Containers & Serverless

High Performance Computing
(Computació d'Altes Prestacions)

Josep Lluís Berral-García – josep.ll.berral@upc.edu
Jordi Torres Viñals – jordi.torres@upc.edu

Introduction

"Encapsulate your environments and applications…

…to better replicate, migrate and deploy anywhere"

Session Objectives

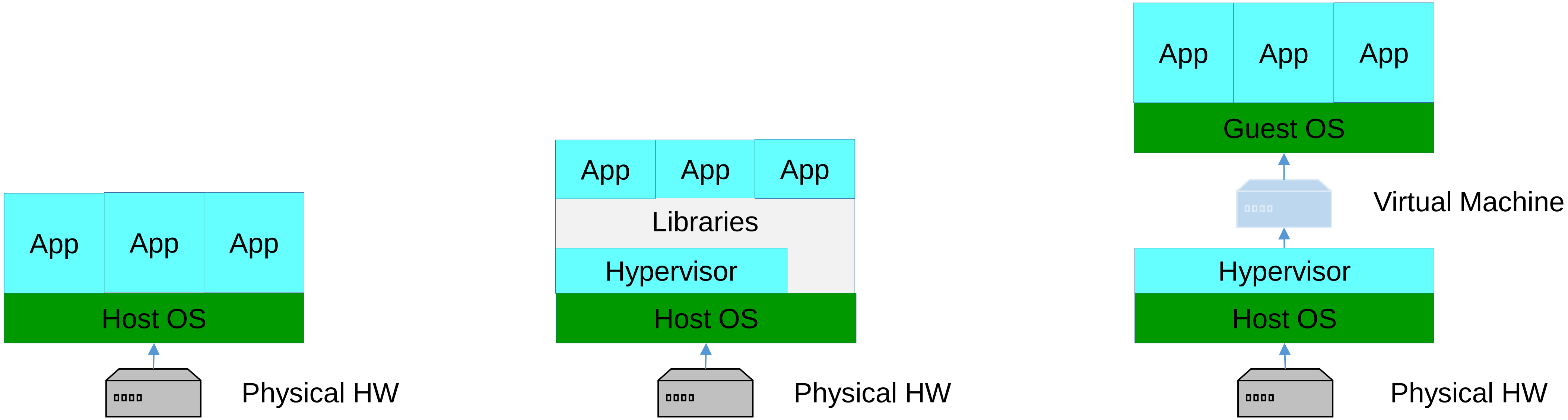
- Explain the difference of containerized applications vs. VMs
- Design a container deployment schema for our AI applications
- Make a list of 5 advantages of containers vs. VMs and raw systems

Containers and Supercomputers

CONTAINERIZATION

Isolation and Environments

- Containerization of Services
  - In between "as is" and "virtualization"



Running in Host

Running in Containers

Running in VMs

Isolation and Environments

- Isolation of Services
  - In between running "as is" and full system virtualization
    - No HW is virtualized/emulated
    - The Host OS and HW are mostly used
    - An environment is deployed for a set of tasks → No persistence expected
  - Hypervisor
    - Crafted to share parts of the OS with the container instances
    - Potential security issues → No full isolation (as in VMs)
    - Reduce weight of container instances

Containers Life-Cycle

Containers ~ isolation of environments as a process, oriented to scenarios of “create-use-destroy”

- Virtual Machines
  - An instance (from an “image”) and disk volume exist per VM
  - Modifications are persistent: changes remain

- Containers
  - An “image” is to be instanced (deployed, “container”) as many times as wanted
  - When a container finishes, it is not re-used

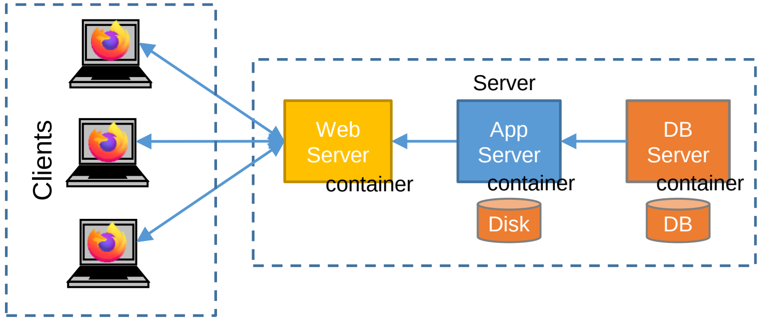
Containers Life-Cycle

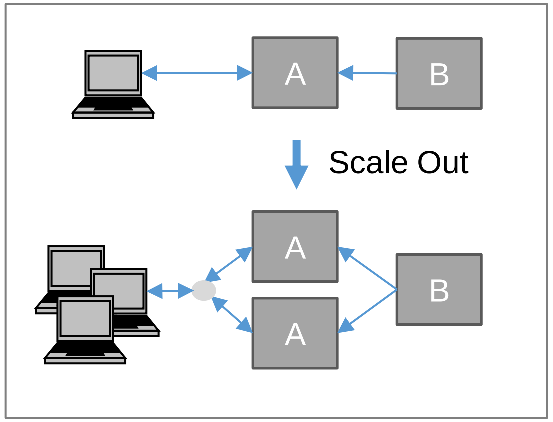
Containers ~ isolation of environments as a process, oriented to scenarios of "create-use-destroy"

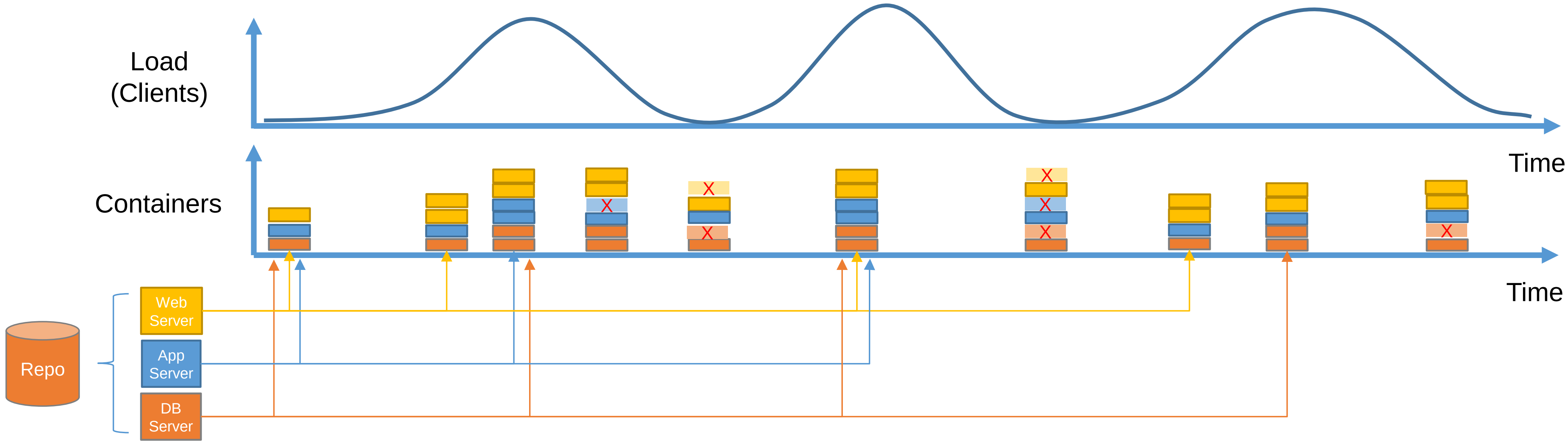
- Virtual Machines
  - Life-cycle:
    - A VM "instance" is created from an "image" + virtual disk volume exist
    - The "instance" (VM) is started and used
    - Changes in the VM persist
    - The "instance" can be stopped and resumed
    - The "instance" is destroyed when is not needed anymore
- Containers
  - Life-cycle:
    - An "instance" is to be instanced (deployed) as many times as wanted
    - Each "instantiation" → a "container"
    - At the end of its use, the "instance" is finished
    - When a container finishes, it is not re-used
    - Changes in the container are not stored

Containers Life-Cycle

- Example for a Web-Service







Containers Life-Cycle

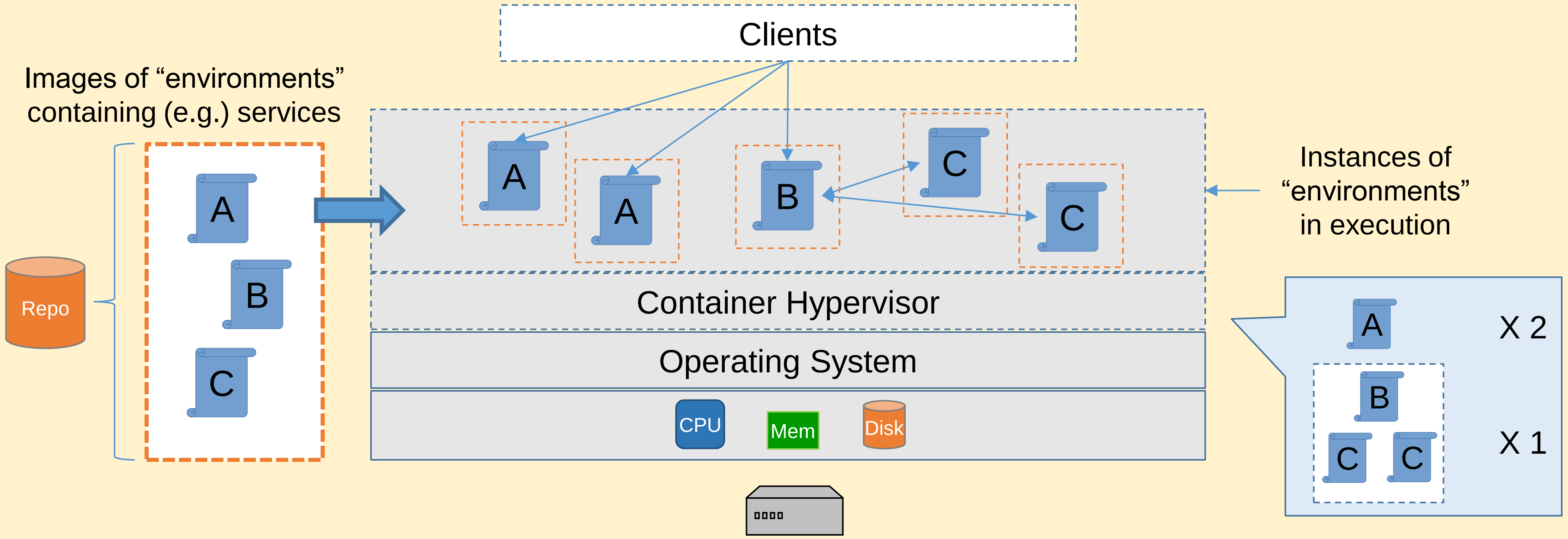
- In the example:
  - We have 3 "servers"
    - Web-server (provides data in the form of "web pages", e.g. Apache)
    - App-server (executes Application scripts, e.g. Python script)
    - DataBase-server (manages data, e.g. MySQL)
  - Users interact with the web-server, that asks for computing data to the app server, that requires data from the DB.
    - Example: Jupyter Notebook (has an interface → served by the web-server; has an application → python session running your scripts; has a DB → in this case the file-system storing notebooks and python libraries)
  - Each server can hold X clients maximum
    - We create several instances of each server, to cover capacity
    - Each server instance is identical to the others
    - All data is stored in the filesystem/database
    - When the number of clients rise → we start more instances; when decrease → we finish instances

Containers Life-Cycle

- Blackboard Exercise
  - What if we have a ML pipeline?
    - An "app" A brings data from users
    - An "app" B trains a ML model from the last batch
    - An "app" C predicts data from A using the last ML model
  - Requirements
    - App A might suddenly double/triple/quadruple data input ratio N
    - App B requires X time to process a batch of size N
    - App C requires predict data A at "real-time" (maximum 1 batch in queue)

Container Architecture

- Another example of Encapsulation of Services



  – Allow to deploy “instances” of services, quickly, orchestrated and in isolation

Container Architecture

- In the example:
  - We have "images" of containers {A, B, C}
    - Each container has installed different environments and applications
    - Instances of "Container A" provide a service to the clients just alone
    - Instances of "Container B" provide a service to the clients, but require service from an instance of "Container C"
  - Request from the service Provider (person that owns A, B and C)
    - Asks que Platform/Infrastructure Provider a deployment
    - The deployment asks for
      - 2 deployments of A
      - 1 deployment containing 1 instance of B connected to 2 instances of C
    - Additionally (not in the picture):
      - The requests from clients to A are balanced between the 2 instances of A
      - The requests from B to C are balanced between the 2 instances of C
      - That way, neither A or C are overloaded → If requested, more instances of A, B and C cam be instanced

Custom Containers

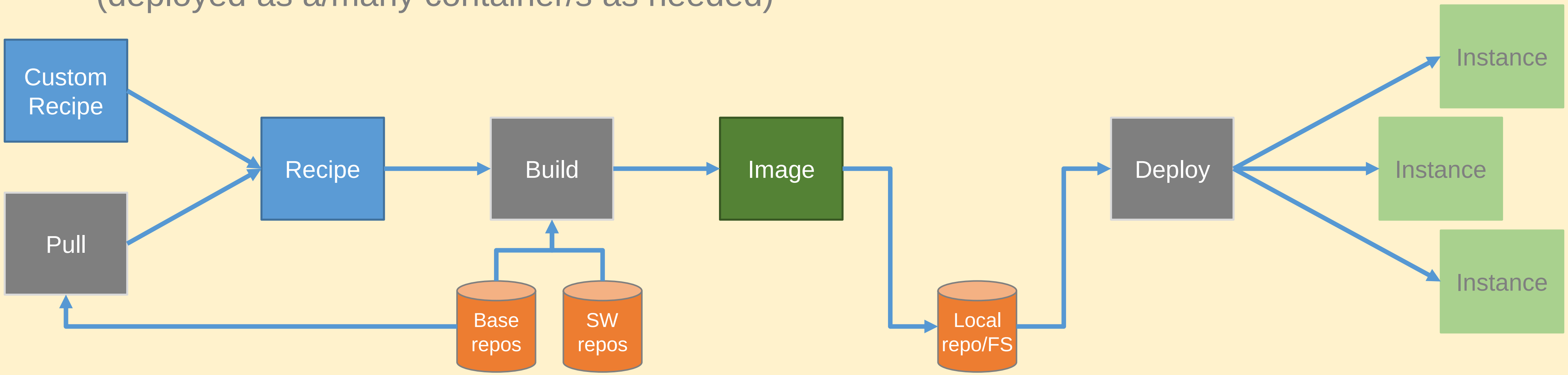
- The "Recipe":
  - Instructions to "build" an image
    - "image base" + additional installs + copy files + environment variables
    - The result is an image ready to be instantiated

- Pipeline:
  - From "recipe" → build "image" of container
  - From "image" → deploy "instances" of container
  - From "image" + "receipe" → build new version of "image"

Custom Containers

- The "Recipe":
  - Instructions to "build" an image
    - Contains the instructions to download an "image base"
    - … plus the instructions to install additional programs
    - … pus copy files from our system inside the container
    - … plus do any other modification
  - E.g.: Docker → the "dockerfile", Singularity → the "singularity file"

  - The result is an image ready to be instantiated
    - (deployed as a/many container/s as needed)



Custom Containers

- Example of Container Descriptor
  - In this case, Singularity container for SPARK and R

```
BootStrap: shub
From: nickjer/singularity-rstudio

%labels
 Maintainer Josep Ll. Berral
 Spark_Version 2.4.7
 Hadoop_Version 2.7
 BSC_Nord3 1.0

%help
 This will run Apache Spark with an RStudio Server base, adapted to MN-IV

%runscript
 exec spark-class "${@}"

%environment
 export SPARK_HOME=/nord3/spark
 export PATH=${SPARK_HOME}/bin:${PATH}
 export JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-amd64
 export J2REDIR=${JAVA_HOME}/jre/
 export J2SDKDIR=${JAVA_HOME}

 export JAVA_BINDIR=${JAVA_HOME}/bin
 export SDK_HOME=${JAVA_HOME}
 export JDK_HOME=${JAVA_HOME}
 export JRE_HOME=${JAVA_HOME}/jre/
 export JAVA_ROOT=${JAVA_HOME}

%post
 # Software versions
 export SPARK_VERSION=2.4.7
 export HADOOP_VERSION=2.7
 export SPARK_MIRROR=http://mirror.cc.columbia.edu/pub/software/apache/spark

 # Install Spark
 apt-get update
 apt-get install -y --no-install-recommends openjdk-8-jre
 if [ ! -d ${SPARK_HOME} ]; then
  mkdir -p ${SPARK_HOME}
  wget --no-verbose -O - "${SPARK_MIRROR}/spark-${SPARK_VERSION}/spark-${SPARK_VERSION}-bin-hadoop${HADOOP_VERSION}.tgz" | tar xz --strip-components=1 -C ${SPARK_HOME}
 fi

 # MareNostrum Mount points
 mkdir -p /gpfs/home
 mkdir -p /gpfs/projects
 mkdir -p /gpfs/archive
 mkdir -p /gpfs/scratch
 mkdir -p /gpfs/apps
 mkdir -p /scratch
 mkdir -p /.statelite/tmpfs/gpfs/projects
 mkdir -p /.statelite/tmpfs/gpfs/scratch
 mkdir -p /.statelite/tmpfs/gpfs/home
 mkdir -p /.statelite/tmpfs/gpfs/apps/MN3

 # Install sparklyr: R interface for Apache Spark
 Rscript -e "withCallingHandlers(install.packages(c('sparklyr'), repo= 'https://cran.rstudio.com/', clean = TRUE ), warning = function(w) stop(w))"

 # Clean up
 rm -rf /var/lib/apt/lists/*
```

Containers & Supercomputers

- Privileges and User-mode
  - Docker [1] → Can scale privileges (do things at root/kernel level)
  - Singularity [2] → Runs under user space and privileges

- In controlled environments…
  - … like the MareNostrum Supercomputer…
  - … all tasks must be contained into the user environment

[1] Docker Container: https://www.docker.com/resources/what-container/
[2] Singularity Container: https://docs.sylabs.io/guides/latest/user-guide/

Containers & Supercomputers

- Privileges and User-mode
  - Docker [1] → Can scale privileges
    - If the user instancing the container has privileges, the container can reach them
    - If the user do not have them, but the hypervisor and OS allow it, the container also can
    - This means, e.g., do things at root/kernel level
  - Singularity [2] → Runs under user space and privileges
    - The container only has the privileges of the user calling
    - And "root" is usually forbidden
- In controlled environments
  - …like the MareNostrum Supercomputer
    - You cannot scale privileges
    - All tasks are contained into the user environment
    - We cannot use Docker in MN-IV, but we can create a Singularity container

[1] Docker Container: https://www.docker.com/resources/what-container/
[2] Singularity Container: https://docs.sylabs.io/guides/latest/user-guide/

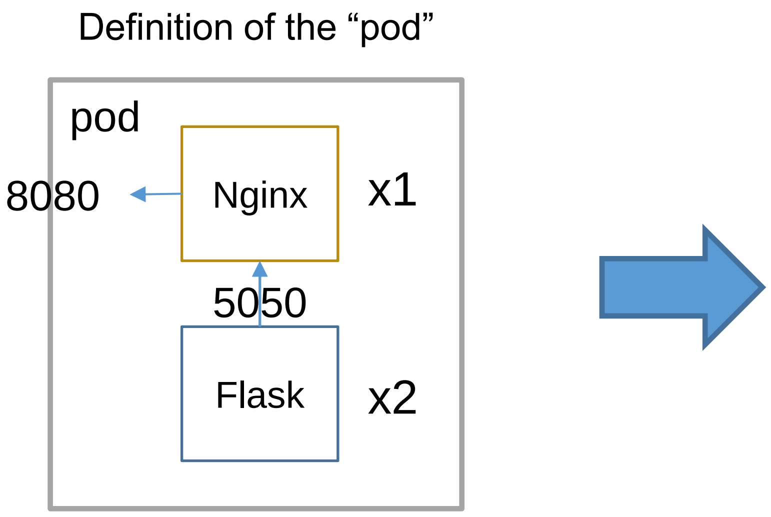
Containers & Supercomputers

- Singularity / "User-mode" containers
  - Created (originally) for scientific environments
  - Clusters and supercomputers without privilege from users
  - Compatible with "recipes" from Docker (the most popular container hypervisor)

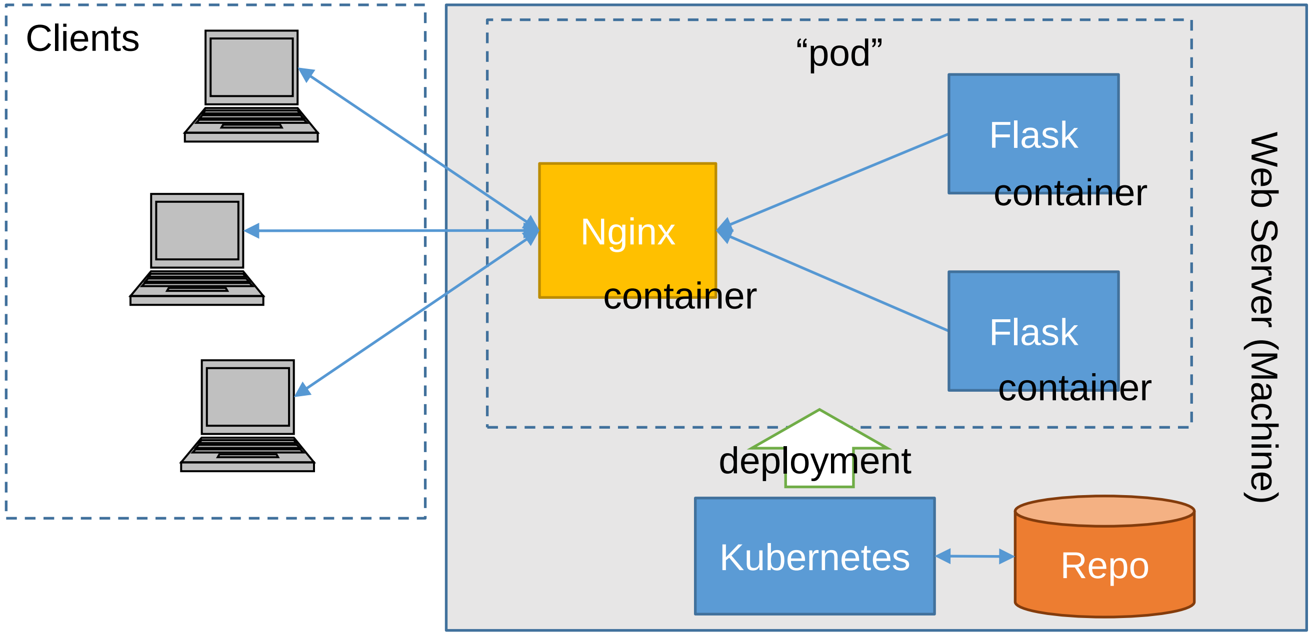
  - Difficult to create complex deployments automatically
    - Virtual network deployments (e.g. like in Docker-Compose or Kubernetes)
    - Access to devices that require special privileges
    - … but good enough to deploy AI applications and science

"Pods" & Reunion of Services

- Grouping services in "Pods"
  1. Define the architecture of applications
  2. Encapsulate in "pods"
  3. Deploy in one shot



- Nginx → Web Server
- Flask → App Server

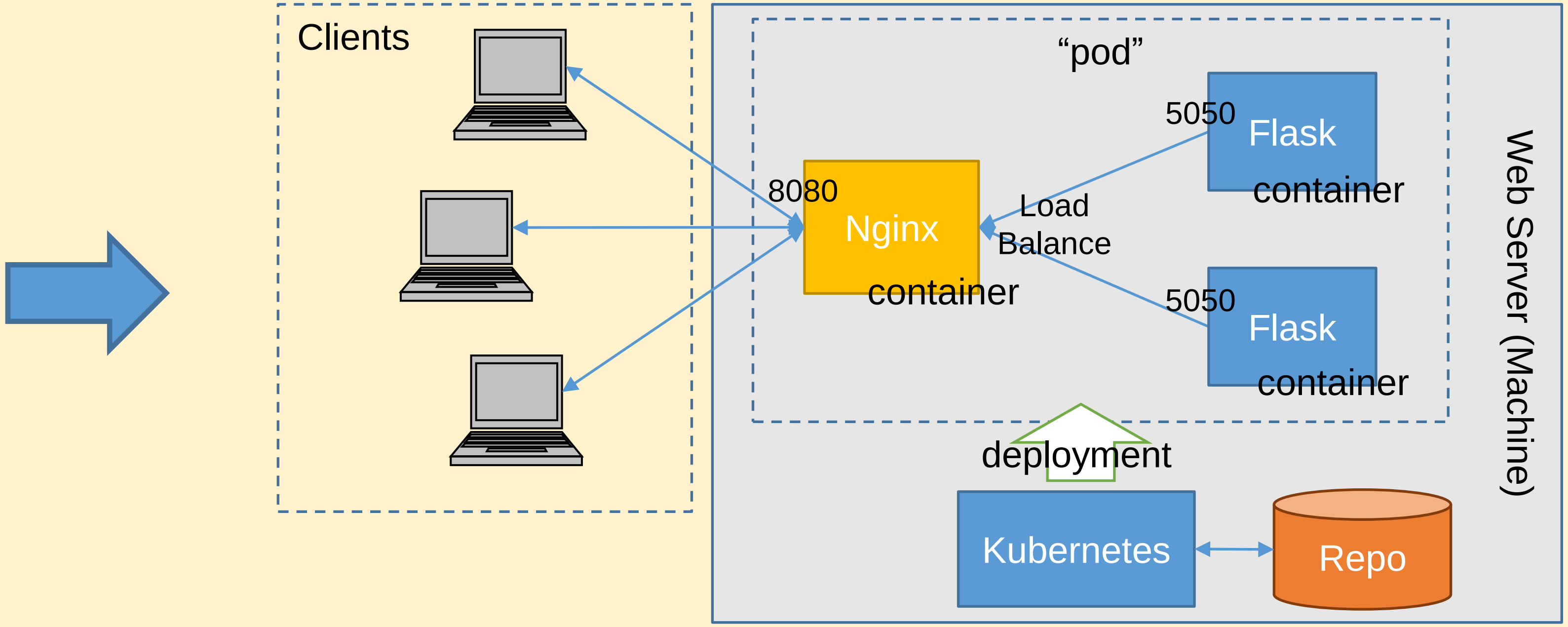


"Pods" & Reunion of Services

- Grouping services in "Pods"
  - Different frameworks: e.g. Kubernetes, Docker-compose, etc…
  - Infrastructure/Platform/Service described
    - A "recipe" for the whole deployment
  - Steps:
    - Define the elements (containerized services) in the "pod"
    - Indicate how they connect and how they are exposed
    - Allocate & manage available resources for the pods/containers
    - Deploy ALL in one shot

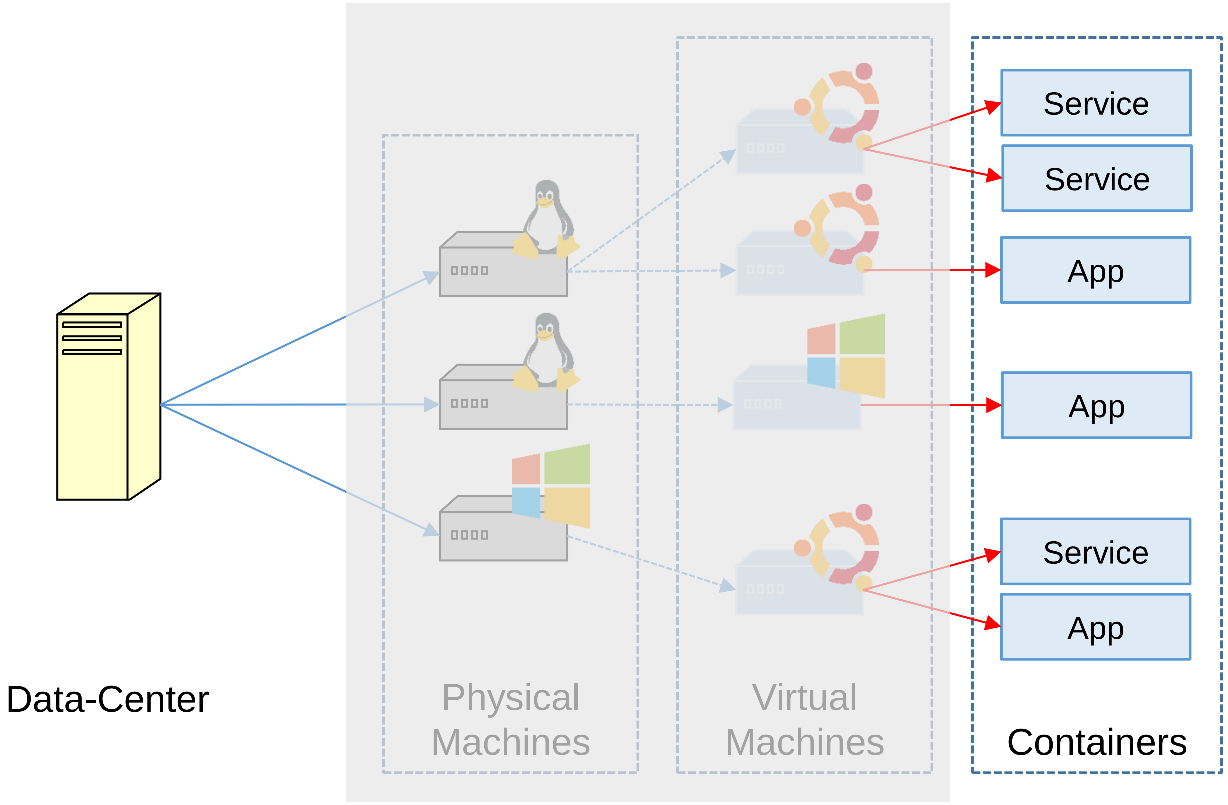
Pod description:

- 1 x Nginx → Web Server
  - Available in port 8080
- 2 x Flask → App Server
  - Application: xxxxx.py
  - Available in port 5050
- Connect
  - Nginx with Flask
  - Load balance Flask



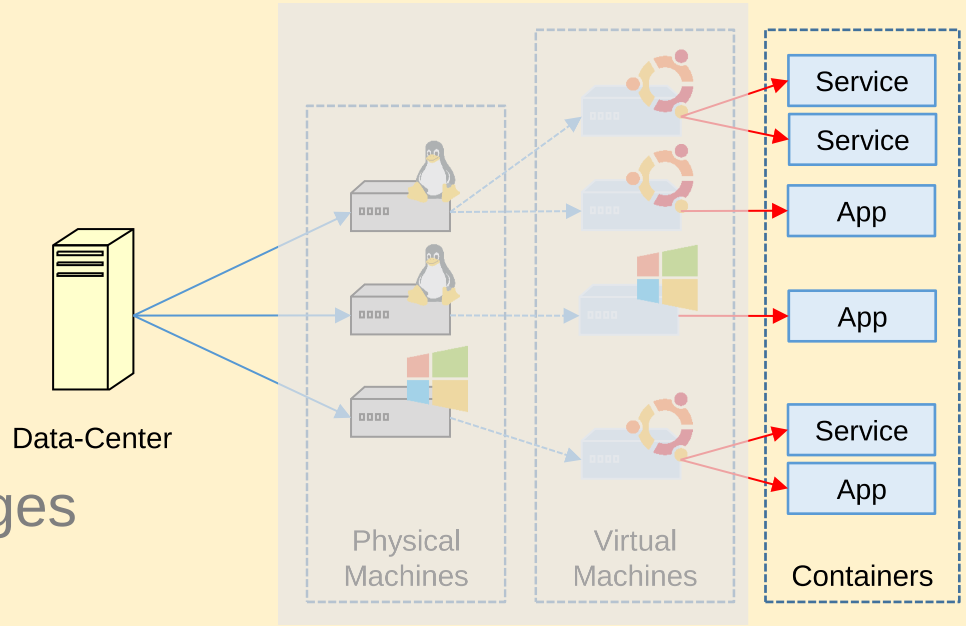
Virtualization in Resource Providers

- Resource providers can offer containers as a service
  - (inside VMs, to keep control of users and enforce security)



Virtualization in Resource Providers

- Resource providers → "Cloud owners"
  - VMs (IaaS) or Docker / Singularity / Kubernetes … (PaaS)
    - To launch containers in scalable way
  - Runs in "transparent" platform of VMs
    - Users provide the container recipes or images
    - The manager checks the instances running
    - User does not care about what's below



Functions as a Service

SERVERLESS COMPUTING

Serverless Computing

- Serverless
  - Note: Actually… there's a "server"…
  - Micro-functions (FaaS)
    - Application logic (code) to process data
    - Common languages (Java, PHP, Python, Go, Ruby, …)
    - Environment: prepared container with language interpreter

Serverless Computing

- Serverless
  - There's a server that receives "micro-functions"
    - Applications, mostly ETL style, that can be put to run to process massive data
    - Function as a Service
  - Micro-functions
    - User uploads the code, to process batches of data
    - The code indicates which data to retrieve and where to store the result
    - Common languages (Java, PHP, Python, Go, Ruby, …)
    - Environment: prepared container with language interpreter

- Examples
  - Technology providers
    - AWS Lambdas
    - Google Cloud Functions, Firebase (DBs)
    - Azure Data Lake (for DBs and data-warehouses, with SQL queries)
  - Example technologies
    - OpenFaaS, Knative (Kubernetes), Lithops

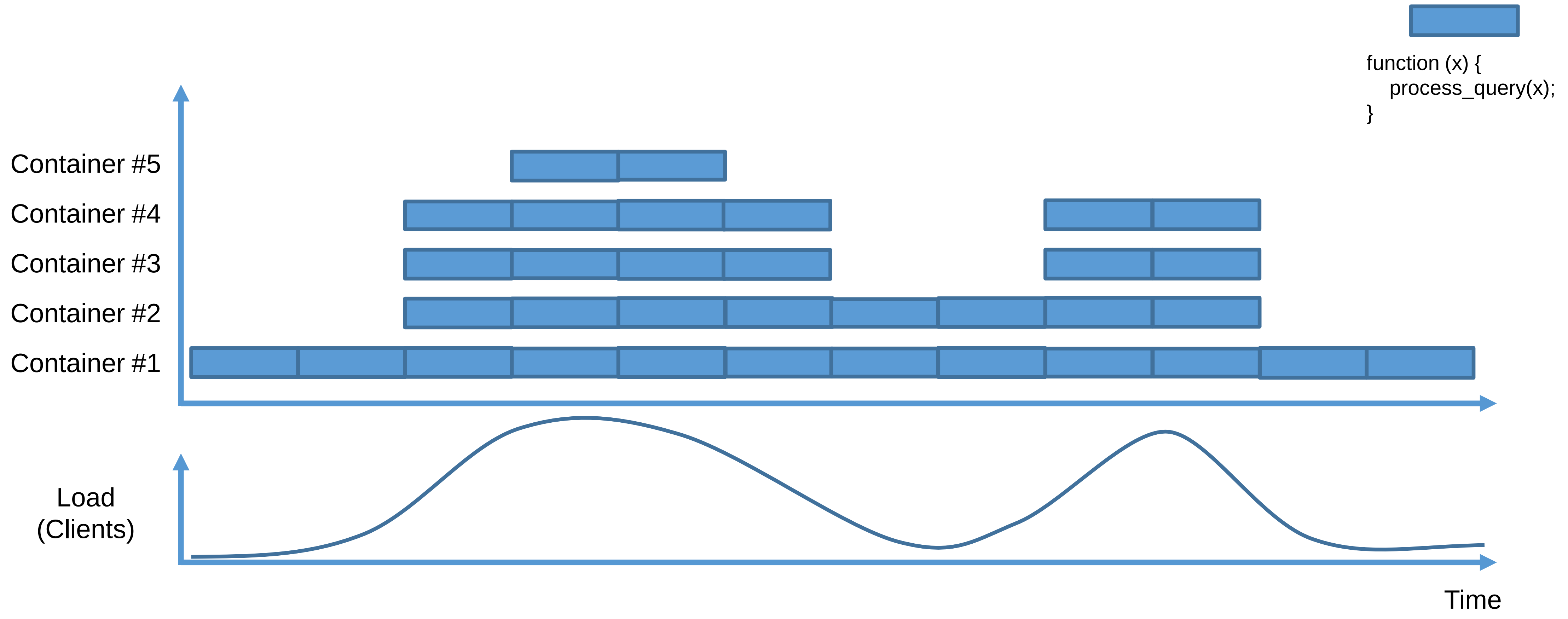
Serverless Computing

- Resources consumption
  - Micro-function only consumes resources when running
  - Micro-function triggers when required only

- Scalability
  - "Infinite" scalability → Elasticity → "Deploy as many needed"
  - <u>Not oriented for HPC</u>, but more towards HPDA / Big-Data

Serverless Computing

- E.g., deploying µ-functions on containers from load



Serverless Computing

- Resources consumption
  - Micro-function
    - Only consumes resources when running
    - Triggers when required only, kills itself when finished
    - Has as input the data to process, and output where to store it
- Scalability
  - "Infinite" scalability → Elasticity → "Deploy as many needed"
    - Batching of data
    - Every function is just a consumer
  - Not oriented for HPC, but more towards HPDA / Big-Data
    - We can apply map-reduce techniques
- In the example
  - Depending on the load (clients or volume of data), the system triggers more or fewer micro-functions
  - For HPC / ETL applications
    - The platform deploys micro-functions depending on the batches in queue

Session Objectives

- Explain the difference of containerized applications vs. VMs
- Design a container deployment schema for our AI applications
- Make a list of 5 advantages of containers vs. VMs and raw systems

Laboratori 2 – Contenidors

PRÀCTICA DE CONTENIDORS

Laboratori

- Entorn:
  - Singularity → Hypervisor (gestiona i executa els contenidors)
  - Repositori de Singularity → Repositori públic de contenidors ja configurats

- Sistema Operatiu:
  - VM amb Ubuntu 19.04
    - Inicialment ho executarem tot dins una VM amb Singularity pre-instal·lat
    - A la pràctica tindrem 2 nivells d'imbricació (màquina → VM → contenidor → App)
  - MareNostrum-IV
    - Desplegarem el que fem a la nostra VM (contenidors) al Supercomputador

- Qüestionari:
  - Durant la pràctica cal resoldre preguntes respecte el que estem executant i observant

Laboratori

- Tenir llest l'entorn de Virtualització
  - Preparar una VM amb VBox per tenir un entorn base llest
- Instal·lar l'entorn de Contenidors
  - Instal·lar Singularity per poder executar contenidors
  - Provar a descarregar una imatge genèrica i fer-la funcionar
- Desplegar contenidors customitzats
  - Preparar una imatge "custom" que contingui pipelines de ML
  - Instanciar la imatge diverses vegades i connectar els contenidors
- Desplegar contenidors al Supercomputador
  - Copiar la imatge de contenidor creada a MareNostrum-IV
  - Preparar un llançador per a executar la pipeline de contenidors
  - Instanciar els contenidors de la pipeline com un treball de MN-IV