Introduction to the eFlows4HPC software stack and HPC Workflows as a Service methodology

Jorge Ejarque (BSC), Jedrzej Rybicki (JSC), Rosa M Badia (BSC)

ISC-HPC, Hamburg May 21st, 2023

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955558. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Germany, France, Italy, Poland, Switzerland, Norway. MCIN/AEI/10.13039/501100011033 and the European Union NextGenerationEU/PRTR (PCI2021-121957)

Agenda

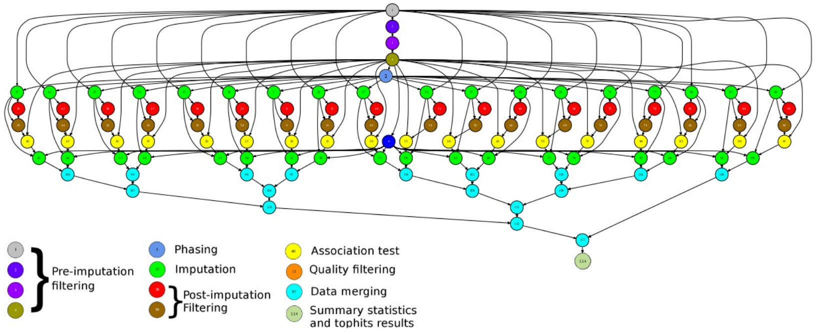
9:00 – 9:15	Overview of eFlows4HPC project and tutorial agenda	Rosa M Badia (BSC)
9:15 - 9:35	Part 1.1: Integrating different computations in PyCOMPSs	Rosa M Badia (BSC)
9:35 – 10:05	Part 1.2: HPC ready container images	Jorge Ejarque (BSC)
10:05 - 10:35	Part 1.3: Data Pipelines and Data Logistics Service (DLS)	Jedrzej Rybicki (JSC)
10:35 - 10:55	Part 1.4: TOSCA Orchestration and HPCWaaS	Jorge Ejarque (BSC)
10:55 – 11:00	Conclusion of part 1	Rosa M Badia (BSC)
11:00 - 11:30	Coffee break	
11:30 - 12:05	Part 2.1: Hands-on session: How to build HPC Ready containers	Jorge Ejarque (BSC)
12:05 - 12:30	Part 2.2: Hands-on session: How to move data with the DLS	Jedrzej Rybicki (JSC)
12:30 - 12:45	Part 2.3: Video demonstrating deployment with Allien4Cloud	
12:45 - 13:00	Tutorial conclusions	all presenters

PART 1.1: INTEGRATING DIFFERENT COMPUTATIONS IN PYCOMPSS

Main element: Workflows in PyCOMPSs

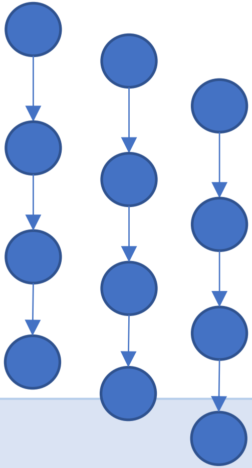
- Sequential programming, parallel execution
- General purpose programming language + annotations/hints
  - To identify tasks and directionality of data
- Builds a task graph at runtime that express potential concurrency
- Tasks can be sequential and parallel (threaded or MPI)
- Offers to applications the illusion of a shared memory in a distributed system
  - The application can address larger data than storage space: support for Big Data apps
  - Support for persistent storage
- Agnostic of computing platform
  - Enabled by the runtime for clusters, clouds and container managed clusters

```
@task(c=INOUT)
def multiply(a, b, c):
    c += a*b
```



PyCOMPSs syntax

- Use of **decorators** to annotate tasks and to indicate arguments directionality
- Small API for data synchronization



Tasks definition

```
@task(c=INOUT)
def multiply(a, b, c):
    c += a*b
```

Main Program

```
initialize_variables()
startMulTime = time.time()
for i in range(MSIZE):
    for j in range(MSIZE):
        for k in range(MSIZE):
            multiply (A[i][k], B[k][j], C[i][j])
compss_barrier()
mulTime = time.time() - startMulTime
```

Synchronization

- Main program and tasks do not share the same memory spaces
- The synchronization compss_wait_on waits for tasks generating the parameter to be finished and moves the data from the remote node to the node where the main program is executed:

```
a = compute (b)
#compute is a task, here we can not check the value of a
...
a = compss_wait_on (a)
#here we can check the value of a
if a:
    ...
```

- Tasks can be also synchronized with a barrier

```
startMulTime = time.time()
for i in range(SIZE):
    compute (A[i], B[i])
compss_barrier()
mulTime = time.time() - startMulTime
```

Other decorators: Tasks’ constraints

- Constraints enable to define HW or SW features required to execute a task
  - Runtime performs the match-making between the task and the computing nodes
  - Support for multi-core tasks and for tasks with memory constraints
  - **Support for heterogeneity on the devices in the platform**

```
@constraint (MemorySize=6.0, ProcessorPerformance=“5000”, ComputingUnits=“8”)
@task (c=INOUT)
def myfunc(a, b, c):
    ...
```

```
@constraint (MemorySize=1.0, ProcessorType =”ARM”, )
@task (c=INOUT)
def myfunc_other(a, b, c):
    ...
```

Failure Management

- Interface than enables the programmer to give hints about failure management

```
@task(file_path=FILE_INOUT,
on_failure='CANCEL_SUCCESSORS',
time_out='$task_timeout')
def task(file_path):
    ...
    if cond :
        raise Exception()
```

- Options: RETRY, CANCEL_SUCCESSORS, FAIL, IGNORE
- Implications on file management:
  - I.e., on IGNORE, output files: are generated empty
- **Possibility of ignoring part of the execution of the workflow, for example if a task fails in an unstable device**
- **Opens the possibility of dynamic workflow behaviour depending on the actual outcome of the tasks**

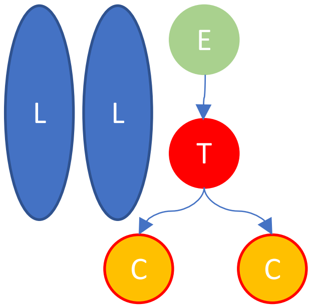
- Tasks can raise exceptions

```
@task(file_path=FILE_INOUT)
def comp_task(file_path):
    ...
    raise COMPSsException("Exception raised")
```

- Combined with groups of tasks enables to cancel the group of tasks on the occurrence of an exception

```
def test_cancellation(file_name):
    try:
        with TaskGroup('failedGroup'):
              long_task(file_name)
              long_task(file_name)
              executed_task(file_name)
              comp_task(file_name)
              cancelledTask(FILE_NAME);
              cancelledTask(FILE_NAME)

except COMPSsException:
        print("COMPSsException caught")
        write_two(file_name)
```



Other decorators: linking with other programming models

eFlows4HPC

- A task can be more than a sequential function
  - A task in PyCOMPSs can be sequential, multicore or multi-node
  - External binary invocation: wrapper function generated automatically
  - Supports for alternative programming models: MPI and OmpSs
- Additional decorators:
  - @binary(binary=“app.bin”)
  - @mpi(binary=“mpiApp.bin”, runner=“mpirun”, processes=8)
  - @ompss(binary=“ompssApp.bin”)
- Can be combined with the @constraint and @implement decorators

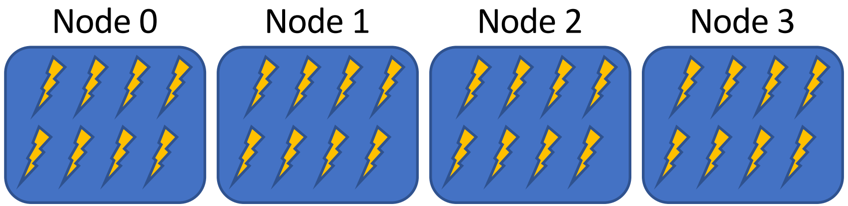
```
@binary(binary="app.bin", workingDir="/myApp")
@task()
def func(l):
    pass
```

Support for MPI tasks

- Resource manager aware of multi-node tasks

```
@mpi (runner="mpirun", processes= ”32”, processes_per_node=8)
@task (returns=int, stdOutFile=FILE_OUT_STDOUT, stdErrFile=FILE_OUT_STDERR)
def nems(stdOutFile, stdErrFile):
    pass
```

Launches MPI execution with
32 processes
8 processes per node



MPMD applications

- The @mpmd_mpi decorator can be used to define Multiple Program Multiple Data (MPMD) MPI tasks

```
@mpmd_mpi(runner="mpirun", working_dir = {{working_dir_exe}},
          programs=[{binary="fesom.x", processes = "$FESOM_PROCS" },
                    {binary="oifs", args="-v ecmwf -e awi3", processes = "$OIFS_PROCS" },
                    {binary="rnfma", processes = "$RNFMA_PROCS"}])
@task(log_file={Type:FILE_OUT, StdIOStream:STDOUT}, working_dir_exe=DIRECTORY_INOUT)
def esm_simulation(log_file, working_dir_exe):
   pass
```

- As a result of the @mpmd_mpi annotation, the following commands will be generated:

```
> cd working_dir_exe; mpirun -n $FESOM_PROCS fesom.x : \
     -n $OIFS_PROCS oifs -v ecmwf -e awi3 : -n $RNFMA_PROCS rnfma
```

Tasks in container images

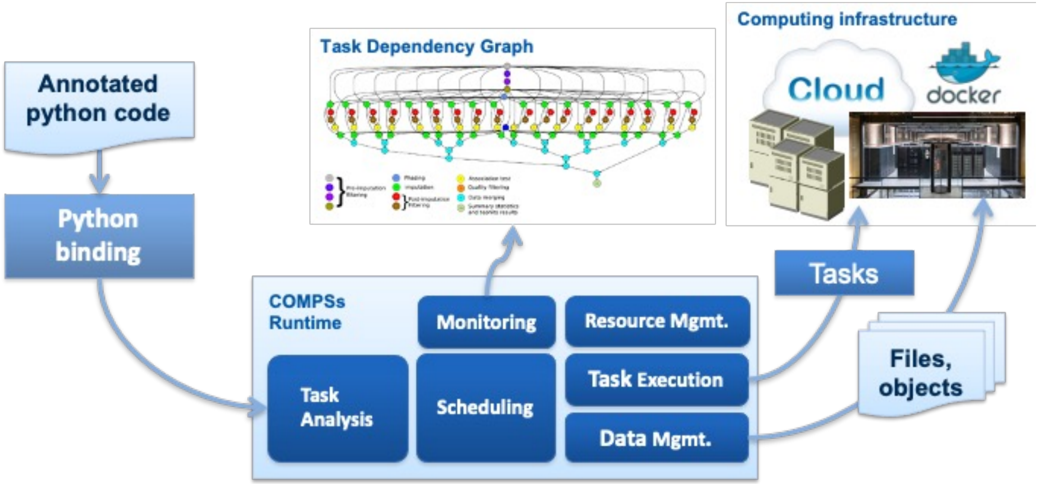
- Goal: enable tasks embedded in container images
- New @container decorator to be used together with the task annotation
- Also supports user-defined tasks

```
@container(engine="DOCKER", image="ubuntu")
@binary(binary="ls")
@task()
def task_binary_empty():
    pass
```

```
@container(engine="DOCKER", image="compss/compss")
@task(returns=1, num=IN, in_str=IN, fin=FILE_IN)
def task_python_return_str(num, in_str, fin):
    print("Hello from Task Python RETURN")
    print("- Arg 1: num -- " + str(num))
    print("- Arg 1: str -- " + str(in_str))
    print("- Arg 1: fin -- " + str(fin))
    return "Hello"
```

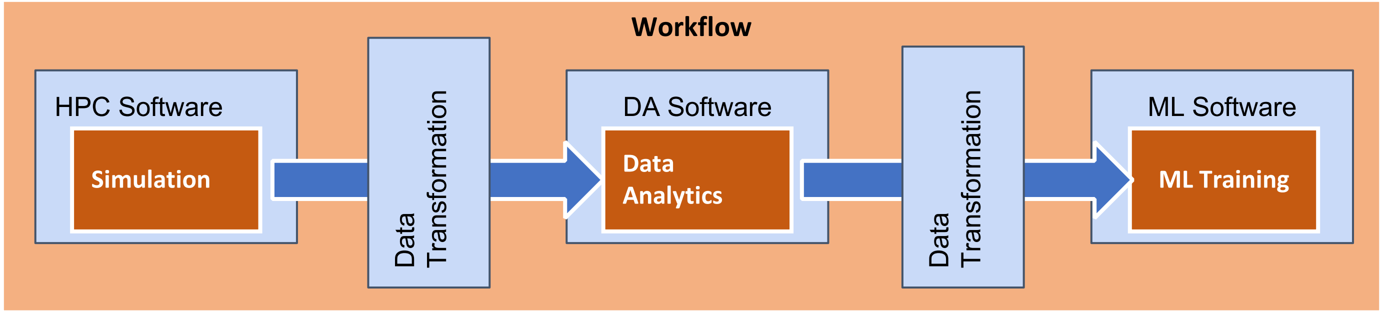
PyCOMPSs runtime

- Support for tasks' constraints – support for heterogeneous infrastructure
- Support for tasks' faults and tasks' exceptions
    - Enlarges the dynamicity of the type of workflows that we support
- Streamed data
    - ... and many others
- Runtime deployed as a distributed master-worker
- All data scheduling decisions and data transfers are performed by the runtime
- Support for elasticity



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Interfaces to integrate HPC/DA/ML



- **Goal:**
  - Reduce the required glue code to invoke multiple complex software steps
  - Developer can focus in the functionality, not in the integration
  - Enables reusability
- **Two paradigms:**
  - Software invocation
  - Data transformations

```
#workflow steps defined as tasks
@data_transformation (input_data, transformation description)
@software (invocation description)
def data_analytics (input_data, result):
    pass

#workflow body
simulation (input_cfg, sim_out)
data_analytics (sim_out, analysis_result)
ml_training (analysis_result, ml_model)
```

Software Invocation description

eFlows4HPC

Admin/user code

```
{
  "type":"mpi",
  "properties":{
    "runner": "mpirun",
    “processes”: “$SW_PROCS”
    "binary": "mpi_sofware.x",
    "params": "-d {{param}}",
    “working_dir”: “{{working_dir}}”},
  "prolog":{
     "binary":"mkdir",
     "params":"{{working_dir}}"},
   "epilog":{
     "binary":"tar",
     "params":"zcvf {{out_tgz}}" {{working_dir}}},
  "constraints":{
    "computing_units": $SW_THREADS}
}
```

Software invocation description
Stored in software catalog

Workflow Code

PyCOMPSs Code

User code

```
#workflow steps defined as tasks
@software(config_file="invocation.json")
def mpi_exec(work_dir, param, out_tgz):
    pass


#workflow body
...
mpi_exec('my_folder', 'hello_world’)
...
```

Computing Infrastructure

COMPSs runtime → Software Invocation

Automatically generated

```
mkdir working_dir
cd working_dir
export OMP_NUM_THREADS=$SW_THREADS
mpirun -n $SW_PROCS mpi_software.x -d param
tar zcvf out_tgz working_dir
```

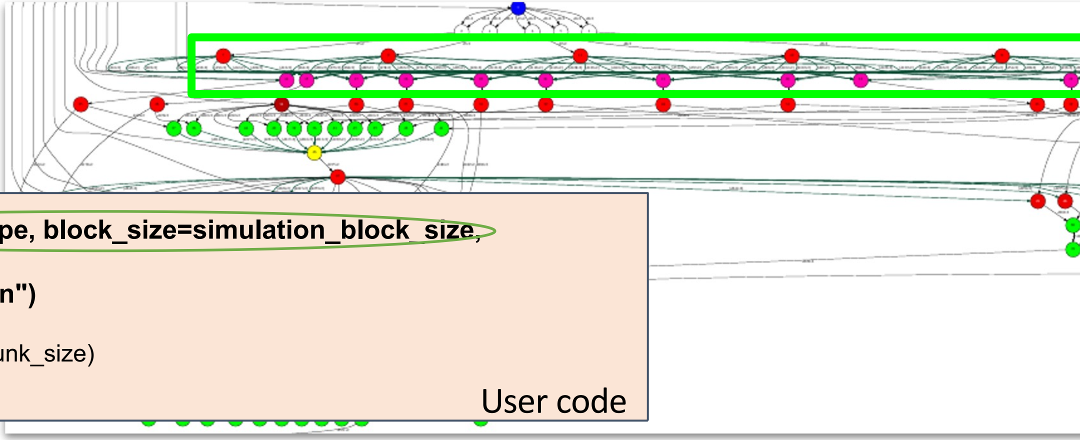
- Converts a Python function of a software invocation to a PyCOMPSs task
- Takes information from the description in json
- Enables reuse in multiple workflows

Data transformations

- **A data transformation changes the data without requiring extra programming from the developer**

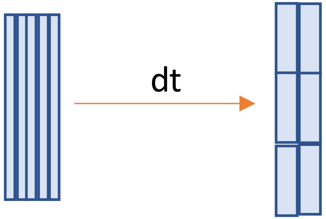
Admin/user code

```
def load_blocks_rechunk(blocks, shape, block_size, new_block_size):
    ...
    SnapshotMatrix = load_blocks_array (final_blocks, shape, block_size);
    return SnapshotMatrix
```



```
@dt("blocks", load_blocks_rechunk, shape=expected_shape, block_size=simulation_block_size,
  new_block_size=desired_block_size, is_workflow=True)
@software(config_file = SW_CATALOG + "/dislib/dislib.json")
def rSVD(blocks, desired_rank=30):
    u,s = rsvd(blocks, desired_rank, A_row_chunk_size, A_column_chunk_size)
    return u
```

User code

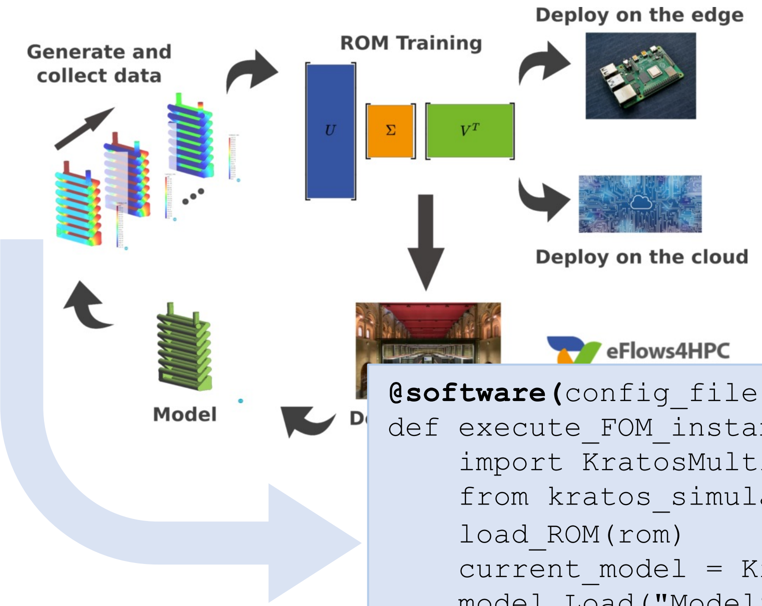



```
...
model, parameters = load_model_parameters(model_file)
for cfg in sim_cfgs:
        sim_results.append(execute_FOM_instance(model,parameters,[cfg]))
rom = rSVD(sim_results, desired_rank)
...
```

User code

Main workflow program

Pillar I: Integration of HPC and data ana... workflow



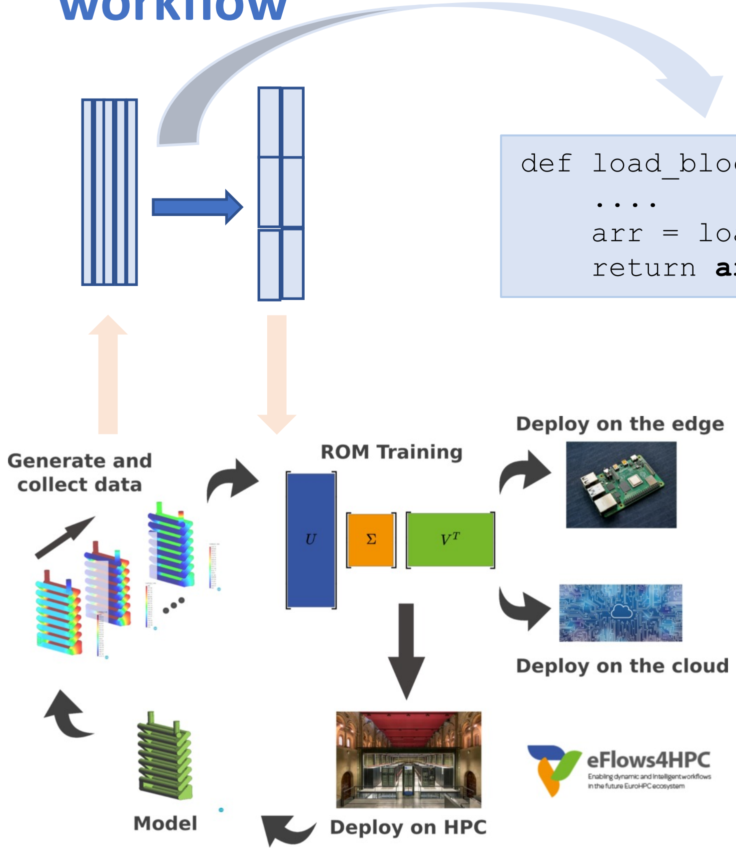
fom.json

```
{
    "execution" : {
        "type":"task"
    },
    "constraints" : {
        "computing_units": "$KRATOS_CUS"
    },
    "parameters" : {
        "returns" : 1,
        "model" : "IN",
        "parameters" : "IN",
        "sample" : "IN"
    }
}
```

```
@software(config_file = SW_CATALOG+"/kratos/fom.json")
def execute_FOM_instance(model,parameters, sample):
    import KratosMultiphysics
    from kratos_simulations import RunROM_SavingData
    load_ROM(rom)
    current_model = KratosMultiphysics.Model()
    model.Load("ModelSerialization",current_model)
    del(model)
    current_parameters = KratosMultiphysics.Parameters()
    parameters.Load("ParametersSerialization",current_parameters)
    del(parameters)
    simulation = RunROM_SavingData(current_model,current_parameters,sample)
    simulation.Run()
    return simulation.GetSnapshotsMatrix()
```

Pillar I: Integration of HPC and data analytics in a single workflow

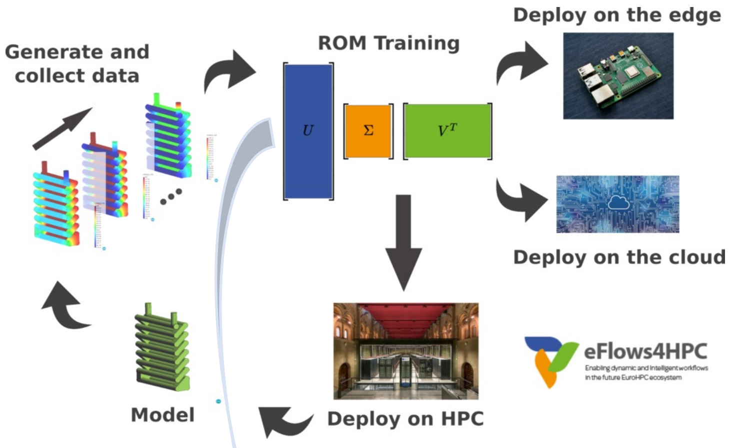
fom.json




```
def load_blocks_rechunk(blocks, shape, block_size, new_block_size):
    ....
    arr = load_blocks_array(final_blocks, shape, block_size)
    return arr.rechunk(new_block_size)
```

Dislib method that re-partitions a set of blocks into a new ds-array of the given block size

Pillar I: Integration of HPC and data analytics in a single workflow



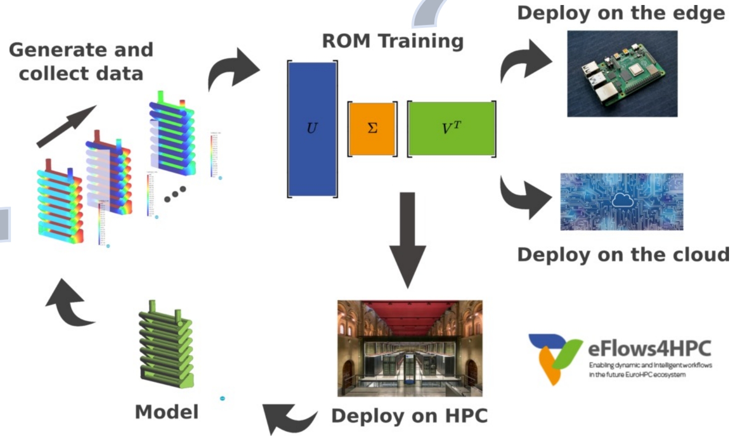
dislib.json

```
{
        "execution": {
                "type":"workflow"
        }
}
```

Method that invokes other PyCOMPSs tasks inside

```
@dt("blocks", load_blocks_rechunk, shape=expected_shape,
block_size=simulation_block_size, new_block_size=desired_block_size,
is_workflow=True)
@software(config_file = SW_CATALOG + "/py-dislib/dislib.json")
def rSVD(blocks, desired_rank=30):
    from dislib_randomized_svd import rsvd
    u,s = rsvd(blocks, desired_rank, A_row_chunk_size, A_column_chunk_size)
    return
```

Pillar I: Integration of HPC and data analytics in a single workflow




```
import dislib as ds

@dt("blocks", load_blocks_rechunk, shape=expected_shape,
block_size=simulation_block_size,new_block_size=desired_
is_workflow=True)
@software(config_file = SW_CATALOG + "/dislib/dislib.jso
def rsvd(blocks, desired_rank):
    k = desired_rank
    ...
    Y = A @ Omega
    Q,R = my_qr(Y._blocks)
    Q=load_blocks_rechunk([Q], ...)
    ...
```

```
@software(config_file=kratos.json)
def ExecuteInstance (model, parameters, Cases, instance):
    ...
    current_paramete
    ...
    simulation = Get
    simulation.Run()
    return simulatio
```

```
....
for instance in range (0,TotalNumberOFCases):
    blocks.append(ExecuteInstance(model, parameters, pars, instance))
...
U, s = rSVD(A, desired_rank)
...
```

eFlows4HPC

Enabling dynamic and Intelligent workflows in the future EuroHPC ecosystem

www.eFlows4HPC.eu

@eFlows4HPC

eFlows4HPC Project

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955558. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Germany, France, Italy, Poland, Switzerland, Norway.