



PerMedCoE Base Package

PerMedCoE Project

Dec 27, 2023

CONTENTS:

1	 Base Package Installation	3
1.1	Introduction	3
1.2	Requirements	3
1.3	Installation from Pypi	3
1.4	Installation from source code	3
1.5	Uninstall from Pypi	4
1.6	Uninstall from source code	4
2	 Base Package Components	5
2.1	Python API	5
2.2	Command line	6
2.2.1	Execution	6
2.2.1.1	Building Block execution	7
2.2.1.2	Application execution	8
2.2.2	Template creation	9
2.2.3	Automatic Deployment	9
2.2.3.1	Building Block Deployment	9
2.2.3.2	Workflow Deployment	10
3	 Existing Building Blocks and Workflows	11
3.1	Available Building Blocks	11
3.1.1	High-throughput Mutant Analysis	11
3.1.2	Meta-analysis of PhysiBoSS Output	11
3.1.3	Personalise Patient	12
3.1.4	PhysiBoSS	12
3.1.5	Single-cell Processing	12
3.1.6	Build Model from Genes	12
3.1.7	Print Drug Results	12
3.1.8	CARNIVAL	13
3.1.9	CARNIVALPy	13
3.1.10	CARNIVAL Feature Merger	13
3.1.11	CARNIVAL Gex Preprocess	13
3.1.12	Export Solver HDF5	13
3.1.13	JAX drug prediction	13
3.1.14	OmniPath	14
3.1.15	PROGENy	14
3.1.16	TF Enrichment	14
3.1.17	CellNOpt	14
3.1.18	Invasion Analysis	14
3.1.19	PhysiBoSS Invasion	14

3.1.20	COBREXA FVA	14
3.1.21	CLL Prepare Data	15
3.1.22	CLL Tf Activities	15
3.1.23	CLL Network Inference	15
3.1.24	CLL Personalize Boolean Models	15
3.1.25	CLL Run Boolean Model	15
3.1.26	CLL Combine Models	15
3.2	Existing Workflows	15
3.2.1	Basic examples	16
3.2.2	COVID-19 Multiscale Modelling of the Virus and Patients' Tissue	16
3.2.3	Drug Synergies Screening	16
3.2.4	Single drug prediction	17
3.2.5	Cancer invasion	17
3.2.6	Cancer diagnosis	18
3.3	Tutorial	19
3.3.1	Local	19
3.3.1.1	Requirements	19
3.3.1.2	Deployment	19
3.3.1.3	Usage	21
3.3.2	Supercomputer	36
3.3.2.1	SCs	36
4	✂ Creating Building Blocks	41
4.1	Building Block Templates	41
4.1.1	Building Block template creation	41
4.1.2	Package structure	42
4.1.3	Building Block structure	43
4.1.4	Deployment	46
4.1.4.1	Installation	46
4.1.4.2	Usage	46
4.1.4.3	Uninstall	47
4.1.5	Best practices	48
4.2	Application Templates	48
4.2.1	Application template creation	48
4.2.2	Folder structure	49
4.2.3	Execution	49
4.2.4	Best practices	50
4.3	Application Execution with Croupier	50
4.3.1	Introduction	50
4.3.2	Prerequisites	50
4.3.3	Croupier's framework	50
4.3.3.1	Croupier plugin installation	51
4.3.4	Application definition (Blueprint)	52
4.3.5	Application installation (Rol: application provider)	58
4.3.6	Application instance deployment (Rol: application consumer)	60
4.3.7	Application instance execution	63
4.4	Tutorial	64
4.4.1	Requirements	64
4.4.2	Step-by-step	64
5	Acknowledgements	79
5.1	Partners	79
5.2	Funding	81



This is the HPC/Exascale Centre of Excellence for Personalised Medicine in Europe (PerMedCoE) artifacts documentation.

The target audiences for this documentation are:

- **Developers** (Python) aiming at contributing to the Personalized Medicine project environment with Building Blocks or Workflows.
- **Researchers** aiming at conducting investigations using the available Building Blocks and Workflows.

This documentation is aimed at showing:

- How to use the `permedcoe` package:
 - Python API
 - Command Line Interface
- How to develop new Building Blocks
- How to execute Building Blocks individually
- How to develop Workflows using Building Blocks
- How to execute Workflows using:
 - PyCOMPSs
 - Croupier
- A step-by-step guide from creating a building block to execute a workflow
- A step-by-step tutorial for deploying and executing the existing Building Blocks and Workflows
- A sample application

The current status of the project is **WORK IN PROGRESS**.

BASE PACKAGE INSTALLATION

1.1 Introduction

PerMedCoE project developments are based on containers enabling reproducible operations in heterogeneous HPC infrastructures, and their inclusion into **building blocks (BBs)** and **workflows**.

The `permedcoe` package provides a command line tool that eases BB execution. It also provides a Python API necessary for the development of BBs and enables the creation of empty BB templates.

This section shows how to install the PerMedCoE base package. A list of package dependencies is also provided.

1.2 Requirements

- Python ≥ 3.6
- Singularity (Apptainer)

1.3 Installation from Pypi

The package is publicly available in Pypi:

```
python3 -m pip install permedcoe
```

1.4 Installation from source code

An automatic installation script is available in the PerMedCoE base package GitHub repository:

```
git clone https://github.com/PerMedCoE/permedcoe.git
cd permedcoe
./install.sh
```

Attention: This script creates a file named `installation_files.txt` to keep track of the installed files. It is used with the `uninstall.sh` script to clean up the system.

1.5 Uninstall from Pypi

The base package can be uninstalled can be done using `pip uninstall`:

```
python3 -m pip uninstall permedcoe
```

1.6 Uninstall from source code

If installed using `install.sh`, the base package can be uninstalled by running:

```
./uninstall.sh
```

The folder can then be cleaned using the `clean.sh` script.

```
./clean.sh
```


BASE PACKAGE COMPONENTS

This section provides an overview of `permedcoe` package components and functionalities. Further to a Python API for Building Block development, the package provides a command line tool that eases the execution of building blocks or applications, and creating templates for them.

2.1 Python API

The `permedcoe` package provides a set of public decorators, parameter type definitions and functions to be used for Building Block implementation.

- Public decorators:

```
from permedcoe import container
from permedcoe import constraint
from permedcoe import binary
from permedcoe import mpi
from permedcoe import task
from permedcoe import julia
```

- Parameter type definitions:

```
from permedcoe import FILE_IN
from permedcoe import FILE_OUT
from permedcoe import FILE_INOUT
from permedcoe import DIRECTORY_IN
from permedcoe import DIRECTORY_OUT
from permedcoe import DIRECTORY_INOUT
from permedcoe import Type
from permedcoe import StdIOStream
from permedcoe import STDIN
from permedcoe import STDOUT
from permedcoe import STDERR
```

- Functions:

```
from permedcoe import set_debug
from permedcoe import get_environment
from permedcoe import invoker
from permedcoe.utils.user_arguments import Arguments
```

- Globals:

```
from permedcoe import TMPDIR
```

The usage of these decorators, type definitions and functions is described in *Building Block structure*.

2.2 Command line

The `permedcoe` command can be used to execute individual Building Blocks or applications. It can also be used to create empty building block or application templates.

```
$ permedcoe -h
usage: permedcoe [-h] [-d] [-l {debug,info,warning,error,critical}] {execute,x,template,
→t,deploy,d} ...

positional arguments:
  {execute,x,template,t,deploy,d}
    execute (x)          Execute a building block.
    template (t)         Shows an example of the requested template.
    deploy (d)           Download and deploy the requested workflow or building block.

options:
  -h, --help            show this help message and exit
  -d, --debug            Enable debug mode. Overrides log_level (default: False)
  -l {debug,info,warning,error,critical}, --log_level {debug,info,warning,error,critical}
                        Set logging level. (default: error)
```

2.2.1 Execution

The execution of building blocks or applications with the `permedcoe` command is performed by indicating `execute` (or `x`) after `permedcoe`. Adding the `-h` flag after *permedcoe execute* can be used to access help:

```
$ permedcoe execute -h
usage: permedcoe execute [-h] {building_block,bb,application,app} ...

positional arguments:
  {building_block,bb,application,app}
    building_block (bb)
                        Execute a building block.
    application (app)  Execute an application.

optional arguments:
  -h, --help            show this help message and exit
```

2.2.1.1 Building Block execution

To execute an available building block, follow `permedcoe execute` by `building_block` (or `bb`).

Warning: The building block to be executed must be installed, and its name (as imported in Python) has to be provided.

```
$ permedcoe execute building_block -h
usage: permedcoe execute building_block [-h] name ...

positional arguments:
  name          Building Block to execute
  parameters    Building Block parameters

options:
  -h, --help    show this help message and exit
```

Tip: Specifying the name of the building block provides the parameters details. This example shows the parameters of the PhysiBoSS Building Block:

```
$ permedcoe execute building_block PhysiBoSS_BB -h
usage: permedcoe [-h] --sample SAMPLE --repetition REPETITION --prefix PREFIX --bnd_file_
↳BND_FILE
                        --cfg_file CFG_FILE --parallel PARALLEL --max_time MAX_TIME --out_file_
↳OUT_FILE
                        --err_file ERR_FILE --results_dir RESULTS_DIR [-c CONFIG] [-d]
                        [-l {debug,info,warning,error,critical}] [--tmpdir TMPDIR] [--processes_
↳PROCESSES]
                        [--gpus GPUS] [--memory MEMORY] [--mount_points MOUNT_POINTS]

This building block is used to perform a multiscale simulation of a population of cells_
↳using
PhysiBoSS. The tool uses the different Boolean models personalised by the Personalise_
↳patient building
block and with the mutants selected by the High-throughput mutant analysis building_
↳block. More
information on this tool can be found in [Ponce-de-Leon et al.
(2022)](https://www.biorxiv.org/content/10.1101/2022.01.06.468363v1) and the [PhysiBoSS_
↳GitHub
repository](https://github.com/PhysiBoSS/PhysiBoSS).

options:
  -h, --help                show this help message and exit
  --sample SAMPLE           (INPUT - str) Patient's identifier
  --repetition REPETITION  (INPUT - int) Number of repetition to be performed
  --prefix PREFIX          (INPUT - str) Name of the model
  --bnd_file BND_FILE      (INPUT - str (file)) Name of the model's BND file
  --cfg_file CFG_FILE      (INPUT - str (file)) Name of the model's CFG file
  --parallel PARALLEL      (INPUT - int) Internal parallelism
  --max_time MAX_TIME      (INPUT - int) PhysiBoSS simulation maximum time
```

(continues on next page)

(continued from previous page)

```

--out_file OUT_FILE    (OUTPUT - str) Main output of the PhysiBoSS run
--err_file ERR_FILE    (OUTPUT - str) Error output of the PhysiBoSS run
--results_dir RESULTS_DIR
                        (OUTPUT - str) Results directory
-c CONFIG, --config CONFIG
                        (CONFIG) Configuration file path
-d, --debug            Enable Building Block debug mode. Overrides log_level
-l {debug,info,warning,error,critical}, --log_level {debug,info,warning,error,critical}
                        Set logging level
--tmpdir TMPDIR        Temp directory to be mounted in the container
--processes PROCESSES  Number of processes for MPI executions
--gpus GPUS            Requirements for GPU jobs
--memory MEMORY        Memory requirement
--mount_points MOUNT_POINTS
                        Comma separated alias:folder to be mounted in the container

```

2.2.1.2 Application execution

Alternatively, `permedcoe execute` can be followed by `application` (or `app`) to execute an application.

Warning: The workflow manager selected must be available in the system.

```

$ permedcoe execute application -h
usage: permedcoe execute application [-h] [-w {none,pycompss,nextflow,snakemake}]
                                     [-f FLAGS [FLAGS ...]]
                                     name [parameters [parameters ...]]

positional arguments:
  name                Application to execute
  parameters          Application parameters (default: None)

optional arguments:
  -h, --help          show this help message and exit
  -w {none,pycompss,nextflow,snakemake}, --workflow_manager {none,pycompss,nextflow,
↪snakemake}
                     Workflow manager to use (default: none)
  -f FLAGS [FLAGS ...], --flags FLAGS [FLAGS ...]
                     Workflow manager flags (default: None)

```

2.2.2 Template creation

The `permedcoe` command can also be used to create an empty building block or application template:

```
$ permedcoe template -h
usage: permedcoe template [-h] [-t {all,pycompss,nextflow,snakemake}]
                           {bb,building_block,app,application} name

positional arguments:
  {bb,building_block,app,application}
                           Creates a Building Block or Application template.
  name
                           Building Block or Application name.

optional arguments:
  -h, --help                show this help message and exit
  -t {all,pycompss,nextflow,snakemake}, --type {all,pycompss,nextflow,snakemake}
                           Application type. (default: all)
```

Hint: Once the artifact is created, it describes the minimal expected implementation actions to be done in order to complete a Building Block or an application.

2.2.3 Automatic Deployment

The `permedcoe` command can also be used to deploy automatically an existing Building Block (from the [PerMedCoE GitHub repository](#)) or an existing Workflow (also from the [PerMedCoE GitHub repository](#)):

```
$ permedcoe deploy -h
usage: permedcoe deploy [-h] {building_block,bb,workflow,wf} ...

positional arguments:
  {building_block,bb,workflow,wf}
    building_block (bb)
                           A specific building block.
    workflow (wf)          A specific workflow.

options:
  -h, --help                show this help message and exit
```

2.2.3.1 Building Block Deployment

The `permedcoe deploy` command can be specified with the `building_block` (or `bb`) in order to request the automatic deployment of an existing Building Block (from the [PerMedCoE GitHub repository](#))

This feature will download automatically the requested Building Block (name), it will install in your machine, and download as well the required container image.

```
$ permedcoe deploy building_block -h
usage: permedcoe deploy building_block [-h] name

positional arguments:
```

(continues on next page)

(continued from previous page)

```
name          Building Block to deploy.

options:
  -h, --help  show this help message and exit
```

Important: The PERMEDCOE_IMAGES environment variable must be set in order to use this feature with the path where to store the container image.

After the Building Block deployment, the Building Block will be ready to be used from the command line or from a Python application.

2.2.3.2 Workflow Deployment

The `permedcoe deploy` command can be specified with the `workflow` (or `wf`) in order to request the automatic deployment of an existing Workflow (from the [PerMedCoE GitHub repository](#))

This feature will download automatically the requested workflow(name). Thus, it will also download all necessary Building Blocks, install them in your machine, and download all necessary container images.

```
$ permedcoe deploy workflow -h
usage: permedcoe deploy workflow [-h] name

positional arguments:
  name          Workflow to deploy.

options:
  -h, --help  show this help message and exit
```

Important: The PERMEDCOE_IMAGES environment variable must be set in order to use this feature with the path where to store the container images.

After the Workflow deployment, a folder containing the workflow source will appear in the current folder. It will contain the source code as well as some helper script to run the Workflow (even for different workflow managers).

EXISTING BUILDING BLOCKS AND WORKFLOWS

This section provides a list of available PerMedCoE building blocks (BB), existing workflows and links to further user documentation.

Building blocks are executed using the **permedcoe** base package. In PerMedCoE, a BB is considered a unit of work, like a black box that performs a particular computation. Consequently, it has inputs and outputs, and can be instantiated into applications and piped as required. Combinations of several building blocks can be used to execute different workflows.

The **permedcoe** package enables BB execution using a variety of workflow managers, such as PyCOMPSs, Snakemake or Nextflow.

3.1 Available Building Blocks

This section provides general descriptions of existing PerMedCoE Building Blocks, as well as links to building block GitHub repositories. For detailed documentation, see the individual building block folders in the [PerMedCoE Building Blocks repository](#). The repository also contains [Singularity \(Apptainer\) definition files](#).

3.1.1 High-throughput Mutant Analysis

This building block uses MaBoSS to screen all the possible knockouts of a given Boolean model. It produces a candidate gene list formatted as a text file (single gene per row). More information on MaBoSS can be found in [Stoll G. et al. \(2017\)](#) and in the [MaBoSS GitHub repository](#).

[Building block GitHub repository](#)

3.1.2 Meta-analysis of PhysiBoSS Output

This building block is designed to analyse the set of simulations generated for each individual in order to assess whether any of the parameters of interest show a differential distribution between the phenotypic subgroups included in a sample. To do so, the building block processes the output generated by the patient-personalized PhysiBoSS runs, including for each individual a given number of replicates of both wild-type and mutant experiments.

With this information, the building block can be used to assess the degree of uncertainty for each parameter and determine whether there are significant differences between the different subgroups, described both graphically and numerically.

[Building block GitHub repository](#)

3.1.3 Personalise Patient

This building block tailors a given MaBoSS Boolean model to a given RNAseq dataset of interest. This RNAseq dataset can come from the *Single cell processing* building block and needs to be normalised as described in [Béal et al. \(2019\)](#) and in the [PROFILE GitHub repository](#). The *Single-cell Processing* building block performs this normalisation step.

Another option of this building block is to personalise a given MaBoSS model using cell line information such as mutations, copy number alterations and expression counts.

[Building block GitHub repository](#)

3.1.4 PhysiBoSS

This building block is used to perform a multiscale simulation of a population of cells using PhysiBoSS. The tool uses the different Boolean models personalised by the *Personalise patient* building block and with the mutants selected by the *High-throughput mutant analysis* building block. More information on this tool can be found in [Ponce-de-Leon et al. \(2022\)](#) and the [PhysiBoSS GitHub repository](#).

[Building block GitHub repository](#)

3.1.5 Single-cell Processing

This building block enables the processing and analysis of single-cell RNA-Seq data from each patient in a sample. The first step of the protocol includes quality control, filtering and normalisation of the count matrices at the cellular level. Next, the number of variable genes in each individual is determined and the corresponding scaled matrices are obtained, allowing in the next step the application of dimensionality reduction techniques such as PCA, T-SNE and UMAP.

Finally, cells are clustered using graph-based techniques and annotated to their corresponding cell type, enabling subsequent building blocks to select and work with the set of cells that are relevant to the disease under study (e.g. epithelial cells in COVID19 disease).

[Building block GitHub repository](#)

3.1.6 Build Model from Genes

This building block performs automatic network construction using OmniPath and pypath. This enables the generation of a Boolean model in MaBoSS format. It takes a list of genes of interest and returns either a simple network as a SIF file or a MaBoSS network as a BND and CFG files.

[Building block GitHub repository](#)

3.1.7 Print Drug Results

This building block generates reports from raw results of multiple drug screenings (for example multiple cell lines). It takes as input a folder containing multiple results for each individual screening (cell line). It returns multiple reports in a result folder, including figures and summary values.

[Building block GitHub repository](#)

3.1.8 CARNIVAL

The CARNIVAL building block contains the refactored CARNIVAL C++ with the new Ant Colony Optimisation (ACO) in C++ with support for OpenMP and MPI. The hdf5 file required as an input can be generated with the *Export Solver HDF5* building block. For a general overview of what CARNIVAL does, see the [CARNIVAL website](#). This building block uses the new developed ACO algorithm to find solutions without the need of using ILP solvers.

[Building block GitHub repository](#)

3.1.9 CARNIVALPy

The *CARNIVALPy* building block is a refactored vanilla CARNIVAL R version for Python with support for many commercial and non-commercial open source MILP solvers. This extends the capabilities of the old CARNIVAL software to support also open-source solvers such as GLPK or CBC, and better integration with the building block ecosystem.

[Building block GitHub repository](#)

3.1.10 CARNIVAL Feature Merger

This building block is used in Use Case 2 (*Drug Synergies Screening*) to merge the features extracted by CARNIVAL for each cell line with the original cell line features, in order to produce a final csv file that can be used for prediction with the *JAX Drug Prediction* building block.

[Building block GitHub repository](#)

3.1.11 CARNIVAL Gex Preprocess

This building block processes (reshapes and scales) gene expression data from the [Genomics of Drug Sensitivity in Cancer \(GDSC\)](#) database for use by other building blocks.

[Building block GitHub repository](#)

3.1.12 Export Solver HDF5

Exports input data required by the [vanilla version of CARNIVAL](#) (sif file, measurements and perturbations) into a HDF5 file required by the optimised version of CARNIVAL with the parallel ACO C++ solver.

[Building block GitHub repository](#)

3.1.13 JAX drug prediction

The *JAX Drug Prediction* building block implements a matrix factorisation approach to predict IC50 response values of cells with different drugs, with or without side features using JAX. This is a wrapper for [a script hosted on the Saez Laboratory GitHub repository](#). This can be used to predict e.g drug responses on cell lines from partial observations of drug/cell responses.

There are two ways of using the building block: for training and for inference (prediction).

[Building block GitHub repository](#)

3.1.14 OmniPath

Downloads the latest OmniPath database to build an initial PKN network for CARNIVAL.

[Building block GitHub repository](#)

3.1.15 PROGENy

The *PROGENy* building block uses PROGENy to extract pathway activities from gene expression data. Further information on PROGENy can be found on the [Saez Laboratory website](#).

[Building block GitHub repository](#)

3.1.16 TF Enrichment

The *TF Enrichment* building block uses DecoupleR and Dorothea to estimate transcription factor activities from perturbational data.

[Building block GitHub repository](#)

3.1.17 CellNOpt

This is the refactored CellNOpt in C++ with the ACO solver with OpenMP/MPI support. A description of what CellNOpt is and how to use it is available on the [Saez Laboratory website](#).

[Building block GitHub repository](#)

3.1.18 Invasion Analysis

This building block extracts quantifications about type of invasion from a PhysiBoSS result.

[Building block GitHub repository](#)

3.1.19 PhysiBoSS Invasion

This building block is used to perform a multiscale simulation of a population of cells using PhysiBoSS. More information on this tool can be found in [Ponce-de-Leon et al. \(2022\)](#) and the [PhysiBoSS GitHub repository](#).

[Building block GitHub repository](#)

3.1.20 COBREXA FVA

This building block runs Flux Variability Analysis (FVA) on a given model. The analysis computes a feasible range of fluxes that may go through each reaction in the model while the model is in near-optimal state.

[Building block GitHub repository](#)

3.1.21 CLL Prepare Data

This building block involves an in-house script for the primary analysis of the input RNA-Seq data, focusing on tasks such as differential expression analysis and batch effect correction.

[Building block GitHub repository](#)

3.1.22 CLL Tf Activities

This building block entails the inference of transcription factor (TF) activities using DecoupleR and the quantification of molecular pathways through PROGENY.

[Building block GitHub repository](#)

3.1.23 CLL Network Inference

This building block involves network inference with CARNIVAL, leveraging Omnipath, as well as DecoupleR and PROGENY results as constraints within the linear programming problem.

[Building block GitHub repository](#)

3.1.24 CLL Personalize Boolean Models

This building block is responsible for building patient-specific boolean models by employing the PROFILE tool and input RNA-Seq data.

[Building block GitHub repository](#)

3.1.25 CLL Run Boolean Model

This building block evaluates a single patient or group-specific model using MaBoSS.

[Building block GitHub repository](#)

3.1.26 CLL Combine Models

This building block combines patient or group-specific results from MaBoSS, assessing whether the obtained profiles are appropriately clustered and can serve as predictors of disease subtype.

[Building block GitHub repository](#)

3.2 Existing Workflows

This section provides general descriptions of existing PerMedCoE Workflows, as well as links to Workflow GitHub repositories.

3.2.1 Basic examples

A basic example application comprised of a single building block can be found in [this repository](#).

A more complex application involving several building blocks that internally call `gromacs` can be found [here](#).

Tip: Note that, for these basic examples, the workflow repository contains both the Building Block and the application. In other examples, Building Blocks are maintained in a [separate repository](#).

3.2.2 COVID-19 Multiscale Modelling of the Virus and Patients' Tissue

Uses multiscale simulations to predict patient-specific SARS-CoV-2 severity subtypes (moderate, severe or control), using single-cell RNA-Seq data, MaBoSS and PhysiBoSS. Boolean models are used to determine the behaviour of individual agents as a function of extracellular conditions and the concentration of different substrates, including the number of virions. Predictions of severity subtypes are based on a meta-analysis of personalised model outputs simulating cellular apoptosis regulation in epithelial cells infected by SARS-CoV-2.

The workflow uses the following building blocks, described in order of execution:

1. High-throughput mutant analysis
2. Single-cell processing
3. Personalise patient
4. PhysiBoSS
5. Analysis of all simulations

For details on individual workflow steps, see the user documentation for each building block.

[GitHub repository](#)

3.2.3 Drug Synergies Screening

This pipeline simulates a drug screening on personalised cell line models. It automatically builds Boolean models of interest, then uses cell lines data (expression, mutations, copy number variations) to personalise them as MaBoSS models. Finally, this pipeline simulates multiple drug intervention on these MaBoSS models, and lists drug synergies of interest.

The workflow uses the following building blocks, described in order of execution:

1. Build model from species
2. Personalise patient
3. MaBoSS
4. Print drug results

For details on individual workflow steps, see the user documentation for each building block. [GitHub repository](#)

3.2.4 Single drug prediction

Complementarily, the workflow supports single drug response predictions to provide a baseline prediction in cases where drug response information for a given drug and cell line is not available. As an input, the workflow needs basal gene expression data for a cell, the drug targets (they need to be known for untested drugs) and optionally CARNIVAL features (sub-network activity predicted with CARNIVAL building block) and predicts log(IC50) values. This workflow uses a custom matrix factorization approach built with Google JAX and trained with gradient descent. The workflow can be used both for training a model, and for predicting new drug responses.

The workflow uses the following building blocks in order of execution (for training a model):

1. **Carnival_gex_preprocess**

- Preprocessed the basal gene expression data from GDSC. The input is a matrix of Gene x Sample expression data.

2. **Progeny**

- Using the preprocessed data, it estimates pathway activities for each column in the data (for each sample). It returns a matrix of Pathways x Samples with activity values for 11 pathways.

3. **Omnipath**

- It downloads latest Prior Knowledge Network of signalling. This building block can be omitted if there exists already a csv file with the network.

4. **TF Enrichment**

- For each sample, transcription factor activities are estimated using Dorothea.

5. **CarnivalPy**

- Using the TF activities estimated before, it runs Carnival to obtain a sub-network consistent with the TF activities (for each sample).

6. **Carnival_feature_merger**

- Preselect a set of genes by the user (if specified) and merge the features with the basal gene expression data.

7. **ML Jax Drug Prediction**

- Trains a model using the combined features to predict IC50 values from GDSC.

For details on individual workflow steps, please check the scripts that use each individual building block in the workflow [GitHub repository](#).

3.2.5 Cancer invasion

Multi-scale models are parametrized by many constants, many of them unknown, and which will impact their behaviour. An important part of the development of such models is to set these constants' values in order to successfully reproduce the observed biological behaviour. To perform this task, many conditions needs to be simulated to test for possible parameter values, a very heavy computational task.

This workflow perform such a task known as a parameter sensitivity analysis, which helps characterize which parameters are important for the observed behaviour, and to which values this parameter should be set.

The workflow uses the following building blocks, described in order of execution:

1. **PhysiBoSS invasion**

- Simulate the tumor invasion model (Ruscone et al., Bioinformatics, 2023) and generate outputs.

2. Invasion analysis

- Analyse the simulation outputs and generates plots with the quantification of single and collective migration, according to the parameter values.

For details on individual workflow steps, see the user documentation for each building block. [GitHub repository](#)

3.2.6 Cancer diagnosis

This use case describes a computational workflow for building a mechanistic model that captures molecular differences between two cancer subtypes, with a focus on Chronic Lymphocytic Leukaemia (CLL). The study uses RNA-Seq data and a specific clinical variable, drawing on the ICGC consortium's data, making it potentially applicable to various cancer types. The analysis aims to understand cellular signalling differences between IGHV groups by employing tools to assess transcription factor activity and provide a signalling network, offering a mechanistic explanation for observed molecular changes. The creation of patient-specific Boolean models allows for studying individual patient trajectories, emphasizing the importance of personalized medicine and tailoring approaches to account for genomic heterogeneity in cancer. Overall, this use case showcases the application of mathematical modelling tools in personalized medicine to understand and adapt approaches based on individual patient characteristics.

1. CLL prepare data

- This involves an in-house script for the primary analysis of the input RNA-Seq data, focusing on tasks such as differential expression analysis and batch effect correction.

2. CLL tf activities

- This block entails the inference of transcription factor (TF) activities using DecoupleR and the quantification of molecular pathways through PROGENY.

3. CLL network inference

- This step involves network inference with CARNIVAL, leveraging Omnipath, as well as DecoupleR and PROGENY results as constraints within the linear programming problem.

4. CLL personalise boolean models

- This block is responsible for building patient-specific boolean models by employing the PROFILE tool and input RNA-Seq data.

5. CLL run boolean model

- It involves evaluating a single patient or group-specific model using MaBoSS.

6. CLL combine results

- This block combines patient or group-specific results from MaBoSS, assessing whether the obtained profiles are appropriately clustered and can serve as predictors of disease subtype.

For details on individual workflow steps, see the user documentation for each building block. [GitHub repository](#)

3.3 Tutorial

This section provides a step-by-step by tutorial on how to use the artifacts designed and created in the PerMedCoE project scope: Building Blocks and Workflows.

It is organized in two sections focused on the target resources: [laptops/pcs](#) and [supercomputers](#).

3.3.1 Local

All PerMedCoE Building Blocks and Workflows can be deployed and executed in local machines. To this end, this section provides step-by-step detailed instructions on the requirements, how to do the deployment, and how to run the workflows.

The first step is to make sure that the target machine has the required [requirements](#).

3.3.1.1 Requirements

For local installations, the `permedcoe` package is **REQUIRED** to be installed. Since it is a Python package available in the Pypi repository, it can be easily installed using `pip`:

```
$ python3 -m pip install permedcoe

Defaulting to user installation because normal site-packages is not writeable
Collecting permedcoe
  Downloading permedcoe-0.0.11-py3-none-any.whl (40 kB)
    40.6/40.6 kB 2.7 MB/s eta 0:00:00
Requirement already satisfied: pyyaml in /home/user/.local/lib/python3.10/site-packages (6.0)
Installing collected packages: permedcoe
Successfully installed permedcoe-0.0.11
```

Tip: Alternatively, it is possible to be [installed from source](#)

It is also **REQUIRED** to install Aptainer (for automatic container download).

Please, check the [Aptainer installation documentation](#).

3.3.1.2 Deployment

The deployment can be done for specific Building Blocks, or for complete Workflows (that automatically deploys its required Building Blocks). This section describes how to deploy a single Building Block and a complete Workflow:

Deploy an existing Building Block

Existing Building Blocks can be deployed automatically with the `permedcoe` command (provided by the `permedcoe` package):

```
$ export PERMEDCOE_IMAGES=/path/where/to/store/the/containers/
$ permedcoe deploy building_block PhysiBoSS
```

[It may take a while since it downloads the required container]

(continues on next page)

(continued from previous page)

```

----- STDOUT -----
Defaulting to user installation because normal site-packages is not writeable
Collecting git+https://github.com/PerMedCoE/BuildingBlocks.git@main
  ↳ #subdirectory=PhysiBoSS
  Cloning https://github.com/PerMedCoE/BuildingBlocks.git (to revision main) to /tmp/pip-
  ↳ req-build-zdw3mlse
  Resolved https://github.com/PerMedCoE/BuildingBlocks.git to commit
  ↳ 84071d6665edb4a8ea90249ffb5b8e2f583ff13a
  Installing build dependencies: started
  Installing build dependencies: finished with status 'done'
  Getting requirements to build wheel: started
  Getting requirements to build wheel: finished with status 'done'
  Preparing metadata (pyproject.toml): started
  Preparing metadata (pyproject.toml): finished with status 'done'
Requirement already satisfied: permedcoe>=0.0.8 in /home/user/.local/lib/python3.10/site-
  ↳ packages (from meta-analysis-BB==0.0.3) (0.0.8)
Requirement already satisfied: pyyaml in /home/user/.local/lib/python3.10/site-packages
  ↳ (from permedcoe>=0.0.8->meta-analysis-BB==0.0.3) (6.0)

----- STDERR -----
Running command git clone --filter=blob:none --quiet https://github.com/PerMedCoE/
  ↳ BuildingBlocks.git /tmp/pip-req-build-zdw3mlse

```

The result of this command will install the PhysiBoSS Building Block and downloads its required container (stored in `${PERMEDCOE_IMAGES}` path).

Important: It is recommended to keep a single `${PERMEDCOE_IMAGES}` folder where to store all Building Block containers.

Tip: A full list of the available Building Blocks can be found in [Available Building Blocks Section](#).

From this point, the Building Block will be available in the machine, and it can be used in two ways: by invoking it directly through command line, or using its Python interface. This is further explained in the execution [Section](#).

Deploy an existing Workflow

Existing Workflows can be deployed automatically with the `permedcoe` command (provided by the `permedcoe` package):

```

$ export PERMEDCOE_IMAGES=/path/where/to/store/the/containers/
$ permedcoe deploy workflow covid-19-workflow
SUCCESS: Workflow deployed.

```

[It may take a while since it deploys all building blocks required by this workflow]

The result of this command will deploy the `covid-19-workflow` Workflow within the current directory. It further installs the required Building Blocks as well as downloads their required containers (storing them in `${PERMEDCOE_IMAGES}` path).

Important: It is recommended to keep a single `${PERMEDCOE_IMAGES}` folder where to store all Building Block containers.

Tip: A full list of the available workflows can be found in [Existing Workflows Section](#).

From this point, the Workflow (as well as all its required Building Blocks) will be available in the machine. Its usage is explained in the usage [Section](#).

3.3.1.3 Usage

This section aims at showing how to use individual Building Blocks and complete Workflows. Please, remind that Building Blocks are designed for a specific purpose, while the workflows use various Building Blocks in order to perform a particular analysis.

Use a Building Block

Building Blocks can be used in two ways: from the command line or through their Python interface.

Command line

Each Building Block provides a command (with the same name as the building block followed by `_BB`) that can be launched from the command line. For example, the PhysiBoSS Building block (previously deployed) provides the `PhysiBoSS_BB` command. Its usage details can be checked using the `--help` or `-h` flag:

```
$ PhysiBoSS_BB --help

usage: PhysiBoSS_BB [-h] --sample SAMPLE --repetition REPETITION --prefix PREFIX --
↳bnd_file BND_FILE --cfg_file CFG_FILE --parallel PARALLEL --max_time MAX_TIME --
↳out_file OUT_FILE --err_file ERR_FILE
      --results_dir RESULTS_DIR [-c CONFIG] [-d] [-l {debug,info,
↳warning,error,critical}] [--tmpdir TMPDIR] [--processes PROCESSES] [--gpus GPUS]
↳[--memory MEMORY] [--mount_points MOUNT_POINTS]

This building block is used to perform a multiscale simulation of a population of
↳cells using PhysiBoSS. The tool uses the different Boolean models personalised by
↳the Personalise patient building block and
with the mutants selected by the High-throughput mutant analysis building block.
↳More information on this tool can be found in [Ponce-de-Leon et al.
(2022)] (https://www.biorxiv.org/content/10.1101/2022.01.06.468363v1) and the
↳[PhysiBoSS GitHub repository] (https://github.com/PhysiBoSS/PhysiBoSS).

options:
-h, --help                show this help message and exit
--sample SAMPLE            (INPUT - str) Patient's identifier
--repetition REPETITION   (INPUT - int) Number of repetition to be performed
--prefix PREFIX            (INPUT - str) Name of the model
--bnd_file BND_FILE        (INPUT - str (file)) Name of the model's BND file
--cfg_file CFG_FILE        (INPUT - str (file)) Name of the model's CFG file
--parallel PARALLEL        (INPUT - int) Internal parallelism
--max_time MAX_TIME        (INPUT - int) PhysiBoSS simulation maximum time
--out_file OUT_FILE        (OUTPUT - str) Main output of the PhysiBoSS run
```

(continues on next page)

(continued from previous page)

```

--err_file ERR_FILE   (OUTPUT - str) Error output of the PhysiBoSS run
--results_dir RESULTS_DIR
                        (OUTPUT - str) Results directory
-c CONFIG, --config CONFIG
                        (CONFIG) Configuration file path
-d, --debug           Enable Building Block debug mode. Overrides log_level
-l {debug,info,warning,error,critical}, --log_level {debug,info,warning,error,
→critical}
                        Set logging level
--tmpdir TMPDIR       Temp directory to be mounted in the container
--processes PROCESSES
                        Number of processes for MPI executions
--gpus GPUS           Requirements for GPU jobs
--memory MEMORY       Memory requirement
--mount_points MOUNT_POINTS
                        Comma separated alias:folder to be mounted in the container

```

Thanks to the `-h` or `--help` flag, the PhysiBoSS Building Block inputs and outputs are shown and described. All ``INPUT and OUTPUT flags are required in order to execute the Building Block. The rest of the parameters are optional and can be used to define particular options (e.g. -l or --log_level can be used to define the level of information to be printed, or the --tmpdir can be used to define a specific folder that will be mounted in the associated container – this can be useful if the input parameters are in a non default mounted folder).`

Caution: All Building Blocks require the `PERMEDCOE_IMAGES` environment variable. Otherwise, it will raise the following exception:

```

Traceback (most recent call last):
File "/home/user/.local/bin/meta_analysis_BB", line 5, in <module>
    from meta_analysis_BB.__main__ import main
File "/home/user/.local/lib/python3.10/site-packages/meta_analysis_BB/__init__.py
→", line 6, in <module>
    from meta_analysis_BB.main import *
File "/home/user/.local/lib/python3.10/site-packages/meta_analysis_BB/main.py",
→line 13, in <module>
    from meta_analysis_BB.definitions import CONTAINER
File "/home/user/.local/lib/python3.10/site-packages/meta_analysis_BB/
→definitions.py", line 2, in <module>
    from permedcoe.bb import CONTAINER_PATH
File "/home/user/.local/lib/python3.10/site-packages/permedcoe/bb.py", line 24,
→in <module>
    CONTAINER_PATH = get_container_path()
File "/home/user/.local/lib/python3.10/site-packages/permedcoe/bb.py", line 21,
→in get_container_path
    raise Exception("Please define %s environment variable with the path." %
→CONTAINER_PATH_VN)
Exception: Please define PERMEDCOE_IMAGES environment variable with the path.

```

The result of executing the Building Block will be a one or more output files that can potentially be used by other Building Block.

Python interface

Since the Building Blocks are developed in Python, they offer a Python interface that can be used from Python applications.

For example, the PhysiBoSS Building Block can be used as follows:

```
from PhysiBoSS_BB import physiboss

physiboss(
    sample="C141",
    repetition=1,
    prefix="epithelial_cell_2_personalized",
    bnd_file="/path/to/input_file.bnd",
    cfg_file="/path/to/input_file.cfg",
    out_file="/path/to/output_file.txt",
    err_file="/path/to/output_file.txt",
    results_dir="/path/to/results",
    parallel=1,
    max_time=8640,
    tmpdir="/path/to/tmpdir"
)
```

Note that this interface requires the same parameters as the command line interface.

Tip: Some Building Blocks may provide more functions since they serve for multiple purposes (e.g. MaBoSS for default behaviour or for sensitivity analysis).

Specific details about the python interface of each Building Block can be consulted in the `main.py` file of each Building Block repository.

Important: The functions provided by the Building Blocks have a set of decorators on top of them that are responsible of hiding the management complexities, but also to enable the automatic parallelization using **Py-COMPSs**. Consequently, any application making use of Building Blocks gets automatically parallelized if run using **PyCOMPSs**. If the application is run using Python directly, it will be executed sequentially.

Use a Workflow

Existing workflows that have been deployed have a particular structure:

```
$ cd covid-19-workflow
$covid-19-workflow> ls
BuildingBlocks  LICENSE  README.md  Resources  Tests  Workflow
```

The contents are:

LICENSE

The workflow license file.

README.md

The workflow description file. It provides details about the workflow.

BuildingBlocks

This folder contains a set of helper scripts to install or uninstall the Workflow required Building Blocks. They are used by the automatic deployment performed by the `permedcoe` package and command.

Resources

This folder contains a set of minimal data in order to run the Workflow (i.e. testing dataset).

Tests

This folder contains a set of testing scripts. These scripts are able to run each individual Building Block for debugging purposes.

Workflow

This is the **MAIN FOLDER** since it contains the workflow files. In particular, it can contain three subfolders:

```
$covid-19-workflow> cd Workflow
$covid-19-workflow/Workflow> ls
NextFlow  PyCOMPSs  SnakeMake
```

Each subfolder is aimed at contain the workflow for that particular workflow manager. For example, the covid-19 workflow is available in for the three workflow managers.

One peculiarity of the three workflow managers is that Snakemake and NextFlow workflows use the command line interface of the Building Blocks involved, while PyCOMPSs workflow uses their Python interface.

In addition to the workflow file, each folder also includes a `launch.sh` script in order to ease the workflow execution.

As seen in the contents, the `Workflow` folder contains the main workflow files. Consequently, it is necessary to go inside that folder in order to execute the workflow.

```
$covid-19-workflow> cd Workflow
$covid-19-workflow/Workflow>
```

The next step is to decide which workflow manager is going to be used:

```
$covid-19-workflow/Workflow> ls
NextFlow  PyCOMPSs  SnakeMake
```

The covid-19 workflow is available for PyCOMPSs, NextFlow and SnakeMake.

Important: The workflow manager **MUST** be installed in the machine in order to run the workflow.

- [PyCOMPSs installation](#)
 - [NextFlow installation](#)
 - [SnakeMake installation](#)
-

Once decided the workflow manager to be used, specific details about how to run the workflow with each of them is provided in the next drop-down sections:

PyCOMPSs

If the chosen workflow manager is PyCOMPSs, the next step is to go inside the folder:

```
$covid-19-workflow/Workflow> cd PyCOMPSs
$covid-19-workflow/Workflow/PyCOMPSs> ls
0_prepare_dataset.sh
a_launch.sh
a_run.sh
b_1_launch.sh
b_1_run.sh
```

(continues on next page)

(continued from previous page)

```

b_2_launch_per_patient.sh
b_2_launch.sh
b_2_run_per_patient.sh
b_2_run.sh
b_3_launch.sh
b_3_run.sh
clean.sh
README.md
src
src_split

```

0_prepare_dataset.sh

This script downloads and configures the testing dataset. It just requires to be executed once and without parameters (`./0_prepare_dataset.sh`)

a_launch.sh

Script that launches the workflow with the testing dataset within a supercomputer.

a_run.sh

Script that runs the workflow with the testing dataset.

src

Folder that contains the workflow written in Python and using PyCOMPSs.

clean.sh

Helper script that cleans the current folder after running the workflow. Use with caution since it removes all result files.

src_split and b_\`*` scripts

The `src_split` folder contains the workflow split in three parts, so that it can be executed partially or even in different machines. Accordingly, the `b_*` scripts are aimed at launching or running each part.

The way to run the workflow (automatically parallelized with PyCOMPSs) is:

```

$ covid-19-workflow/Workflow/PyCOMPSs> ./a_run.sh

WARNING: PERMEDCOE_IMAGES environment variable not set. Using default: /home/user/github/
↳ projects/PerMedCoE/BuildingBlocks/Resources/images/
[ INFO ] Inferred PYTHON language
[ INFO ] Using default location for project file: /opt/COMPSs//Runtime/configuration/xml/
↳ projects/default_project.xml
[ INFO ] Using default location for resources file: /opt/COMPSs//Runtime/configuration/
↳ xml/resources/default_resources.xml
[ INFO ] Using default execution type: compss

----- Executing covid19_pilot.py -----

WARNING: COMPSs Properties file is null. Setting default values
[(834)  API] - Starting COMPSs Runtime v3.2.rc2310 (build 20231017-1637.
↳ r77b4be4b8ac4f722dd3de105161229b849a545d4)

-----
| Covid-19 Pilot Workflow |
-----

>>> WELCOME TO THE PILOT WORKFLOW

```

(continues on next page)

(continued from previous page)

```

> Parameters:
  - metadata file: /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/
↳ PyCOMPSSs/../../Resources/data/metadata_small.tsv
  - model prefix: /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/
↳ PyCOMPSSs/../../Resources/data/epithelial_cell_2
  - output folder: /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/
↳ PyCOMPSSs/results/
  - ko file: /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSSs/
↳ ko_file.txt
  - replicates: 2
  - model: epithelial_cell_2
  - data folder: /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/
↳ PyCOMPSSs/../../Resources/data
  - max time: 100

KO file not detected, running MABOSS
> SINGLE CELL PROCESSING C141
> PERSONALIZING PATIENT C141
>> prefix: epithelial_cell_2_personalized
>>> Repetition: 1
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSSs/results/
↳ C141/physiboss_results/output_C141_epithelial_cell_2_personalized_1.out
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSSs/results/
↳ C141/physiboss_results/output_C141_epithelial_cell_2_personalized_1.err
>>> Repetition: 2
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSSs/results/
↳ C141/physiboss_results/output_C141_epithelial_cell_2_personalized_2.out
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSSs/results/
↳ C141/physiboss_results/output_C141_epithelial_cell_2_personalized_2.err
>> prefix: epithelial_cell_2_personalized__M_ko
>>> Repetition: 1
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSSs/results/
↳ C141/physiboss_results/output_C141_epithelial_cell_2_personalized__M_ko_1.out
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSSs/results/
↳ C141/physiboss_results/output_C141_epithelial_cell_2_personalized__M_ko_1.err
>>> Repetition: 2
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSSs/results/
↳ C141/physiboss_results/output_C141_epithelial_cell_2_personalized__M_ko_2.out
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSSs/results/
↳ C141/physiboss_results/output_C141_epithelial_cell_2_personalized__M_ko_2.err
>> prefix: epithelial_cell_2_personalized__CASP9_cell_active_ko
>>> Repetition: 1
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSSs/results/
↳ C141/physiboss_results/output_C141_epithelial_cell_2_personalized__CASP9_cell_active_
↳ ko_1.out
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSSs/results/
↳ C141/physiboss_results/output_C141_epithelial_cell_2_personalized__CASP9_cell_active_
↳ ko_1.err
>>> Repetition: 2
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSSs/results/
↳ C141/physiboss_results/output_C141_epithelial_cell_2_personalized__CASP9_cell_active_

```

(continues on next page)

(continued from previous page)

```

↪ko_2.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪C141/physiboss_results/output_C141_epithelial_cell_2_personalized__CASP9_cell_active_
↪ko_2.err
>> prefix: epithelial_cell_2_personalized__CASP8_ko
>>> Repetition: 1
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪C141/physiboss_results/output_C141_epithelial_cell_2_personalized__CASP8_ko_1.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪C141/physiboss_results/output_C141_epithelial_cell_2_personalized__CASP8_ko_1.err
>>> Repetition: 2
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪C141/physiboss_results/output_C141_epithelial_cell_2_personalized__CASP8_ko_2.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪C141/physiboss_results/output_C141_epithelial_cell_2_personalized__CASP8_ko_2.err
>> prefix: epithelial_cell_2_personalized__FASLG_ko
>>> Repetition: 1
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪C141/physiboss_results/output_C141_epithelial_cell_2_personalized__FASLG_ko_1.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪C141/physiboss_results/output_C141_epithelial_cell_2_personalized__FASLG_ko_1.err
>>> Repetition: 2
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪C141/physiboss_results/output_C141_epithelial_cell_2_personalized__FASLG_ko_2.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪C141/physiboss_results/output_C141_epithelial_cell_2_personalized__FASLG_ko_2.err
>> prefix: epithelial_cell_2_personalized__FADD_ko
>>> Repetition: 1
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪C141/physiboss_results/output_C141_epithelial_cell_2_personalized__FADD_ko_1.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪C141/physiboss_results/output_C141_epithelial_cell_2_personalized__FADD_ko_1.err
>>> Repetition: 2
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪C141/physiboss_results/output_C141_epithelial_cell_2_personalized__FADD_ko_2.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪C141/physiboss_results/output_C141_epithelial_cell_2_personalized__FADD_ko_2.err
>> prefix: epithelial_cell_2_personalized__CASP3_ko
>>> Repetition: 1
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪C141/physiboss_results/output_C141_epithelial_cell_2_personalized__CASP3_ko_1.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪C141/physiboss_results/output_C141_epithelial_cell_2_personalized__CASP3_ko_1.err
>>> Repetition: 2
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪C141/physiboss_results/output_C141_epithelial_cell_2_personalized__CASP3_ko_2.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪C141/physiboss_results/output_C141_epithelial_cell_2_personalized__CASP3_ko_2.err
>> prefix: epithelial_cell_2_personalized__FAS_FASL_complex_ko
>>> Repetition: 1
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪C141/physiboss_results/output_C141_epithelial_cell_2_personalized__FAS_FASL_complex_ko_

```

(continues on next page)

(continued from previous page)

```

↪ 1.out
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C141/physiboss_results/output_C141_epithelial_cell_2_personalized__FAS_FASL_complex_ko_
↪ 1.err
>>> Repetition: 2
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C141/physiboss_results/output_C141_epithelial_cell_2_personalized__FAS_FASL_complex_ko_
↪ 2.out
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C141/physiboss_results/output_C141_epithelial_cell_2_personalized__FAS_FASL_complex_ko_
↪ 2.err
>> prefix: epithelial_cell_2_personalized__Apoptosome_complex_ko
>>> Repetition: 1
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C141/physiboss_results/output_C141_epithelial_cell_2_personalized__Apoptosome_complex_
↪ ko_1.out
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C141/physiboss_results/output_C141_epithelial_cell_2_personalized__Apoptosome_complex_
↪ ko_1.err
>>> Repetition: 2
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C141/physiboss_results/output_C141_epithelial_cell_2_personalized__Apoptosome_complex_
↪ ko_2.out
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C141/physiboss_results/output_C141_epithelial_cell_2_personalized__Apoptosome_complex_
↪ ko_2.err
> SINGLE CELL PROCESSING C142
> PERSONALIZING PATIENT C142
>> prefix: epithelial_cell_2_personalized
>>> Repetition: 1
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized_1.out
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized_1.err
>>> Repetition: 2
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized_2.out
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized_2.err
>> prefix: epithelial_cell_2_personalized__M_ko
>>> Repetition: 1
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__M_ko_1.out
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__M_ko_1.err
>>> Repetition: 2
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__M_ko_2.out
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__M_ko_2.err
>> prefix: epithelial_cell_2_personalized__CASP9_cell_active_ko
>>> Repetition: 1

```

(continues on next page)

(continued from previous page)

```

- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__CASP9_cell_active_
↪ ko_1.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__CASP9_cell_active_
↪ ko_1.err
>>> Repetition: 2
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__CASP9_cell_active_
↪ ko_2.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__CASP9_cell_active_
↪ ko_2.err
>> prefix: epithelial_cell_2_personalized__CASP8_ko
>>> Repetition: 1
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__CASP8_ko_1.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__CASP8_ko_1.err
>>> Repetition: 2
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__CASP8_ko_2.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__CASP8_ko_2.err
>> prefix: epithelial_cell_2_personalized__FASLG_ko
>>> Repetition: 1
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__FASLG_ko_1.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__FASLG_ko_1.err
>>> Repetition: 2
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__FASLG_ko_2.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__FASLG_ko_2.err
>> prefix: epithelial_cell_2_personalized__FADD_ko
>>> Repetition: 1
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__FADD_ko_1.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__FADD_ko_1.err
>>> Repetition: 2
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__FADD_ko_2.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__FADD_ko_2.err
>> prefix: epithelial_cell_2_personalized__CASP3_ko
>>> Repetition: 1
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__CASP3_ko_1.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__CASP3_ko_1.err

```

(continues on next page)

(continued from previous page)

```

>>> Repetition: 2
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__CASP3_ko_2.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__CASP3_ko_2.err
>> prefix: epithelial_cell_2_personalized__FAS_FASL_complex_ko
>>> Repetition: 1
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__FAS_FASL_complex_ko_
↪ 1.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__FAS_FASL_complex_ko_
↪ 1.err
>>> Repetition: 2
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__FAS_FASL_complex_ko_
↪ 2.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__FAS_FASL_complex_ko_
↪ 2.err
>> prefix: epithelial_cell_2_personalized__Apoptosome_complex_ko
>>> Repetition: 1
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__Apoptosome_complex_
↪ ko_1.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__Apoptosome_complex_
↪ ko_1.err
>>> Repetition: 2
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__Apoptosome_complex_
↪ ko_2.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_results/output_C142_epithelial_cell_2_personalized__Apoptosome_complex_
↪ ko_2.err
>> prefix: epithelial_cell_2_personalized
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C141/physiboss_replicates_analysis/epithelial_cell_2_personalized.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C141/physiboss_replicates_analysis/epithelial_cell_2_personalized.err
>> prefix: epithelial_cell_2_personalized__M_ko
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C141/physiboss_replicates_analysis/epithelial_cell_2_personalized__M_ko.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C141/physiboss_replicates_analysis/epithelial_cell_2_personalized__M_ko.err
>> prefix: epithelial_cell_2_personalized__CASP9_cell_active_ko
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C141/physiboss_replicates_analysis/epithelial_cell_2_personalized__CASP9_cell_active_
↪ ko.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C141/physiboss_replicates_analysis/epithelial_cell_2_personalized__CASP9_cell_active_
↪ ko.err

```

(continues on next page)

(continued from previous page)

```

>> prefix: epithelial_cell_2_personalized__CASP8_ko
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
  ↪C141/physiboss_replicates_analysis/epithelial_cell_2_personalized__CASP8_ko.out
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
  ↪C141/physiboss_replicates_analysis/epithelial_cell_2_personalized__CASP8_ko.err
>> prefix: epithelial_cell_2_personalized__FASLG_ko
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
  ↪C141/physiboss_replicates_analysis/epithelial_cell_2_personalized__FASLG_ko.out
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
  ↪C141/physiboss_replicates_analysis/epithelial_cell_2_personalized__FASLG_ko.err
>> prefix: epithelial_cell_2_personalized__FADD_ko
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
  ↪C141/physiboss_replicates_analysis/epithelial_cell_2_personalized__FADD_ko.out
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
  ↪C141/physiboss_replicates_analysis/epithelial_cell_2_personalized__FADD_ko.err
>> prefix: epithelial_cell_2_personalized__CASP3_ko
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
  ↪C141/physiboss_replicates_analysis/epithelial_cell_2_personalized__CASP3_ko.out
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
  ↪C141/physiboss_replicates_analysis/epithelial_cell_2_personalized__CASP3_ko.err
>> prefix: epithelial_cell_2_personalized__FAS_FASL_complex_ko
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
  ↪C141/physiboss_replicates_analysis/epithelial_cell_2_personalized__FAS_FASL_complex_ko.
  ↪out
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
  ↪C141/physiboss_replicates_analysis/epithelial_cell_2_personalized__FAS_FASL_complex_ko.
  ↪err
>> prefix: epithelial_cell_2_personalized__Apoptosome_complex_ko
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
  ↪C141/physiboss_replicates_analysis/epithelial_cell_2_personalized__Apoptosome_complex_
  ↪ko.out
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
  ↪C141/physiboss_replicates_analysis/epithelial_cell_2_personalized__Apoptosome_complex_
  ↪ko.err
>> prefix: epithelial_cell_2_personalized
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
  ↪C142/physiboss_replicates_analysis/epithelial_cell_2_personalized.out
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
  ↪C142/physiboss_replicates_analysis/epithelial_cell_2_personalized.err
>> prefix: epithelial_cell_2_personalized__M_ko
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
  ↪C142/physiboss_replicates_analysis/epithelial_cell_2_personalized__M_ko.out
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
  ↪C142/physiboss_replicates_analysis/epithelial_cell_2_personalized__M_ko.err
>> prefix: epithelial_cell_2_personalized__CASP9_cell_active_ko
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
  ↪C142/physiboss_replicates_analysis/epithelial_cell_2_personalized__CASP9_cell_active_
  ↪ko.out
  - /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
  ↪C142/physiboss_replicates_analysis/epithelial_cell_2_personalized__CASP9_cell_active_
  ↪ko.err
>> prefix: epithelial_cell_2_personalized__CASP8_ko

```

(continues on next page)

(continued from previous page)

```

- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_replicates_analysis/epithelial_cell_2_personalized__CASP8_ko.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_replicates_analysis/epithelial_cell_2_personalized__CASP8_ko.err
>> prefix: epithelial_cell_2_personalized__FASLG_ko
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_replicates_analysis/epithelial_cell_2_personalized__FASLG_ko.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_replicates_analysis/epithelial_cell_2_personalized__FASLG_ko.err
>> prefix: epithelial_cell_2_personalized__FADD_ko
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_replicates_analysis/epithelial_cell_2_personalized__FADD_ko.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_replicates_analysis/epithelial_cell_2_personalized__FADD_ko.err
>> prefix: epithelial_cell_2_personalized__CASP3_ko
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_replicates_analysis/epithelial_cell_2_personalized__CASP3_ko.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_replicates_analysis/epithelial_cell_2_personalized__CASP3_ko.err
>> prefix: epithelial_cell_2_personalized__FAS_FASL_complex_ko
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_replicates_analysis/epithelial_cell_2_personalized__FAS_FASL_complex_ko.
↪ out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_replicates_analysis/epithelial_cell_2_personalized__FAS_FASL_complex_ko.
↪ err
>> prefix: epithelial_cell_2_personalized__Apoptosome_complex_ko
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_replicates_analysis/epithelial_cell_2_personalized__Apoptosome_complex_
↪ ko.out
- /home/user/github/projects/PerMedCoE/covid-19-workflow/Workflow/PyCOMPSS/results/
↪ C142/physiboss_replicates_analysis/epithelial_cell_2_personalized__Apoptosome_complex_
↪ ko.err
[(1277810)    API] - Execution Finished
-----

```

And the results will be stored within the current folder within the `results` folder.

```

$covid-19-workflow/Workflow/PyCOMPSS> cd results
$covid-19-workflow/Workflow/PyCOMPSS/results> tree

```

```

.
├── C141
│   ├── personalize_patient
│   │   └── [personalize_patient results]
│   ├── physiboss_replicates_analysis
│   │   └── [physiboss_replicates_analysis results]
│   ├── physiboss_results
│   │   └── [physiboss_results results]
│   ├── single_cell_processing
│   │   └── [single_cell_processing results]

```

(continues on next page)

(continued from previous page)

```

└─ C142
  └─ personalize_patient
    └─ [personalize-patient-results]
  └─ physiboss_replicates_analysis
    └─ [physiboss_replicates_analysis results]
  └─ physiboss_results
    └─ [physiboss_results results]
  └─ single_cell_processing
    └─ [single_cell_processing results]
└─ meta_analysis
  └─ cd8s_C141_Apoptosome_complex_ko_.png
  └─ cd8s_C141_CASP3_ko_.png
  └─ cd8s_C141_CASP8_ko_.png
  └─ cd8s_C141_CASP9_cell_active_ko_.png
  └─ cd8s_C141_FADD_ko_.png
  └─ cd8s_C141_FAS_FASL_complex_ko_.png
  └─ cd8s_C141_FASLG_ko_.png
  └─ cd8s_C141_M_ko_.png
  └─ cd8s_C141.png
  └─ cd8s_C142_Apoptosome_complex_ko_.png
  └─ cd8s_C142_CASP3_ko_.png
  └─ cd8s_C142_CASP8_ko_.png
  └─ cd8s_C142_CASP9_cell_active_ko_.png
  └─ cd8s_C142_FADD_ko_.png
  └─ cd8s_C142_FAS_FASL_complex_ko_.png
  └─ cd8s_C142_FASLG_ko_.png
  └─ cd8s_C142_M_ko_.png
  └─ cd8s_C142.png
  └─ cd8_traces_C141_Apoptosome_complex_ko_.png
  └─ cd8_traces_C141_CASP3_ko_.png
  └─ cd8_traces_C141_CASP8_ko_.png
  └─ cd8_traces_C141_CASP9_cell_active_ko_.png
  └─ cd8_traces_C141_FADD_ko_.png
  └─ cd8_traces_C141_FAS_FASL_complex_ko_.png
  └─ cd8_traces_C141_FASLG_ko_.png
  └─ cd8_traces_C141_M_ko_.png
  └─ cd8_traces_C141.png
  └─ cd8_traces_C142_Apoptosome_complex_ko_.png
  └─ cd8_traces_C142_CASP3_ko_.png
  └─ cd8_traces_C142_CASP8_ko_.png
  └─ cd8_traces_C142_CASP9_cell_active_ko_.png
  └─ cd8_traces_C142_FADD_ko_.png
  └─ cd8_traces_C142_FAS_FASL_complex_ko_.png
  └─ cd8_traces_C142_FASLG_ko_.png
  └─ cd8_traces_C142_M_ko_.png
  └─ cd8_traces_C142.png
  └─ clustermap_genes.png
  └─ clustermap_patients.png
  └─ clustermap.png
  └─ clustermap_traces.png
  └─ dendogram_genes.png
  └─ dendogram_patients.png

```

(continues on next page)

(continued from previous page)

- dendogram.png
- dendogram_traces.png
- epithelial_C141_Apoptosome_complex_ko_.png
- epithelial_C141_CASP3_ko_.png
- epithelial_C141_CASP8_ko_.png
- epithelial_C141_CASP9_cell_active_ko_.png
- epithelial_C141_FADD_ko_.png
- epithelial_C141_FAS_FASL_complex_ko_.png
- epithelial_C141_FASLG_ko_.png
- epithelial_C141_M_ko_.png
- epithelial_C141.png
- epithelial_C142_Apoptosome_complex_ko_.png
- epithelial_C142_CASP3_ko_.png
- epithelial_C142_CASP8_ko_.png
- epithelial_C142_CASP9_cell_active_ko_.png
- epithelial_C142_FADD_ko_.png
- epithelial_C142_FAS_FASL_complex_ko_.png
- epithelial_C142_FASLG_ko_.png
- epithelial_C142_M_ko_.png
- epithelial_C142.png
- epithelial_traces_C141_Apoptosome_complex_ko_.png
- epithelial_traces_C141_CASP3_ko_.png
- epithelial_traces_C141_CASP8_ko_.png
- epithelial_traces_C141_CASP9_cell_active_ko_.png
- epithelial_traces_C141_FADD_ko_.png
- epithelial_traces_C141_FAS_FASL_complex_ko_.png
- epithelial_traces_C141_FASLG_ko_.png
- epithelial_traces_C141_M_ko_.png
- epithelial_traces_C141.png
- epithelial_traces_C142_Apoptosome_complex_ko_.png
- epithelial_traces_C142_CASP3_ko_.png
- epithelial_traces_C142_CASP8_ko_.png
- epithelial_traces_C142_CASP9_cell_active_ko_.png
- epithelial_traces_C142_FADD_ko_.png
- epithelial_traces_C142_FAS_FASL_complex_ko_.png
- epithelial_traces_C142_FASLG_ko_.png
- epithelial_traces_C142_M_ko_.png
- epithelial_traces_C142.png
- macrophage_C141_Apoptosome_complex_ko_.png
- macrophage_C141_CASP3_ko_.png
- macrophage_C141_CASP8_ko_.png
- macrophage_C141_CASP9_cell_active_ko_.png
- macrophage_C141_FADD_ko_.png
- macrophage_C141_FAS_FASL_complex_ko_.png
- macrophage_C141_FASLG_ko_.png
- macrophage_C141_M_ko_.png
- macrophage_C141.png
- macrophage_C142_Apoptosome_complex_ko_.png
- macrophage_C142_CASP3_ko_.png
- macrophage_C142_CASP8_ko_.png
- macrophage_C142_CASP9_cell_active_ko_.png
- macrophage_C142_FADD_ko_.png

(continues on next page)

(continued from previous page)

```

├── macrophages_C142_FAS_FASL_complex_ko_.png
├── macrophages_C142_FASLG_ko_.png
├── macrophages_C142_M_ko_.png
└── macrophages_C142.png

```

11 directories, 100 files

NextFlow

If the chosen workflow manager is NextFlow, the next step is to go inside the folder:

```

$covid-19-workflow/Workflow> cd NextFlow
$covid-19-workflow/Workflow/NextFlow> ls
covid19_pilot.nf  launch.sh

```

covid-19_pilot.nf

This is the workflow script.

launch.sh

Script that launches the workflow with the testing dataset.

The way to run the workflow is:

```

$covid-19-workflow/Workflow/NextFlow> ./launch.sh

[Wait for completion]

```

And the results will be stored within the current folder within the **results** folder.

SnakeMake

If the chosen workflow manager is SnakeMake, the next step is to go inside the folder:

```

$covid-19-workflow/Workflow> cd SnakeMake
$covid-19-workflow/Workflow/SnakeMake> ls
config.yml  launch.sh  run.sh  Snakefile  split.sh

```

config.yml

Configuration file.

launch.sh

Script that launches the workflow with the testing dataset using SLURM.

run.sh

Script that launches the workflow with the testing dataset.

Snakefile

This is the workflow script.

split.sh

Helper script required by the Snakefile.

The way to run the workflow is:

```

$covid-19-workflow/Workflow/SnakeMake> ./run.sh

[Wait for completion]

```

And the results will be stored within the current folder within the **results** folder.

3.3.2 Supercomputer

All PerMedCoE Building Blocks and Workflows can be deployed and executed in Supercomputers/Clusters.

Their deployment can be performed for each user in a Supercomputer following the same instructions as with [local deployment](#) (the requirements are the same, as well as the workflow manager and Singularity). Note that it is required that the Supercomputer's login node have access to Internet.

The main difference relies in the execution, since the Supercomputers have a job queuing system (e.g. SLURM), and the workflows have to be submitted as jobs. To this end, the workflows have helper scripts to automatize the job submission.

This Section will go step-by-step showing how to use the PerMedCoE artifacts in the following Supercomputers, where the available Building Blocks and Workflows are already deployed and ready to be used by all users with access to that Supercomputer.

3.3.2.1 SCs

MN4

The [MareNostrum 4](#) (MN4) supercomputer is designated as a Special Scientific/Technical Infrastructure Facility by the Spanish Ministry of Economy, Industry and Competitiveness and is part of the PRACE Research Infrastructure as one of the 7 Tier-0 Systems currently available for European scientists.

More details on its structure and specifications can be found in the [MN4 documentation](#).

Requirements

The main requirement is to have access to the MN4 Supercomputer.

With the access granted, the first step is to connect to the MN4 Supercomputer. Specific instructions to this end are provided in the [MN4 documentation](#).

Deployment

It is not necessary to deploy the Building Blocks, neither the Workflows, since they have already been deployed and are available to all users by means of a module.

The users only need to load the required workflow manager (which in this example is PyCOMPSs), singularity, and the `permedcoe` module, which enables to get any of the available workflows:

```
$ export COMPSS_PYTHON_VERSION=3
$ module load COMPSS/3.2
load java/8u131 (PATH, MANPATH, JAVA_HOME, JAVA_ROOT, JAVA_BINDIR, SDK_HOME, JDK_HOME, ↵
↪JRE_HOME)
load papi/5.5.1 (PATH, LD_LIBRARY_PATH, C_INCLUDE_PATH)
load PYTHON/3.7.4 (PATH, MANPATH, LD_LIBRARY_PATH, LIBRARY_PATH, PKG_CONFIG_PATH, C_
↪INCLUDE_PATH, CPLUS_INCLUDE_PATH, PYTHONHOME, PYTHONPATH)
load COMPSS/3.2 (PATH, CLASSPATH, MANPATH, GAT_LOCATION, COMPSS_HOME, JAVA_TOOL_OPTIONS, ↵
↪LDFLAGS, CPPFLAGS)
$ module load singularity/3.5.2
load SINGULARITY/3.5.2 (PATH)
$ module use /apps/modules/modulefiles/tools/COMPSS/libraries
$ module load permedcoe
```

(continues on next page)

(continued from previous page)

```
load permedcoe (PATH, MANPATH, IT_HOME)
Using Python 3.7.4.
```

Now, the `permedcoe` tool is available, as well as some other commands that will ease the workflow get. In particular, the available commands are:

get_covid19workflow

Invoking this command will copy the Covid19 workflow in the current directory. Since all building blocks are available, there is no need to do anything else.

get_drug_synergies_workflow

Invoking this command will copy the Drug Synergies workflow in the current directory. Since all building blocks are available, there is no need to do anything else.

get_single_drug_prediction_workflow

Invoking this command will copy the Single Drug Prediction workflow in the current directory. Since all building blocks are available, there is no need to do anything else.

get_cancer_invasion_workflow

Invoking this command will copy the Cancer Invasion workflow in the current directory. Since all building blocks are available, there is no need to do anything else.

For example, if the target workflow is Cancer Invasion, it is only necessary to:

```
$ get_cancer_invasion_workflow
$ ls
cancer-invasion-workflow
$ cd cancer-invasion-workflow
$cancer-invasion-workflow> ls
BuildingBlocks LICENSE README.md Resources Tests Workflow
```

The result is the same as in the local deployment.

Execution

Once achieved the workflow sources, the next step is to go inside the `Workflow` folder, and inside the folder for the selected workflow manager (which in this case is `PyCOMPSS`):

```
$cancer-invasion-workflow> cd Workflow/PyCOMPSS
$cancer-invasion-workflow/Workflow/PyCOMPSS> ls
clean.sh launch.sh README.md results run.sh src
```

Inside this folder, there is a `launch.sh` script that submits the execution of the Cancer Invasion workflow to the queuing system (SLURM).

Important: This script contains the workflow input parameters. So, if you want to run it with different configurations, it will be necessary to edit it and update the required parameters (e.g. other dataset, more replicas, or more simulation time).

```
$cancer-invasion-workflow/Workflow/PyCOMPSS> ./launch.sh
load java/8u131 (PATH, MANPATH, JAVA_HOME, JAVA_ROOT, JAVA_BINDIR, SDK_HOME, JDK_HOME, ↵
↪ JRE_HOME)
load papi/5.5.1 (PATH, LD_LIBRARY_PATH, C_INCLUDE_PATH)
```

(continues on next page)

(continued from previous page)

```

load PYTHON/3.7.4 (PATH, MANPATH, LD_LIBRARY_PATH, LIBRARY_PATH, PKG_CONFIG_PATH, C_
↳ INCLUDE_PATH, CPLUS_INCLUDE_PATH, PYTHONHOME, PYTHONPATH)
load COMPSs/3.2 (PATH, CLASSPATH, MANPATH, GAT_LOCATION, COMPSS_HOME, JAVA_TOOL_OPTIONS,
↳ LD_FLAGS, CPPFLAGS)
load SINGULARITY/3.5.2 (PATH)
remove permedcoe (PATH, MANPATH, IT_HOME)
Using Python 3.7.4.
load permedcoe (PATH, MANPATH, IT_HOME)
Using Python 3.7.4.
SC Configuration:          default.cfg
JobName:                   COMPSs
Queue:                     default
Deployment:                 Master-Worker
Reservation:               disabled
Num Nodes:                 3
Num Switches:              0
GPUs per node:             0
Job dependency:            None
Exec-Time:                 02:00:00
QoS:                       debug
Constraints:               disabled
Storage Home:              null
Storage Properties:
Storage container image:   false
Storage cpu affinity:      disabled
Other:                     --wall_clock_limit=7140
                           --sc_cfg=default.cfg
                           --qos=debug
                           --worker_working_dir=/gpfs/projects/bscXX/bscXXYYY/PROJECTS/PerMedCoE/cancer-
↳ invasion-workflow/Workflow/PyCOMPSs
                           --log_level=off
                           --graph
                           --tracing
                           --generate_trace=true
                           --python_interpreter=python3 /gpfs/projects/bscXX/bscXXYYY/PROJECTS/
↳ PerMedCoE/cancer-invasion-workflow/Workflow/PyCOMPSs/src/cancer_invasion.py /gpfs/
↳ projects/bscXX/bscXXYYY/PROJECTS/PerMedCoE/cancer-invasion-workflow/Workflow/PyCOMPSs/.
↳ ../../Resources/data//parameters_small.csv /gpfs/projects/bscXX/bscXXYYY/PROJECTS/
↳ PerMedCoE/cancer-invasion-workflow/Workflow/PyCOMPSs/results/ 5 4500

Temp submit script is: /scratch/tmp/tmp.dXN9zJ0u7
Requesting 144 processes
Submitted batch job 30404848

```

The result of submitting the job is its identifier, which in this example is 30404848. The job status can be checked with `queue`:

```

$ queue
JOBID PARTITION  NAME      USER ST      TIME  NODES NODELIST(REASON)
30404848      main    COMPSs  bscXXYYY R        1:37      3 s01r1b25,s24r1b[60,63]

```

Which in this example, shows that it has already started and the workflow is being executed.

After executing the whole workflow, three main elements will appear:

compss-30404848.err

This file contains the standard error messages shown during the execution. Any issue will appear in this file.

compss-30404848.out

This file contains the standard output messages shown during the execution. In particular, it displays the Py-COMPSs execution output.

results

This folder will contain the workflow results:

```
$ cd results
$results> tree
.
├── plots
│   ├── migration_bias.png
│   └── migration_bias_ratio.png
└── simulations
    ├── parameter_0
    │   └── [intermediate results]
    ├── parameter_1
    │   └── [intermediate results]
    ├── parameter_10
    │   └── [intermediate results]
    ├── parameter_2
    │   └── [intermediate results]
    ├── parameter_3
    │   └── [intermediate results]
    ├── parameter_4
    │   └── [intermediate results]
    ├── parameter_5
    │   └── [intermediate results]
    ├── parameter_6
    │   └── [intermediate results]
    ├── parameter_7
    │   └── [intermediate results]
    ├── parameter_8
    │   └── [intermediate results]
    └── parameter_9
        └── [intermediate results]
```

And thats it!

CREATING BUILDING BLOCKS

This section describes how to develop Building Blocks using the **permedcoe** package, and how to use them in an application. Details are also provided on application execution using the Croupier meta-orchestrator. A step-by-step tutorial for Building Block creation can also be found under this section.

4.1 Building Block Templates

4.1.1 Building Block template creation

The first suggested step is to start with a template created with the **permedcoe** command:

```
permedcoe template building_block my_building_block
```

The result of this command is a folder containing a python package with all scripts and base code to start developing your Building Block.

Tip: More than one Building Block can be included within the created package.

Hint: The Building Block template creation provides the main actions to be performed in order to complete your building block:

```
Creating Building Block template
-----
To be completed:

- definitions.py:(11): TODO: Define your container.
- main.py:(35): TODO: (optional) Pure python code calling to PyCOMPSs tasks (that can be
  ↳ defined in this file or in another).
- main.py:(39): TODO: Define the binary to be used (can be within my_building_block_
  ↳ ASSETS_PATH (e.g. my_binary.sh)).
- main.py:(40): TODO: Define the inputs and output parameters.
- main.py:(41): TODO: Define a representative task name.
- main.py:(42): TODO: Define the binary parameters.
- main.py:(43): TODO: Define the binary parameters.
- main.py:(44): TODO: Define the binary parameters.
- main.py:(45): TODO: Add tmpdir=TMPDIR if the tmpdir will be used by the asset script.
- main.py:(71): TODO: Define the arguments required by the Building Block in definition.
  ↳ json file.
```

(continues on next page)

(continued from previous page)

```
- main.py:(73): TODO: Declare how to run the binary specification (convert config into_
↳building_block_task call).
- __main__.py:(13):      TODO: Add require_tmpdir=True if the asset requires to write_
↳within the tmpdir.
-----
```

4.1.2 Package structure

The package contains the following scripts:

```
.
├── build.sh
├── clean.sh
├── container
│   ├── create_container.sh
│   └── sample.def
├── install_sc.sh
├── install.sh
├── pyproject.toml
├── README.md
├── setup.cfg
├── setup.py
├── src
│   └── my_building_block
│       ├── assets
│       │   └── my_binary.sh
│       ├── definition.json
│       ├── definitions.py
│       ├── __init__.py
│       ├── __main__.py
│       └── main.py
└── uninstall.sh
```

File name	Description
build.sh	Script that builds the package
clean.sh	Script that cleans build files
container/create_container.sh	Script that builds the container defined in <i>sample.def</i>
container/sample.def	Container definition file
install_sc.sh	Manual installation script for supercomputer
install.sh	Manual installation script
pyproject.toml	Building requirements
README.md	Package description
setup.cfg	Package configuration (Pypi metadata)
setup.py	Package setup
src/my_building_block/	Folder that contains the building block
src/my_building_block/assets/	Folder that contains the building block assets
src/my_building_block/assets/my_binary.sh	Asset script example
src/my_building_block/definition.json	Building Block parameters definition file
src/my_building_block/definitions.py	Building Block global definitions
src/my_building_block/__init__.py	Package import resolver
src/my_building_block/__main__.py	Building block invocation file
src/my_building_block/main.py	Building block main file
uninstall.sh	Uninstall script

4.1.3 Building Block structure

There are a set of rules to implement a PerMedCoE compliant Building Block:

- Complete *definition.json* file with the Building Block required parameters. - Declare inputs and outputs.
- Complete *definitions.py* with the appropriate container file name. - Define the container/s within *definitions.py* file (*CONTAINER* variable).
- Provide a Python script *main.py* with the following structure and adapt to your needs (check all *TODO* marked lines):
 - Sample *main.py*:

```
# Decorator imports
from permedcoe import constraint      # To define constraints needs (e.g. ↪
↪number of cores)
from permedcoe import container      # To define container related needs
from permedcoe import binary        # To define binary to execute related ↪
↪needs
from permedcoe import mpi            # To define an mpi binary to execute ↪
↪related needs (can not be used with @binary)
from permedcoe import task           # To define task related needs
# @task supported types
from permedcoe import FILE_IN        # To define file type and direction
from permedcoe import FILE_OUT       # To define file type and direction
from permedcoe import FILE_INOUT     # To define file type and direction
from permedcoe import DIRECTORY_IN   # To define directory type and direction
from permedcoe import DIRECTORY_OUT  # To define directory type and direction
from permedcoe import DIRECTORY_INOUT # To define directory type and direction
# Other permedcoe available functionalities
from permedcoe import Arguments      # Arguments definition
```

(continues on next page)

(continued from previous page)

```

from permedcoe import get_environment # Get variables from invocation (tmpdir,
↳ processes, gpus, memory)
from permedcoe import TMPDIR          # Default tmpdir key

# Import single container and assets definitions
from NEW_NAME.definitions import NEW_NAME_ASSETS_PATH # binary could be in
↳ this folder
from NEW_NAME.definitions import NEW_NAME_CONTAINER
from NEW_NAME.definitions import COMPUTING_UNITS

def function_name(*args, **kwargs):
    """Extended python interface:
    To be used only with PyCOMPSSs - Enables to define a workflow within the
    ↳ building block.
    Tasks are not forced to be binaries: PyCOMPSSs supports tasks that are pure
    ↳ python code.

    # PyCOMPSSs help: https://pycompss.readthedocs.io/en/latest/Sections/02_App_
    ↳ Development/02_Python.html

    Requirement: all tasks should be executed in a container (with the same
    ↳ container definition)
    to ensure that they all have the same requirements.
    """
    print("Building Block entry point to be used with PyCOMPSSs")
    # TODO: (optional) Pure python code calling to PyCOMPSSs tasks (that can be
    ↳ defined in this file or in another).

@container(engine="SINGULARITY", image=NEW_NAME_CONTAINER)
@binary(binary="cp") # TODO: Define the
↳ binary to be used (can be within NEW_NAME_ASSETS_PATH (e.g. my_binary.sh)).
@task(input_file=FILE_IN, output_file=FILE_OUT) # TODO: Define the
↳ inputs and output parameters.
def building_block_task( # TODO: Define a
↳ representative task name.
    input_file=None, # TODO: Define the
↳ binary parameters.
    output_file=None, # TODO: Define the
↳ binary parameters.
    verbose="-v"): # TODO: Define the
↳ binary parameters.
    # TODO: Add tmpdir=TMPDIR if the tmpdir will be used by the asset script.
    """Summary.

    The Definition is equal to:
    cp <input_file> <output_file> -v
    Empty function since it represents a binary execution:

    :param input_file: Input file description, defaults to None
    :type input_file: str, optional
    :param verbose: Verbose description, defaults to "-v"
    :type verbose: str, optional

```

(continues on next page)

(continued from previous page)

```

# :param tmpdir: Temporary directory, defaults to TMPDIR
# :type tmpdir: str, optional
"""
pass

def invoke(arguments, config):
    """Common interface.

    Args:
        arguments (args): Building Block parsed arguments.
        config (dict): Configuration dictionary.

    Returns:
        None
    """
    # TODO: Define the arguments required by the Building Block in definition.
    ↪ json file.

    # TODO: Declare how to run the binary specification (convert config into ↪
    ↪ building_block_task call).
    # Sample config parameter get:
    #     operation = config["operation"]
    # Then operation can be used to tune the building_block_task parameters or ↪
    ↪ even be a parameter.
    # Sample permedcoe environment get:
    #     env_vars = get_environment()
    # Retrieves the extra flags from permedcoe.
    input_file = arguments.model
    output_file = arguments.result
    # tmpdir = arguments.tmpdir
    building_block_task(input_file=input_file,
                        output_file=output_file)
    # tmpdir=tmpdir)

```

- Use the decorators provided by *permedcoe* package. They provide the capability to use the BB in various workflow managers transparently. In other words, the BB developer does not have to deal with the peculiarities of the workflow managers.
- A BB can be a single executable, but it can be a more complex code if the *NEW_NAME_extended* function is implemented and used with PyCOMPSs.
- It is necessary to have function (*invoke*) with a specific signature: (*arguments, config*).
- The *invoke* function provides the command line interface for the BB as shown in the [usage](#usage) section. In addition, it parses the Yaml config file and invokes the *NEW_NAME* function with the appropriate parameters.
- The BB *binary* must be defined with the *@task*, *@binary* and *@container* decorators (*NEW_NAME_task*). This function needs to declare the binary flags, and it is invoked from the *NEW_NAME* function.
- The *@task* decorator must declare the type of the file or directories for the binary invocation. In particular, using the parameter name and *FILE_IN/FILE_OUT/DIRECTORY_IN/DIRECTORY_OUT* to define if the parameter is a file or a directory and if the binary is consuming the file/directory or it is producing it.
- Uncomment *tmpdir* variable if the binary uses an asset that requires to writing permissions. So the asset writes in a controlled temporary directory. Don't forget to set *require_tmpdir=True* in *__main__.py* within the *invoker* call.

4.1.4 Deployment

4.1.4.1 Installation

The package provides two ways to install this package (from Pypi and manually):

- From Pypi:

After uploading the package to Pypi it can be installed as usual Python packages:

```
pip install my_building_block
```

or more specifically:

```
python3 -m pip install my_building_block
```

- From source code:

This package provides an automatic installation script, but it is necessary to install the `permedcoe` package before the `my_building_block` package since it is required by `my_building_block`.

```
# Install permedcoe package
git clone https://github.com/PerMedCoE/permedcoe.git
cd permedcoe
./install.sh
# Install my_building_block
cd ../my_building_block
./install.sh
```

4.1.4.2 Usage

The `my_building_block` package provides a clear interface that allows it to be used with multiple workflow managers (e.g. PyCOMPSs, NextFlow and Snakemake).

- Command line interface:

Once installed the `my_building_block` package, it provides the `my_building_block` command, that can be used from the command line. For example:

```
$ my_building_block -h
usage: my_building_block [-h] --model MODEL --result RESULT [-c CONFIG] [-d] [-l
→{debug,info,warning,error,critical}] [--tmpdir TMPDIR] [--processes PROCESSES] [--
→gpus GPUS] [--memory MEMORY]
                        [--mount_points MOUNT_POINTS]

my_building_block Building Block short description. Give more details about the
→Building Block.

options:
  -h, --help            show this help message and exit
  --model MODEL          (INPUT - str (file)) Input file (model)
  --result RESULT        (OUTPUT - str) Result file
  -c CONFIG, --config CONFIG
                        (CONFIG) Configuration file path
  -d, --debug            Enable Building Block debug mode. Overrides log_level
```

(continues on next page)

(continued from previous page)

```

-l {debug,info,warning,error,critical}, --log_level {debug,info,warning,error,
→critical}
                                Set logging level
--tmpdir TMPDIR                 Temp directory to be mounted in the container
--processes PROCESSES           Number of processes for MPI executions
--gpus GPU                      Requirements for GPU jobs
--memory MEMORY                Memory requirement
--mount_points MOUNT_POINTS     Comma separated alias:folder to be mounted in the container

```

This interface can be used within any workflow manager that requires binaries (e.g. NextFlow and Snakemake).

In addition, it can be used with PyCOMPSs by importing the decorated function or any other specific for PyCOMPSs.

```

from my_building_block import building_block_task

building_block_task(input=None,
                    output=None)

```

- Extension for PyCOMPSs:

Moreover, a BB can also implement a Python function not limited to the input (file/s or directory/ies), output (file/s or directory/ies) and config (yaml file) signature. This enables application developers to use the BB with PyCOMPSs using Python objects instead of files among BBs.

```

from my_building_block import function_name

function_name(*args, **kwargs) # specific interface

```

4.1.4.3 Uninstall

Uninstall can be done as usual pip packages:

There are two ways to uninstall this package, that depends on the way that it was installed (from Pypi or using `install.sh`):

- From Pypi:

```
pip uninstall my_building_block
```

or more specifically:

```
python3 -m pip uninstall my_building_block
```

- From manual installation (using `install.sh`):

```
./uninstall.sh
```

And then the folder can be cleaned as well using the `clean.sh` script.

```
./clean.sh
```

4.1.5 Best practices

There are a set of best practices suggested to BB developers:

- Use a code style:
 - `pep8`
 - `black`
- Document your BB.

4.2 Application Templates

4.2.1 Application template creation

The first suggested step is to start with a template created with the `permedcoe` command:

```
permedcoe template application my_application
```

The result of this command is a folder containing a three folders, one for the three supported workflows managers (PyCOMPSs, Nextflow and Snakemake), where a template for each of them is provided.

Hint: The application template creation provides the main actions to be performed in order to complete your building block:

```
-----  
To be completed:
```

```
- app.py:(9):      TODO: Import the desired building blocks and use invoke or any other  
↪function.  
- Snakefile:(0):   TODO: Declare the building blocks to be used as rules.  
- Snakefile:(9):   TODO: Change bb to the building block name.  
- NextFlow.nf:(7): TODO: Declare the building blocks to be used as process.  
- NextFlow.nf:(18): TODO: Change bb to the building block name.  
-----
```

Note: The template provides support for PyCOMPSs, Nextflow and Snakemake, but it is not mandatory to implement the application for all of them. However, this decision will tight your application to a single workflow manager that needs to be available where the execution is going to take place.

4.2.2 Folder structure

The application contains the following scripts:

```

.
├── NextFlow
│   ├── launch.sh
│   └── NextFlow.nf
├── PyCOMPSs
│   ├── app.py
│   ├── launch.sh
│   └── launch_without_pycompss.sh
├── SnakeMake
│   ├── launch.sh
│   └── Snakefile

```

File name	Description
NextFlow	Folder containing the Nextflow application template
NextFlow/launch.sh	Launch with Nextflow script
NextFlow/NextFlow.nf	Application template for Nextflow
PyCOMPSs	Folder containing the PyCOMPSs application template
PyCOMPSs/app.py	Application template for PyCOMPSs
PyCOMPSs/launch.sh	Launch with PyCOMPSs script
PyCOMPSs/launch_without_pycompss.sh	Launch without PyCOMPSs script
SnakeMake	Folder containing the Snakemake application template
SnakeMake/launch.sh	Launch with Snakemake script
SnakeMake/Snakefile	Application template for Snakemake

The developer responsibility is to complete at least one of the following files:

- NextFlow.nf
- app.py
- Snakefile

And the mainly expected content of the application is the usage of building blocks.

Tip: Due to the Python nature of PyCOMPSs applications, it can also support python code and objects among building blocks, enabling the implementation of complex workflows with inner parallelism.

4.2.3 Execution

The execution of the application depends on the workflow manager choosen. Consequently, the execution will be performed following the workflow manager instructions.

For example, the execution with PyCOMPSs locally is performed with the `runcompss` command (see [runcompss documentation](#) for more details), whilst for supercomputers is performed with the `enqueue_compss` command (see [enqueue_compss documentation](#) for more details).

4.2.4 Best practices

There are a set of best practices suggested to application developers:

- Use a code style if using PyCOMPSs:
 - `pep8`
 - `black`
- Document your Application.

4.3 Application Execution with Croupier

4.3.1 Introduction

Croupier is a meta-orchestrator that enables end-users, either application providers and consumers to deploy and execute applications across multiple HPC infrastructures, and move data from one infrastructure to another. Application providers use Croupier to make their applications available for consumers in the Croupier marketplace. Application consumers browse available applications and execute them in selected HPC infrastructures.

4.3.2 Prerequisites

The Croupier administrator should take care of installing all required dependencies and the Croupier plugin. Atos make Croupier ecosystem accessible through its frontend at <http://frontend.croupier.ari-aidata.eu>.

4.3.3 Croupier's framework

Croupier framework consist of several interconnected services, namely:

- Cloudify (<https://cloudify.co/>) is the Cloud workflow engine that hosts HPC-based workflow executions by using Croupier as a plugin.
- Keycloak (<https://www.keycloak.org/>) is an IAM service that offers a SSO across multiple application. Croupier frontend uses KeyCloak to authenticate users
- Hashicorp Vault (<https://www.vaultproject.io>) is a secret store. Croupier frontend uses Vault to retrieve HPC user's credentials to get access to the target HPC frontend on behalf of the user.
- Vault Secret Uploader (<https://github.com/ari-apc-lab/vault-secret-uploader>) offers an REST API interface for CRUD operations of Vault secrets. This API is used by users to store their credentials
- HPC Exporter (<https://github.com/ari-apc-lab/hpc-exporter/tree/nojwt>) collects HPC partition/queue and job execution metrics when requested by the main monitoring engine (i.e. Prometheus)
- Prometheus (<https://prometheus.io/>) is the main monitoring engine in the Croupier framework, storing collected metrics and offering SQL-like querying language PROMQL.
- Grafana (<https://grafana.com/>) is the main metrics visualization tool used to render HPC partition/queue and job dashboards
- Grafana Registry (<https://github.com/ari-apc-lab/grafana-registry>) tool is used to register Grafana dashboards for users executing workflows
- Croupier backend (<https://github.com/ari-apc-lab/croupier-backend/tree/permedcoe>) is the backend implementation of the Croupier Web-based interface. This component interacts with Cloudify for workflow management and execution

- Croupier frontend (<https://github.com/ari-apc-lab/croupier-frontend/tree/permedcoe>) is the frontend implementation of the Croupier Web-based interface. It provides the Web UI that interfaces application consumers

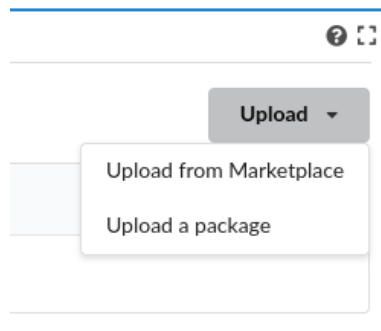
Contact Croupier administrator (jesus.gorronogoitia@atos.net) for instructions to deploy and configure instances of these services required by Croupier.

Kubernetes (k8s) manifests for the manual deployment of the Croupier services listed above are available at GitHub: <https://github.com/ari-apc-lab/k8s-resources/tree/permedcoe>

Helm charts for k8s semi-automatic deployment are available at (e.g. some manual configuration is still required): <https://github.com/ari-apc-lab/k8s-resources/tree/permedcoe/helm-charts>

4.3.3.1 Croupier plugin installation




To install Croupier plugin, you need a Croupier Wagon file, compiled for CentOS. Contact Croupier administrator (jesus.gorronogoitia@atos.net) for the latest Croupier wheel and the `plugin.yaml` descriptor. Next, log into Cloudify the with **admin** account. Click on the **Resources** tab in the leftmost panel. Next, click on the **Plugins** tab in the main page. Then, click on the **upload** button on the right of the **Plugin list** table. Select **upload a package**



Then, in the wizard, browse click on the **File** button to search in your file system for the **Croupier Wagon file**, and similarly for the **Croupier YAML file**. Once selected, check the Plugin title is set to `croupier`. Click on the **Upload** button to accept.

 A screenshot of the 'Upload plugin' wizard in Cloudify. The form has four sections: 'Wagon file' with a file input containing 'croupier-3.2.0-py36-none-linux_x86_64.wgn'; 'YAML file' with a file input containing 'plugin.yaml'; 'Plugin title' with a text input containing 'croupier'; and 'Icon file' with a URL input and a 'Provide the plugin's icon file URL or click browse to select a file' prompt. At the bottom right are 'Cancel' and 'Upload' buttons.

Check the Croupier plugin installation in the list of plugins.

Plugins list									
<div>Search...</div> <div>Upload</div>									
	Plugin	Package name	Package version	Supported platform	Distribution	Distribution release	Uploaded at	Creator	
ID	 croupier	 croupier	3.2.0	linux_x86_64	centos	core	02-02-2022 13:26	admin	

4.3.4 Application definition (Blueprint)

Application providers define their applications as meta-workflows that execute multiple tasks (in sequence or in parallel) distributed across one or more target HPC infrastructures. These workflows are named **blueprints** in Cloudify terminology. They may also specify data objects, their role as tasks' inputs and/or outputs, the data infrastructures where those data objects are located at and the transfer entities that move such data objects from one source to a target. Croupier's workflows are specified in YAML by using the **OASIS TOSCA** language (<https://docs.cloudify.co/latest/developer/blueprints/>). For the following, we use the Covid19 application as an example.

The application workflow starts with a header that at least declares the imports to use the Croupier plugin. Other imports could be possible if required by the application.

```
tosca_definitions_version: cloudify_dsl_1_3
imports:
  - http://raw.githubusercontent.com/ari-apc-lab/croupier/master/resources/
    ↪ types/cfy_types.yaml
  - plugin:croupier
```

Next, application data types can be optionally declared. In the following example, the Covid19 application input arguments are declared

```
data_types:
permedcoe.covid19.args:
  properties:
    metadata:
      type: string
      required: true
    model_prefix:
      type: string
      required: true
    outdir:
      type: string
      required: true
    ko_file:
      type: string
      required: true
    reps:
      type: integer
      required: true
    model:
      type: string
      required: true
    data_folder:
      type: string
      required: true
    simulation_time:
```

(continues on next page)

(continued from previous page)

```

type: integer
required: true

```

These data types are application specific, and determined by the application provider.

Then, the application inputs should be declared. There is a number of common inputs for a common application, whose examples below are taken from Covid19 app (**Note:** some of these concrete inputs are application specific, determined by the application provider. Common ones are mentioned below):

- **VAULT arguments required to obtain access credentials for target HPC and Data Service (DS), namely:**
 - iam_jwt: jwt token issued by Keycloak as a result of a valid user authentication
 - iam_user: Keycloak user
- **HPC infrastructure properties:**
 - hpc_host: HPC frontend endpoint
 - hpc_scheduler: HPC scheduler used for task schedule. Valid supported options: SLURM, PBS, PYCOMPSS
 - hpc_scheduler_modules: optional list of module commands required to enable the hpc_scheduler (Note: this may be required for PYCOMPSSs)
 - monitor_scheduler: HPC scheduler use for monitoring. Valid supported options: SLURM, PBS
- **Data access infrastructures:**
 - hpc_dai_host: Data Access Infrastructure (DAI) hosting some data sources
 - hpc_dai_internet_access: boolean flag specifying if DAI host supports internet access
- List of PYCOMPSSs arguments, defined by type croupier.datatypes.pycompss_options.
- **Application specific args:**
 - job_pre_script: optional list of bash script commands to executed before task submission
 - covid19_args: list of Covid19 arguments

The number and type of arguments are variable and they are decided by the application provider. For instance, several Vault services can be defined, sharing the same vault_user, but requiring different tokens. Several HPC infrastructures can be used to distribute workflow tasks, requiring dedicated configuration inputs for each infrastructure, hence. Similarly for data access infrastructures that host the data to be consumed or produced by workflow tasks. The number and kind of PYCOMPSSs arguments can be different across applications.

Note: This COVID-19 workflow example uses PYCOMPSSs as the workflow scheduler.

```

inputs:
##### VAULT #####
iam_jwt:
  type: string
iam_user:
  type: string

##### HPC Infrastructures #####
hpc_host:
  type: string

hpc_scheduler:

```

(continues on next page)

(continued from previous page)

```

    type: string

hpc_scheduler_modules:
    type: list

monitor_scheduler:
    type: string

##### DS Infrastructures #####
hpc_dai_host:
    type: string

hpc_dai_internet_access:
    type: boolean

target_dai_host:
    type: string

target_dai_internet_access:
    type: boolean

##### Covid 19 application #####
job_pre_script:
    type: list

covid19_args:
    type: permedcoe.covid19.args

##### PYCOMPSSs args #####
pycompss_args:
    type: croupier.datatypes.pycompss_options

```

Next, if the application workflow needs to collect task execution metrics for monitoring, one `hpc_exporter` instance, of type of type `croupier.nodes.HPCExporter`, must be declared

```

node_templates:
  hpc_exporter:
    type: croupier.nodes.HPCExporter

```

Then, one or more Vault nodes must be declared. Vault nodes are used as secret stores where to retrieve from the credentials required to access the target HPC infrastructures, through ssh, to schedule workflow's tasks (as jobs). The following block declares one Vault node of type `croupier.nodes.Vault`. Note that Vault properties (`jwt` and `user`) are taken from inputs by using the `get_input` function:

```

vault:
  type: croupier.nodes.Vault
  properties:
    jwt: { get_input: iam_jwt }
    user: { get_input: iam_user }

```

WORKFLOW SPECIFICATION

Then, one of more HPC infrastructures (where to execute the workflow's tasks) are declared as node instances of the type `croupier.nodes.InfrastructureInterface`. The mandatory properties of this type must be overridden by

this node definition. Other optional properties as well. In particular:

- `config/infrastructure_interface` must be given with the name of the target HPC scheduler used to launch job tasks.
- `credentials/host` must also be given with the host name of the HPC frontend.

In this example, HPC configuration is read from declared inputs, as the application's consumer will be prompted to provide those values. This is a common approach when the consumer selects a target HPC infrastructure where to execute the application. Alternatively, a fixed target HPC infrastructure can be specified in the workflow.

- `job_prefix` declare a prefix for naming the submitted jobs.
- `base_dir` declares the path where Croupier folder for workflow execution will be created.
- `monitoring_options/monitor_period` declares the period of Croupier's requests to the HPC frontend to check the task job execution/queue status.
- `monitoring_options/monitor_interface` declares the HPC scheduler used to collect partition/queue and task execution metrics. If not set, the HPC scheduler for task submission is used. It is required if task scheduler is PYCOMPSs, as it cannot be used for monitoring
- `workdir_prefix` declares the name of the folder create for every task job executed. This folder will contain the deployed application, its execution logs and

Finally, the HPC infrastructure node is associated to the Vault node, by using a relationship of type `retrieve_credentials_from_vault` that states that the HPC `credentials` will be retrieved from that node, declared in the target. Similarly, the relationship `interface_monitored_by` establishes the monitoring exporter used to collect HPC partition/queues and task metrics

```
hpc:
  type: croupier.nodes.InfrastructureInterface
  properties:
    config:
      infrastructure_interface: { get_input: hpc_scheduler }
      modules: { get_input: hpc_scheduler_modules}
    credentials:
      host: { get_input: hpc_host }
    job_prefix: croupier
    base_dir: $HOME
    monitoring_options:
      monitor_period: 60
      monitor_interface: { get_input: monitor_scheduler }
    skip_cleanup: true
    workdir_prefix: "covid19-deploy"
  relationships:
    - type: retrieve_credentials_from_vault
      target: vault
    - type: interface_monitored_by
      target: hpc_exporter
```

In a similar way, the workflow provider can define additional HPC infrastructures in case the workflow's tasks are distributed across them.

Next, one or more tasks are defined, as node instances of type `croupier.nodes.Job` or its subclasses. In the following example, a application task to be executed by **PYCOMPSs** is defined, as an instance of type `croupier.nodes.PyCOMPSsJob`:

```

job:
  type: croupier.nodes.PyCOMPSsJob
  properties:
    job_options:
      pre_script: { get_input: job_pre_script }
      app_name: covid19
      app_source: permedcoe_apps/covid19/covid-19-workflow-main/Workflow/
      ↪PyCOMPSs/src
      env:
        - COMPSS_PYTHON_VERSION: 3
        - PERMEDCOE_IMAGES: ${PERMEDCOE_IMAGES}
        - dataset: $HOME/permedcoe_apps/covid19/covid-19-workflow-main/
      ↪Resources/data
      compss_args: { get_input: pycompss_args }
      app_file: '$(pwd)/covid19_pilot.py'
      app_args: { get_input: covid19_args }
    deployment:
      bootstrap: "scripts/deploy.sh"
      revert: "scripts/revert.sh"
      hpc_execution: false
      skip_cleanup: True
    relationships:
      - type: task_managed_by_interface
        target: hpc
      - type: input
        target: data_small
      - type: output
        target: covid_results
      - type: deployment_source
        target: github_data_access_infra

```

Every task type has its own properties, including those inherited from the base type. For tasks of type `croupier.nodes.PyCOMPSsJob`, like in above example, the properties required to define a task are encoded under the `job_options` property:

- `pre_script`: list of commands to be executed before the application is submitted by the PYCOMPSs manager.
- `app_name`: the name of the application
- `app_source`: path to the application task source, from where it will be executed
- `env`: list of environment variables
- `compss_args`: list of PYCOMPSs arguments. See [PYCOMPSs documentation](#) for more details
- `app_file`: path to the application executable file, in the deployed folder
- `app_args`: list of application arguments. Consult the concrete application documentation

Optionally, tasks can include a `deployment` property to request the deployment of the task app, before it is scheduled in the target HPC. This property includes:

- `bootstrap`: the path to the script that deploys the task application. This path is relative to the **blueprint zip** installed in Cloudify. This script is provided by the application workflow's provider.
- `revert`: the path to the script that undeploys the task application
- `hpc_execution`: boolean stating whether or not the script should be executed within the HPC frontend. If false, it will be executed from Cloudify/Croupier host. This is relevant when HPC has not Internet access and

app deployment requires external resources.

Next, the task is declared to be run in a HPC infrastructure by setting a relationship of type `task_managed_by_interface` whose target points at a HPC node declared before. Optionally, tasks inputs and outputs can be declared by using the `input` and `output` relationships, respectively. They refer to **data objects** declared within the **dataflow** specification. See below subsection in **Dataflow Specification**. In case a deployment block has been specified within the properties block, the server source for application deployment can optionally be specified with the `deployment_source` relationship. This is required when this deployment source is not hardcoded in the deployment script, so the application can be deployed from a source to specify.

Note that in this specification of a PYCOMPSs task, some properties are hardcoded by the application provider, while others (e.g. `app_source` of `compss_args`, or `app_args`) are taken from the declared workflow's inputs, by using the `get_input` function. The application provider decides what data must be provided by the consumer as input.

DATAFLOW SPECIFICATION

Besides the specification of the workflow, the application blueprint can include the specification of the dataflow, which consist of the declaration of :

- **data access infrastructures** that host the data consumed/produced by the workflow
- **data objects** consumed/produced by workflow tasks as inputs/outputs
- **data transfer objects** that move data from one source to a target

Data access infrastructures are declared as node templates of type `croupier.nodes.DataAccessInfrastructure`

```
hpc_data_access_infra:
  type: croupier.nodes.DataAccessInfrastructure
  properties:
    endpoint: { get_input: hpc_dai_host }
    internet_access: { get_input: hpc_dai_internet_access }
    supported_protocols:
      - RSync
  relationships:
    - type: retrieve_credentials_from_vault
      target: vault
```

The mandatory `endpoint` property declares the data access infrastructure internet address: **http(s)://<host>:<port>**. `internet_access` property declares whether or not that data infrastructure has access to Internet. Depending on this, the **data transfer** objects can adopt different data transfer strategies. Alike the HPC infrastructure, the `retrieve_credentials_from_vault` relationship can be established to use a declared **Vault** instance for retrieving the user's credentials for accessing this infrastructure.

Data objects are declared as node instances whose types depends on the kind of data object. Currently, there are supported:

- `croupier.nodes.FileDataSource`: data object located at the filesystem of a remote server, typically accessible by (s)ftp or rsync
- `croupier.nodes.WebDataSource`: data object located at a Cloud Web server, accessible by HTTP

Next example, from Covid 19 app, declares a data object of `croupier.nodes.WebDataSource` kind.

```
data_small_source:
  type: croupier.nodes.WebDataSource
  properties:
    resource: /PerMedCoE/covid-19-workflow/tree/main/Resources/data/small
  relationships:
```

(continues on next page)

(continued from previous page)

```
- type: ds_located_at
  target: github_data_access_infra
```

For Web data sources the property `resource` declares the route to the data within the infrastructure it is located, which is declared with the `ds_located_at` relationship.

Next example, from Covid 19 app, declares a data object of `croupier.nodes.FileDataSource` kind.

```
data_small:
  type: croupier.nodes.FileDataSource
  properties:
    filepath: ~/permedcoe_apps/covid19/covid-19-workflow-main/Resources/data/small/
  relationships:
    - type: ds_located_at
      target: hpc_data_access_infra
```

For File data sources, the property `filepath` declares the path where the data is located within the filesystem of the hosting infrastructure.

Data transfer objects declare objects that transfer the data located in their `from_source` relationship into the data source target declared in their `to_target` relationship, by using the data transfer protocol specified in the property `transfer_protocol`

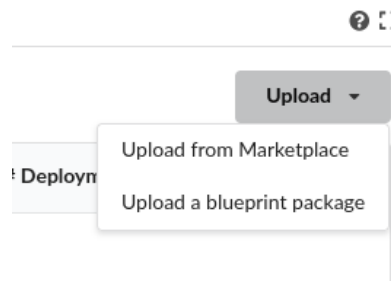
```
dt_http:
  type: croupier.nodes.DataTransfer
  properties:
    transfer_protocol: HTTP
  relationships:
    - type: from_source
      target: data_small_source
    - type: to_target
      target: data_small
```

4.3.5 Application installation (Rol: application provider)

Frontend: Cloudify Web UI: <http://cloudify.croupier.ari-aidata.eu/>

The application provider can deploy an application into Croupier, by taking the following procedure:

- Package the folder that contains the application workflow YAML description (and possibly other artefacts required for its deployment) into a zip file, named with with the application name.
- Log into Cloudify service, go to the **Blueprints** tab in the leftmost panel, click on the **Upload** button located on the right side, select the **Upload a blueprint package** option.



- In the wizard, click on the left button with a folder icon located at the line for the **Blueprint package** field to browse your file system and locate your workflow zip file. When selected, the other wizard fields will be filled in automatically. Then, accept by clicking on **Upload** button.

Upload blueprint

Blueprint package

File

Blueprint name

Blueprint YAML file

Blueprint icon

URL

Cancel Upload

















- Confirm your application is listed in the list of blueprints

Blueprints

Blueprints

Search...

Upload

Name	Created	Updated	Creator	Main Blueprint File	State	# Deployments	
 multihpc-monitoring 	05-07-2022 12:15	05-07-2022 12:15	admin	blueprint.yaml	Uploaded	1	 
 single-drug-prediction 	05-07-2022 10:49	05-07-2022 10:50	admin	blueprint.yaml	Uploaded	1	 
 drug-synergies 	05-07-2022 10:49	05-07-2022 10:49	admin	blueprint.yaml	Uploaded	0	 
 covid19 	05-07-2022 09:37	05-07-2022 09:38	admin	blueprint.yaml	Uploaded	0	 

This procedure is followed by any application provider to deploy her applications into the Croupier marketplace, so that they will be available to be executed by any consumer that gets access.

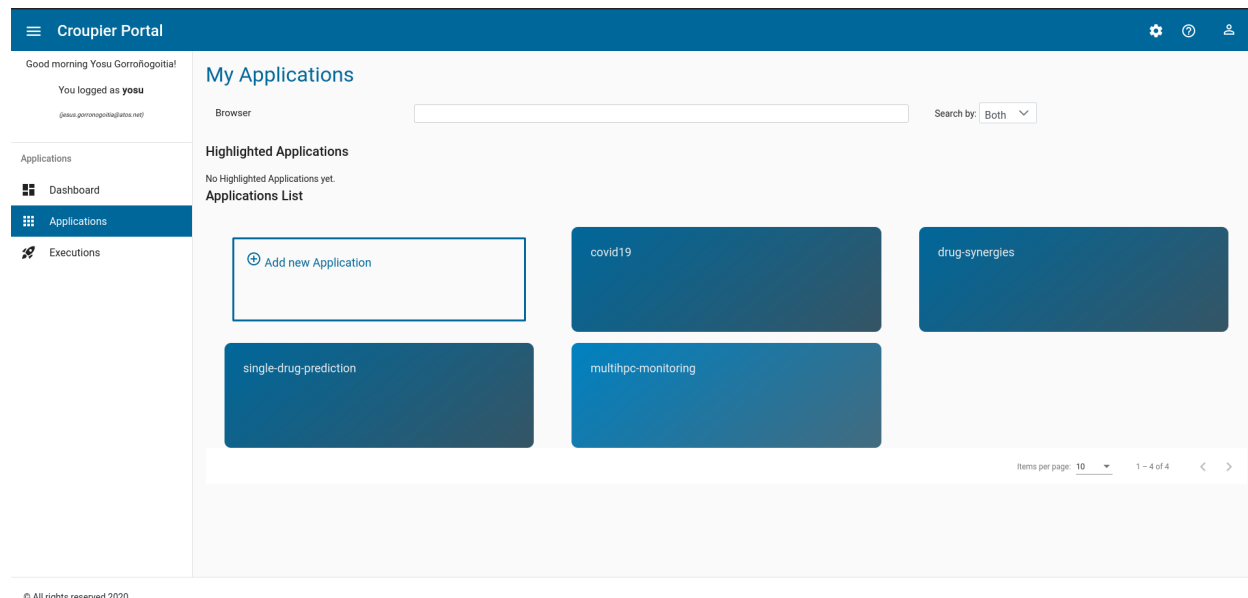
Consumers can execute selected applications (from the list of blueprints) by following a procedure that consists of two steps:

- A consumer's instance of the application (i.e. workflow) is deployed in the target infrastructure(s) with a given set of inputs
- The application's instance is executed in the target infrastructure(s)

4.3.6 Application instance deployment (Rol: application consumer)

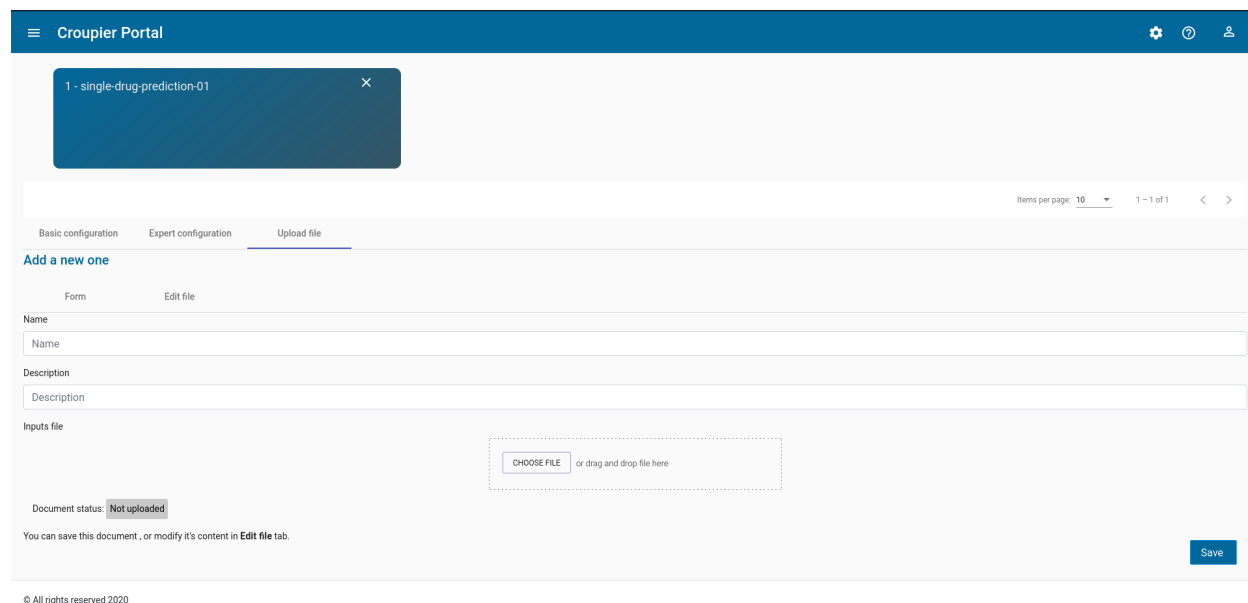
Frontend: Croupier Web UI: <http://frontend.croupier.ari-aidata.eu/>

An application consumer can browse the list of available applications in the Croupier Web UI. For that, open the leftmost option bar (open it by clicking on the icon on the left of the Croupier Portal header), and select Applications. Then, available applications will be displayed in the page.



To deploy a consumer's instance of an application, the consumer takes the following procedure:

- Click on the icon of the selected application. Then, Croupier Web will show a page showing the application details. This page includes three tabs (i.e. Basic configuration, Expert configuration and Upload file) to provide inputs for the application instance that will be created. In this example, we use an external inputs file, so select the Upload file tab



- Click on the Choose file button. Browse your file system and select your application inputs file.

- Next, you can verify the loaded inputs by selecting the **Expert** configuration tab. Verify the loaded inputs, complete or modify them, if needed. Provide a name to your application instance.

The screenshot shows the 'Croupier Portal' interface with the 'Expert configuration' tab selected. The 'Parameters:' section contains the following fields:

- Name: single-drug-prediction-01
- Description: (empty)
- hpc_host: mn1.bsc.es
- hpc_scheduler: PYCOMPSS
- monitor_scheduler: SLURM
- hpc_dai_host: dt01.bsc.es
- hpc_dai_internet_access: (toggle off)
- target_dai_host: sodalite-fe.hirs.de
- target_dai_internet_access: (toggle off)
- single_drug_prediction_args: Open to display the fields of single_drug_prediction_args.
- num_nodes: 2
- exec_time: 15

A 'Save' button is located at the bottom right of the configuration area.

- Once the application inputs' values are provided, click on the **Save button** to create the instance.

An example of `inputs.yaml` file for our Covid19 application is given below:

```
# WORKFLOW
# HPC infrastructures
# HPC
hpc_host: mn1.bsc.es
hpc_scheduler: PYCOMPSS
hpc_scheduler_modules:
  - export COMPSS_PYTHON_VERSION=3
  - module load COMPSS/3.0
  - module load singularity/3.5.2
  - module use /apps/modules/modulefiles/tools/COMPSS/libraries
  - module load permedcoe
monitor_scheduler: SLURM

# COVID19 args
job_pre_script:
-
covid19_args:
  metadata: '${dataset}/metadata_clean.tsv'
  model_prefix: '${dataset}/epithelial_cell_2'
  outdir: '${pwd}/results/'
  ko_file: '${pwd}/ko_file.txt'
  reps: 2
  model: 'epithelial_cell_2'
  data_folder: '${dataset}'
  simulation_time: 100

# PYCOMPSS args
pycompss_args:
```

(continues on next page)

(continued from previous page)

```
num_nodes: 2
exec_time: 45
log_level: 'off'
graph: true
tracing: 'false'
python_interpreter: python3
qos: debug

# DATAFLOW
hpc_dai_host: dt01.bsc.es
hpc_dai_internet_access: false

target_dai_host: sodalite-fe.hlrs.de
target_dai_internet_access: false
```

These consumer’s specific inputs correspond to those declared in the Covid19 application’s workflow specification above. In particular, the consumer specifies the Mare Nostrum 4 as the HPC infrastructure where to deploy the application, as well as PyCOMPSs as its scheduler. Then, the consumer’s required inputs for the Covid19 application are also given, together with few PyCOMPSs execution parameters, which must be tuned according to the size of the Covid19 application inputs. Moreover, the data access infrastructures involved in this application dataflow are also provided.

When the application deployment starts, shows the application instance page. This page gives details about the instance, including associated inputs, in the **Details** tab. The **Logs** tab shows the logs of last executions. The **Executions** tab shows the list of executions.

Croupier Portal

<

SINGLE-DRUG-PREDICTION-01 Details

Details

Logs

Executions

ID:

1

Name:

single-drug-prediction-01

Description:

null

Inputs

iam_jwt

eyJhbGciOiJIUzI1NiIsInR5cCI6Ikpia6Ij0uIiwia2kiOiJ0J0U0aGZlczxqbTFSZmRvbk9PTUdFazRCMVBS1hWMUg2MzEzYmxQUnFFIn0.eyJleHAiOiJlE2NTcwMTI1NzY5Imh0C16MTY1IjNzAXIjWx1NnTubS8rAH0Zs7EB7uxBuT-HQAOmfPhn3orLhJfMjOFGS9UDHgUDZ6o4Zu2Ygfo29GZOD2sQA6G0z9mmjpcHdbH6bZhnog4LlB6RwUnJ2REtZ4LY67Sqa5fezieHane9U9agRU7XjSifWgN_J8LmbNUpCOCEE6Anha9P5_VeZocNmzMBt83Ag-p03Xcaiguwf9AqpZnPCRCIE8teebYeAEYhUjZBSWkolPGvPnq4_YW-mhl12wolhQ

iam_user

yosu

hpc_host

mn1.bsc.es

hpc_scheduler

PYCOMPSS

monitor_scheduler

SLURM

single_drug_prediction_args

cell_list: -cell_list \$(dataset)/cell_list_example.txt
gene_expression: -gene_expression \$(dataset)/Cell_line_RMA_proc_basalExp.txt
genelist: -genelist \$(pwd)/results_csvs/genelist.csv
jax_input: -jax_input \$(dataset)/IC50
network: -network \$(dataset)/network.csv

© All rights reserved 2020

4.3.7 Application instance execution

Once a consumer's instance of the application has been deployed into the target HPC infrastructures, it can be executed.

To start a new execution, click on **Execute** button located at the bottom of the **Details** tab.

To see execution logs, click on the **Logs** tab. You can browse the logs by moving through the pages, filter the logs by type, by event type, and log level. You can refresh the logs page to fetch new ones.

Croupier Portal

SINGLE-DRUG-PREDICTION-01 Details

Details **Logs** Executions

Execution logs:

Type: All Event type: Select a eve... Log levels: Select a level

Status	Timestamp	Type	Workflow	Message
Workflow started	7/5/22, 11:15 AM	Event	run_jobs	Starting 'run_jobs' workflow execution
Workflow node event	7/5/22, 11:15 AM	Event	run_jobs	Queueing job..
Task sent	7/5/22, 11:15 AM	Event	run_jobs	Sending task 'croupier_plugin.tasks.send_job'
Info	7/5/22, 11:16 AM	Log	run_jobs	Executing send_job task job_43s9pu
Info	7/5/22, 11:16 AM	Log	run_jobs	Processing http data transfer proxied by Croupier from source cell_list to target cell_list_target
Info	7/5/22, 11:16 AM	Log	run_jobs	http(wget) data transfer: executing command: cd /tmp/tmpxgfd8i7s; svn export --username jesu-goronogoltia --password ghj_5qcpwKSV3pFAALJw9plgR4dlCuK70uic2z --non-interactive https://github.com/PerMedCoE/single-drug-prediction-workflow/trunk/Resources/data/cell_list_example.txt
Info	7/5/22, 11:16 AM	Log	run_jobs	http(rsync) data transfer: executing command: rsync -ratlz --no-R --no-implied-dirs -e 'ssh -o IdentitiesOnly=yes -o StrictHostKeyChecking=no -i /tmp/tmpx2f2g57e' /tmp/tmpxgfd8i7s/cell_list_example.txt bsc08642@dt01.bsc.es:permedcoe_apps/single-drug-prediction/single-drug-prediction-workflow-main/Resources/data/cell_list_example.txt
Info	7/5/22, 11:16 AM	Log	run_jobs	Submitting the job job_43s9pu
				Invoking command: export COMPSS_PYTHON_VERSION=3; module load COMPSS/3.0; module load singularity/3.5.2; module use /apps/modules/modulefiles/tools/COMPSS/libraries; module load permedcoe; unset PYTHONSTARTUP; unset PYTHONHOME; export PERMEDCOE_IMAGES=\${PERMEDCOE_IMAGES}; export COMPUTING_UNITS=1; export

© All rights reserved 2020

To see the list of the instance executions, click on the **Executions** tab. For each execution, timing, termination status, and the occurrence of errors is reported.

Croupier Portal

SINGLE-DRUG-PREDICTION-01 Details

Details Logs **Executions**

Executions

Created on	Finished on	Status	Has errors	Errors	Open
7/5/22, 11:15 AM	7/5/22, 11:39 AM	terminated	✓	0	Open

<< < 1 > >>

© All rights reserved 2020

As a summary, the frontend dashboard (accessible from the leftmost option panel) shows all deployed instances (for any application) and the executions

Good morning Yosu Gorroñogaitia!
You logged as **yosu**
(yosu.gorroñogaitia@jatos.net)

Applications

- Dashboard
- Applications
- Executions

Dashboard

Browser Search by: Both

Instances list

Name: multihpc-monitoring-01

App: multihpc-monitoring

Name: single-drug-prediction-01

App: single-drug-prediction

Items per page: 10 1 - 2 of 2 < >

Latests Executions

ID: 69e66e50-ff8a-42c0-a640-836fcee737bb

Instance: 2

ID: c8ee1c22-6429-405d-990e-2ae2fb103f2b

Instance: 1

© All rights reserved 2020

4.4 Tutorial

This section provides a step-by-step by tutorial on how to develop Building Blocks using the **permedcoe** package and an application that uses them.

4.4.1 Requirements

It is **required** to install the **permedcoe** package. To this end, you can use **pip**:

```
python3 -m pip install permedcoe
```

Tip: Alternatively, it is possible to be [installed from source](#)

And it is also **required** to install singularity. Please, check the [Singularity installation documentation](#).

4.4.2 Step-by-step

Step 1: Create a Building Block

The first step to create a building block is to generate a template to start with. The template can be created with the following **permedcoe** command:

```
$ permedcoe template building_block my_building_block
Creating Building Block template
-----
To be completed:
- definitions.py:(11):      TODO: Define your container.
- main.py:(35):           TODO: (optional) Pure python code calling to PyCOMPSS tasks (that...
```

(continues on next page)

(continued from previous page)

```

↪can be defined in this file or in another).
- main.py:(39):      TODO: Define the binary to be used (can be within my_building_block_
↪ASSETS_PATH (e.g. my_binary.sh)).
- main.py:(40):      TODO: Define the inputs and output parameters.
- main.py:(41):      TODO: Define a representative task name.
- main.py:(42):      TODO: Define the binary parameters.
- main.py:(43):      TODO: Define the binary parameters.
- main.py:(44):      TODO: Define the binary parameters.
- main.py:(45):      TODO: Add tmpdir=TMPDIR if the tmpdir will be used by the asset_
↪script.
- main.py:(71):      TODO: Define the arguments required by the Building Block in_
↪definition.json file.
- main.py:(73):      TODO: Declare how to run the binary specification (convert config_
↪into building_block_task call).
- __main__.py:(13):  TODO: Add require_tmpdir=True if the asset requires to write within_
↪the tmpdir.
-----

```

The result of this command is a folder named `my_building_block` containing a python package with all scripts and base code to start developing your Building Block.

Important: The Building Block name in this example is `my_building_block`, but should be defined for your Building Block with a specific name (e.g. tool used inside the Building Block or functionality).

This tutorial will continue using `my_building_block`, so take it into account if you define a different name and swap `my_building_block` with yours.

Step 2: Create your container

The Building Block template already provides a sample container definition located in:

```

$ tree my_building_block/container

my_building_block/container/
├── create_container.sh
└── sample.def

```

This folder contains:

sample.def

Container definition file: Recipe to create the container.

create_container.sh

Container creation script from the container definition file. It invokes Singularity with `sample.def` to generate the container (`sample.sif`).

In order to continue with the tutorial it is necessary to create the sample container from `sample.def` with `create_container.sh`:

```

$ cd my_building_block/container
$ ./create_container.sh

```

Caution: The `create_container.sh` script requires root privileges and assumes that the `singularity` command is available in `$PATH` as root.

If `sudo: singularity: command not found` error happens, the container can be created by defining explicitly the singularity path:

```
$ sudo /usr/local/bin/singularity build sample.sif sample.def
```

Step 3: Develop the Building Block logic

Now its time to add logic to the Building Block template. To this end, you can use your desired text editor or Python IDE.

There are three files that require attention: **definition.json**, **definitions.py** and **main.py**:

```
$ cd my_building_block/src/my_building_block
```

First, edit the `definition.json` file and define the Building Block parameters.

“definition.json” development

In this sample `definition.json` file, there are two parameters defined that will be required when the Building Block is executed through the command line:

- Input: model of file type.
- Output: result of file type.

It includes the Building Block short and long descriptions.

```
{
  "short_description": "my_building_block Building Block short description.",
  "long_description": "my_building_block Building Block short description. Give more_
↳ details about the Building Block.",
  "use_description": "long",
  "parameters": {
    "default": [
      {
        "type": "input",
        "name": "model",
        "format": "file",
        "description": "Input file (model)"
      },
      {
        "type": "output",
        "name": "result",
        "format": "file",
        "description": "Result file"
      }
    ]
  }
}
```

Tip: The `parameters` dictionary can contain multiple elements, so that a Building Block can have multiple operation modes.

Second, edit the *definitions.py* file and define the Building Block container.

“definition.json” development

In the *definitions.py* file, there is one line that requires attention:

```
# ...
CONTAINER = "my_bulding_block.sif"
# ...
```

It must be updated with the container name if changed.

Third and finally, edit the *main.py* file defining the main building block logic.

“main.py” development

In this file, it is necessary to develop the following methods:

- `building_block_task` [*MANDATORY*]
- `invoke` [*MANDATORY*]
- `function_name` [*OPTIONAL*]

The next Subsections describe each method development:

“building_block_task” method development

The `building_block_task` is the core method of the Building Block. It is used to define the Building Block functionality, and requires attention in two main places: its decorators and its parameters.

The `building_block_task` needs to be decorated with the `permedcoe`’s decorators (in the same order from top to bottom):

@container

To define the container to be used for the building block execution.

@binary

To define if the `building_block_task` represents the execution of a binary application (that must be available in the container). It includes the information related to the binary to be executed when the method is invoked. This decorator is *optional*. When this decorator is present, the `building_block_task` method must be empty (include only `pass`). However, the `building_block_task` can contain Python code when it is not defined.

@task

To define that the method `building_block_task` is going to be considered as a single task. It includes the information related to the parameters, such as type and direction (e.g. `FILE_IN`, `FILE_OUT`, `FILE_INOUT`, `DIRECTORY_IN`, `DIRECTORY_OUT`, `DIRECTORY_INOUT`).

Lets consider the following code as example:

```
@container(engine="SINGULARITY", image=my_building_block_CONTAINER)
@binary(binary="cp")
@task(input_file=FILE_IN, output_file=FILE_OUT)
def building_block_task(input_file=None,
                        output_file=None,
                        verbose="-v"):
    # Empty function since it represents a binary execution:
    pass
```

The `building_block_task` is a method that is equivalent to:

```
cp <input_file> <output_file> -v
```

Where the decorators define:

@container(engine="SINGULARITY", image=SAMPLE_CONTAINER)

The container that will be used to execute the Building Block.

It must be updated with the container path for your Building Block.

@binary(binary="cp")

The binary to be executed by the Building Block.

It must be updated with the binary path for your Building Block.

@task(input_file=FILE_IN, output_file=FILE_OUT)

The parameters type and direction. In this example, there are two parameters that are files, one used as input (input_file) and another produced by the binary execution (output_file).

It must be updated with the function's parameters type (if files or directories) and/or direction.

And the function is defined:

```
def building_block_task(input_file=None,
                        output_file=None,
                        verbose="-v"):
```

Each parameter is interpreted in order, and all of them should include the default value to ease the invocation (e.g. None is useful for FILES and DIRECTORIES, whilst for the rest integers or strings is enough).

The two required actions in the function definition are:

- **Define a representative function name** (e.g. building_block_task in the example)
- **Define the function parameters**

Important: Please, check carefully the function parameters as well as the @task parameter definition.

Hint: It can also be a normal python function that calls decorated methods.

This will enable to exploit inner parallelism when used with PyCOMPSs.

“invoke” method development

The invoke method is necessary to bind the parameters defined through command line into the invocation of building_block_task function.

Caution: The name invoke **MUST NOT** be changed.

The invoke method receives two parameters:

arguments

Parsed arguments. Received from command line invocation.

config

Dictionary with the yaml configuration provided.

Consider the following invoke example:


```
def invoke(arguments, config):
    input_file = arguments.model
    output_file = arguments.result
    building_block_task(input_file=input_file,
                       output_file=output_file)
```

This example shows how to get the operation field from config, gets the input and output files and invokes the `building_block_task` method specifying the necessary parameters explicitly (`input_file` and `output_file`).

“function_name” method development

A building block can be invoked through command line, and the method used in this case is `invoke`.

However, since the Building Block can be used in PyCOMPSs workflows, the methods can be invoked directly. This means that the workflow can directly invoke `building_block_task` or any other method.

Consequently, this method is **OPTIONAL** but it is recommended in order to ease the Building Block call from a PyCOMPSs workflow application.

Hint: It can also be a normal python function that calls decorated methods.

This will enable to exploit inner parallelism when used with PyCOMPSs.

Step 4: Test the Building Block

In order to test the Building Block it is necessary to install it. To this end, the Building Block already includes a `install.sh` script:

```
$ my_building_block/./install.sh
```

Once installed the `my_building_block` package, it provides the `my_building_block` command, that can be used from the command line. For example:

```
$ my_building_block -h
usage: my_building_block [-h] --model MODEL --result RESULT [-c CONFIG] [-d]
                        [-l {debug,info,warning,error,critical}] [--tmpdir TMPDIR]
                        [--processes PROCESSES] [--gpus GPUS] [--memory MEMORY]
                        [--mount_points MOUNT_POINTS]

my_building_block Building Block short description. Give more details about the Building_
↪Block.

options:
  -h, --help                show this help message and exit
  --model MODEL              (INPUT - str (file)) Input file (model)
  --result RESULT            (OUTPUT - str) Result file
  -c CONFIG, --config CONFIG
                             (CONFIG) Configuration file path
  -d, --debug                Enable Building Block debug mode. Overrides log_level
  -l {debug,info,warning,error,critical}, --log_level {debug,info,warning,error,critical}
                             Set logging level
  --tmpdir TMPDIR            Temp directory to be mounted in the container
  --processes PROCESSES      Number of processes for MPI executions
  --gpus GPUS                Requirements for GPU jobs
```

(continues on next page)

(continued from previous page)

```
--memory MEMORY      Memory requirement
--mount_points MOUNT_POINTS
                        Comma separated alias:folder to be mounted in the container
```

And to test the Building Block, provide the necessary parameters:

```
# Generate a testing files
$ echo "hello world" > hi.txt
$ my_building_block --model hi.txt --result bye.txt
```

The result will be the appearance of the `bye.txt` file, which is a copy of `hi.txt`.

Step 5: Create an Application

The first step to create an application is to generate a template to start with. The template can be created with the following `permedcoe` command:

```
$ permedcoe template application my_application

Creating Application template
-----
To be completed:

- app.py:(6):      TODO: Import the desired building blocks entry points and use invoke_
  ↳ or any other function.
- Snakefile:(0):   TODO: Declare the building blocks to be used as rules.
- Snakefile:(9):   TODO: Change bb to the building block name.
- NextFlow.nf:(7): TODO: Declare the building blocks to be used as process.
- NextFlow.nf:(18): TODO: Change bb to the building block name.
-----
```

The result of this command is a folder named `my_application` containing a three folders: `NextFlow`, `PyCOMPSs` and `Snakemake`. Each subfolder contains a template for that workflow manager with all scripts and base code to start developing your application.

Important: The application name in this example is `my_application`, but should be defined for your application with a specific name.

This tutorial will continue using `my_application``, so take it into account if you define a different name and swap ```my_application` with yours.

Tip: It is possible to create the template for a single workflow manager by specifying it in the template creation:

```
$ permedcoe template application -t <WORKFLOW_MANAGER> my_application
```

Where `<WORKFLOW_MANAGER>` can be: *pycompss*, *nextflow* or *snakemake*.

Step 6: Use the Building Block from the Application

The Building Block is ready to be used from multiple workflow managers, and we already have a template for each workflow manager:

PyCOMPSs

Now its time to add logic to the application template. To this end, you can use your desired text editor or Python IDE. There is only one file that requires attention (**app.py**):

```
$ cd my_application/PyCOMPSs/
# Edit: app.py
```

In this file, it is necessary to develop the following:

Imports [MANDATORY]

It is necessary to import the building block (in the template my_building_block) methods:

```
# Import building block entry points
from my_building_block import invoke
from my_building_block import my_building_block_task
```

Tip: You can import any other method from the Building Block.

main method [MANDATORY]

It is required to implement the logic of the application that uses the building blocks methods.

It can use directly `invoke` to mimic the command line interface. But you can use any other method from the Building Blocks and invoke them directly.

Tip: The my_application already uses my_building_block.

```
def main():
    # Sample application:
    print("Sample python application using my_building_block BB")
    # Get parameters
    input_file = str(sys.argv[1])
    output_file = str(sys.argv[2])
    conf = {} # conf is empty since it is not used by my_building_block
    # Building Block invocation
    building_block_task(input_file=input_file,
                       output_file=output_file)
```

NOTE: That the parameters match the building_block_task function definition.

Snakemake

Now its time to add logic to the SnakeMake application template. To this end, you can use your desired text editor or Python IDE.

There is only one file that requires attention (**Snakefile**):

```
$ cd my_application/SnakeMake/
# Edit: Snakefile
```

In this file, it is necessary to declare the building blocks as rules and implement their dependencies. The next code snippet shows an example using *my_bulding_block*:

```
rule BUILDINGBLOCK:
input:
    dataset="/path/to/dataset",
    config="/path/to/conf.yaml"
output:
    result="/path/to/result"
shell:
    "permedcoe execute building_block my_building_block --model {input.dataset} --result
↪{output.result} --config {input.config}"
```

NextFlow

Now its time to add logic to the NextFlow application template. To this end, you can use your desired text editor or Python IDE.

There is only one file that requires attention (**NextFlow**):

```
$ cd my_application/NextFlow/

# Edit: NextFlow.nf
```

In this file, it is necessary to declare the building blocks as rules and implement their dependencies. The next code snippet shows an example using *my_bulding_block*:

```
params.input="/path/to/dataset"
params.config="/path/to/conf.yaml"

input_ch = Channel.fromPath(params.input)
conf_ch = Channel.fromPath(params.config)

process BUILDINGBLOCK {
    input:
        file dataset from input_ch
        file conf from conf_ch

    output:
        file "output" into res_ch

    """
    permedcoe execute building_block my_building_block --model $dataset --result output -
↪-config $conf
    """
}
```

Step 7: Test the Application

The template application (*my_application*) is ready to be used with multiple workflow managers:

PyCOMPSs

Caution: It is necessary to have installed PyCOMPSs in order to test the application.

Please, check the [PyCOMPSs installation manual](#)

There is a ready to use script (*launch.sh*) in order to execute the application:

```

$ cd my_application/PyCOMPSS
$ ./launch.sh hi.txt bye2.txt

-----
----- STDOUT -----
[ INFO] Inferred PYTHON language
[ INFO] Using default location for project file: /opt/COMPSS/Runtime/configuration/xml/
→projects/default_project.xml
[ INFO] Using default location for resources file: /opt/COMPSS/Runtime/configuration/
→xml/resources/default_resources.xml
[ INFO] Using default execution type: compss

----- Executing app.py -----

Binding debug is activated
[BINDING-COMMONS] - @JNI_On - Creating the JVM
[BINDING-COMMONS] - @create_vm - reading file in JVM_OPTIONS_FILE
[BINDING-COMMONS] - @create_vm - Launching JVM
[BINDING-COMMONS] - @create_vm - JVM Ready
[BINDING-COMMONS] - @JNI_On - Obtaining Runtime classes
Loading LoggerManager
[BINDING-COMMONS] - @JNI_On - Creating runtime object
[(505) API] - Deploying COMPSS Runtime v3.1
[BINDING-COMMONS] - @JNI_On - Calling runtime start
[(506) API] - Starting COMPSS Runtime v3.1
[(506) API] - Initializing components
[(880) API] - Ready to process tasks
[BINDING-COMMONS] - @Init JNI Types
[BINDING-COMMONS] - @Init JNI Types DONE
[BINDING-COMMONS] - @Init JNI Master
[BINDING-COMMONS] - @Init JNI Methods
[BINDING-COMMONS] - @Init JNI Methods DONE
[BINDING-COMMONS] - @Init JNI OnFailure Types
[BINDING-COMMONS] - @Init JNI Direction Types
[BINDING-COMMONS] - @Init JNI Direction Types DONE
[BINDING-COMMONS] - @Init JNI Stream Types
[BINDING-COMMONS] - @Init JNI Stream Types
[BINDING-COMMONS] - @Init JNI Parameter Prefix
[BINDING-COMMONS] - @Init JNI Parameter Prefix DONE
[BINDING-COMMONS] - @Init Master DONE
[BINDING-COMMONS] - @JNI_Get_AppDir - Getting application directory.
[BINDING-COMMONS] - @JNI_Get_AppDir - directory name: /home/user/.COMPSS/app.py_01/
Sample python application using my_building_block BB
[(888) API] - Registering CoreElement my_building_block.main.building_block_task
[(889) API] - - Implementation: my_building_block.main.building_block_task
[(889) API] - - Constraints :
[(889) API] - - Type : CONTAINER
[(889) API] - - I/O : False
[(889) API] - - ImplTypeArgs :
[(889) API] - Arg: SINGULARITY
[(889) API] - Arg: /home/user/permedcoe/my_building_block/container/
→sample.sif
[(889) API] - Arg: CET_BINARY

```

(continues on next page)

(continued from previous page)

```

[(889)    API] - Arg: cp
[(889)    API] - Arg: [unassigned]
[(889)    API] - Arg: [unassigned]
[(889)    API] - Arg: false
[BINDING-COMMONS] - @JNI_RegisterCE - Task registered: my_building_block.main.building_
↳block_task
[BINDING-COMMONS] - @JNI_ExecuteTaskNew - Processing task execution in bindings-common.
[BINDING-COMMONS] - @JNI_ExecuteTaskNew - Processing parameter 0
[BINDING-COMMONS] - @process_param
[BINDING-COMMONS] - @process_param - ENUM DATA_TYPE: 9
[BINDING-COMMONS] - @process_param - File: hi.txt
[BINDING-COMMONS] - @process_param - ENUM DIRECTION: 0
[BINDING-COMMONS] - @process_param - ENUM STD IO STREAM: 3
[BINDING-COMMONS] - @process_param - PREFIX: null
[BINDING-COMMONS] - @process_param - NAME: #kwarg_input_file
[BINDING-COMMONS] - @process_param - CONTENT TYPE: FILE
[BINDING-COMMONS] - @process_param - WEIGHT : 1.0
[BINDING-COMMONS] - @process_param - KEEP RENAME : 0
[BINDING-COMMONS] - @JNI_ExecuteTaskNew - Processing parameter 1
[BINDING-COMMONS] - @process_param
[BINDING-COMMONS] - @process_param - ENUM DATA_TYPE: 9
[BINDING-COMMONS] - @process_param - File: bye2.txt
[BINDING-COMMONS] - @process_param - ENUM DIRECTION: 1
[BINDING-COMMONS] - @process_param - ENUM STD IO STREAM: 3
[BINDING-COMMONS] - @process_param - PREFIX: null
[BINDING-COMMONS] - @process_param - NAME: #kwarg_output_file
[BINDING-COMMONS] - @process_param - CONTENT TYPE: FILE
[BINDING-COMMONS] - @process_param - WEIGHT : 1.0
[BINDING-COMMONS] - @process_param - KEEP RENAME : 0
[BINDING-COMMONS] - @JNI_ExecuteTaskNew - Processing parameter 2
[BINDING-COMMONS] - @process_param
[BINDING-COMMONS] - @process_param - ENUM DATA_TYPE: 8
[BINDING-COMMONS] - @process_param - String: -v
[BINDING-COMMONS] - @process_param - ENUM DIRECTION: 0
[BINDING-COMMONS] - @process_param - ENUM STD IO STREAM: 3
[BINDING-COMMONS] - @process_param - PREFIX: null
[BINDING-COMMONS] - @process_param - NAME: #kwarg_verbose
[BINDING-COMMONS] - @process_param - CONTENT TYPE: #UNDEFINED#:#UNDEFINED#
[BINDING-COMMONS] - @process_param - WEIGHT : 1.0
[BINDING-COMMONS] - @process_param - KEEP RENAME : 1
[(902)    API] - Creating task from method my_building_block.main.building_block_task,
↳for application 0
[(902)    API] - There are 3 parameters
[(905)    API] - Parameter 0 has type FILE_T
[(909)    API] - Parameter 1 has type FILE_T
[(909)    API] - Parameter 2 has type STRING_T
[BINDING-COMMONS] - @JNI_ExecuteTaskNew - Task processed.
[BINDING-COMMONS] - @JNI_Off
[BINDING-COMMONS] - @Off - Waiting to end tasks
[(912)    API] - No more tasks for app 0
[(4229)   API] - Getting Result Files for app0
[BINDING-COMMONS] - @Off - Stopping runtime

```

(continues on next page)

(continued from previous page)

```

[(4265)  API] - Stopping Wall Clock limit Timer
[(4265)  API] - Stop IT reached
[(4265)  API] - Stopping Graph generation...
[(4265)  API] - Stopping AP...
[(4266)  API] - Stopping TD...
[(4418)  API] - Stopping Comm...
[(4419)  API] - Runtime stopped
[(4419)  API] - Execution Finished
[BINDING-COMMONS] - @Off - Revoke thread access to JVM
[BINDING-COMMONS] - @Off - Removing JVM
[BINDING-COMMONS] - @Off - Removing environment
[BINDING-COMMONS] - @Off - End

-----
----- STDERR -----
WARNING: COMPSs Properties file is null. Setting default values
2021-11-16 15:43:46,197 - DEBUG - Executing container_f wrapper.
2021-11-16 15:43:46,198 - DEBUG - Configuring @container core element.
2021-11-16 15:43:46,198 - DEBUG - Executing binary_f wrapper.
2021-11-16 15:43:46,198 - DEBUG - Configuring @binary core element.
2021-11-16 15:43:46,198 - DEBUG - [@TASK] Task type of function building_block_task in_
↳module my_building_block.main: CONTAINER
2021-11-16 15:43:46,199 - DEBUG - Configuring core element.
2021-11-16 15:43:46,199 - DEBUG - [@TASK] Registering the function building_block_task_
↳in module my_building_block.main
2021-11-16 15:43:46,199 - DEBUG - Registering CE with signature: my_building_block.main.
↳building_block_task
2021-11-16 15:43:46,199 - DEBUG -      - Implementation signature: my_building_block.
↳main.building_block_task
2021-11-16 15:43:46,199 - DEBUG -      - Implementation constraints:
2021-11-16 15:43:46,199 - DEBUG -      - Implementation type: CONTAINER
2021-11-16 15:43:46,199 - DEBUG -      - Implementation type arguments: SINGULARITY /
↳home/user/permedcoe/my_building_block/container/sample.sif CET_BINARY cp [unassigned]_
↳[unassigned] false
2021-11-16 15:43:46,204 - DEBUG - CE with signature my_building_block.main.building_
↳block_task registered.
2021-11-16 15:43:46,209 - DEBUG - Final type for parameter #kwarg_input_file: 9
2021-11-16 15:43:46,210 - DEBUG - Final type for parameter #kwarg_output_file: 9
2021-11-16 15:43:46,210 - DEBUG - Final type for parameter #kwarg_verbose: 8
2021-11-16 15:43:46,210 - DEBUG - TASK: building_block_task of type 1, in module my_
↳building_block.main, in class
2021-11-16 15:43:46,210 - DEBUG - Processing task:
2021-11-16 15:43:46,210 - DEBUG -      - App id: 0
2021-11-16 15:43:46,210 - DEBUG -      - Signature: my_building_block.main.building_
↳block_task
2021-11-16 15:43:46,210 - DEBUG -      - Has target: False
2021-11-16 15:43:46,210 - DEBUG -      - Names: #kwarg_input_file #kwarg_output_file
↳#kwarg_verbose
2021-11-16 15:43:46,210 - DEBUG -      - Values: hi.txt bye2.txt -v
2021-11-16 15:43:46,210 - DEBUG -      - COMPSs types: 9 9 8
2021-11-16 15:43:46,210 - DEBUG -      - COMPSs directions: 0 1 0

```

(continues on next page)

(continued from previous page)

```

2021-11-16 15:43:46,210 - DEBUG - - COMPSs streams: 3 3 3
2021-11-16 15:43:46,211 - DEBUG - - COMPSs prefixes: null null null
2021-11-16 15:43:46,211 - DEBUG - - Content Types: FILE FILE #UNDEFINED#:#UNDEFINED
↪#
2021-11-16 15:43:46,211 - DEBUG - - Weights: 1.0 1.0 1.0
2021-11-16 15:43:46,211 - DEBUG - - Keep_renames: False False True
2021-11-16 15:43:46,211 - DEBUG - - Priority: False
2021-11-16 15:43:46,211 - DEBUG - - Num nodes: 1
2021-11-16 15:43:46,211 - DEBUG - - Reduce: False
2021-11-16 15:43:46,211 - DEBUG - - Chunk Size: 0
2021-11-16 15:43:46,211 - DEBUG - - Replicated: False
2021-11-16 15:43:46,211 - DEBUG - - Distributed: False
2021-11-16 15:43:46,211 - DEBUG - - On failure behavior: RETRY
2021-11-16 15:43:46,212 - DEBUG - - Task time out: 0
2021-11-16 15:43:46,223 - DEBUG - --- END ---
2021-11-16 15:43:46,223 - INFO - Stopping runtime...
2021-11-16 15:43:46,223 - INFO - Cleaning objects...
2021-11-16 15:43:46,223 - INFO - Stopping COMPSs...
2021-11-16 15:43:50,330 - INFO - Cleaning temps...
2021-11-16 15:43:50,345 - INFO - COMPSs stopped
-----

```

Caution: If your application requires parameters, the `launch.sh` script needs to be tuned accordingly.

Tip: The output is very verbose since PyCOMPSs has been executed in debug mode. It can be executed silently by removing the `-d` flag from the `launch.sh` script.

Tip: If any error occurs, it is necessary to debug the execution. To this end it is helpful to execute in debug mode (as it is currently) and check the [Troubleshooting for Python](#) section from the PyCOMPSs documentation.

Tip: It is possible to run the application without PyCOMPSs installed using the `launch_without_pycompss.sh` script. However, the execution of the application will be performed entirely sequentially.

Snakemake

Caution: It is necessary to have installed Snakemake in order to test the application.

There is a ready to use script (`launch.sh`) in order to execute the application:

```

$ cd my_application/SnakeMake
$ ./launch.sh

```

NextFlow

Caution: It is necessary to have installed NextFlow in order to test the application.

There is a ready to use script (`launch.sh`) in order to execute the application:

```
$ cd my_application/NextFlow
$ ./launch.sh
```

Step 8: Run your Application in a Supercomputer

The final steps are: to deploy the Building Blocks and Application into a supercomputer and run the Application using a workflow manager.

To this end, contact the PerMedCoE partners: infoPerMedCoE@bsc.es

ACKNOWLEDGEMENTS

5.1 Partners



**Barcelona
Supercomputing
Center**

Centro Nacional de Supercomputación





HEIDELBERG
UNIVERSITY
HOSPITAL

Atos



cnag

centre nacional d'anàlisi genòmica
centro nacional de análisis genómico



MDC

MAX DELBRÜCK CENTER
FOR MOLECULAR MEDICINE
IN THE HELMHOLTZ ASSOCIATION

Univerza v Ljubljani



5.2 Funding

The PerMedCoE project has received funding from the European Union's Horizon 2020 research and innovation programme under the grant agreement N°951773

