical and horizontal elasticity upgrade, which means adding a new resource inside a slice part or adding a new slice part, respectively. This is done when a defined threshold is reached. For this Demonstration, the SRO is observing the CPU load of the machine where the IoT Agent is running. When the load reaches 80%, the elasticity is triggered. Again, as with step 2, the elasticity upgrade (both vertical and horizontal) abstracted here in a single step, is fully described in D5.1;
- Step 7: This final step represents the slice decommission call done by the Tenant to delete the slice as well as the service running inside it.