



Project Title	Artificial Intelligence in Secure Privacy-preserving computing coNTinuum
Project Acronym	AI-SPRINT
Project Number	101016577
Type of project	RIA - Research and Innovation action
Topics	ICT-40-2020 - Cloud Computing: towards a smart cloud computing continuum (RIA)
Starting date of Project	01 January 2021
Duration of the project	36 months
Website	www.ai-sprint-project.eu/

AI-SPRINT Studio User Guide

Free version



AI-SPRINT defines a novel framework for the design and operation of AI applications in computing continua.

AI-SPRINT goes beyond supporting AI applications development by enabling the seamless design and partition of AI applications among the plethora of cloud-based solutions and AI-based sensor devices, providing security and privacy guarantees.

Contact

Table of Contents

1. About AI-SPRINT	5
What problems can AI-SPRINT solve?	6
2. AI-SPRINT Framework	6
3. AI-SPRINT Application Architecture	10
3.1 Application DAG and Components	10
3.2 Candidate Resources and Deployments	11
4. AI-SPRINT Design Abstractions	13
Component Name Annotation	14
Execution Time Annotation	14
5. PyCOMPSs and dislib	16
6. OSCAR-P	18
7. SPACE4AI-D	21
8. SCONE	22
9. Getting Started	24
9.1 AI-SPRINT Studio Installation	24
Use the available Docker Image	24
Step 1: Pull the AI-SPRINT Studio Docker Image	25
Step 2a: Verification (AI-SPRINT Design)	25
expected output	25
Step 2b: Verification (TOSCARIZER)	25
expected output	25
10. Code Examples	25
10.1 Prerequisites	25
10.2 Build a New AI-SPRINT Application	26
Create a new application with the AI-SPRINT	26
Step 1 Run aisprint new-application	26
Step 2: Verification	27
expected output	27
10.3 Execute AI-SPRINT Studio	27
Step 1: Download the Mask Detection application (mask_detection_v1)	32
Expected output	33
Step 2: Run aisprint design	37
Step 3: Run TOSCARIZER	38
Step 4: Deploy Base Deployment	39
Expected output	40
Step 5: Run OSCAR-P	41
Step 6: Run SPACE4AI-D	49
11. AI-SPRINT Runtime environment	58

11.1 AI-SPRINT Monitoring Subsystem (AMS)	58
11.1.1 Infrastructure monitoring	59
11.1.2 QoS monitoring	60
11.1.3 Custom metrics	61
11.5 SPACE4AI-R	62
Installation	63
Output	63

1. About AI-SPRINT

The aim of the AI-SPRINT “Artificial intelligence in Secure PRivacy-preserving computing coNTinuum” project is to develop a framework composed of design and runtime management tools to seamlessly design, partition and operate Artificial Intelligence (AI) applications among the current plethora of cloud-based solutions and AI-based sensor devices (i.e., devices with intelligence and data processing capabilities), providing resource efficiency, performance, data privacy, and security guarantees.

What problems can AI-SPRINT solve?

Table 1.1 Summarizes the AI-SPRINT framework end-users and problems solved.









Without AI-SPRINT	With AI-SPRINT
 <p>Application Developers</p> <p>Need to design manually each component of the AI application pipeline</p> <p>Need to manage the application parallelization</p>	 <p>Application Developers</p> <p>High-level QoS annotations and automatically partition DNNs</p> <p>Transparently implement the parallelization of compute-intensive applications</p>
 <p>Application Manager</p> <p>Need to manually configure the environment</p> <p>Qualitative appraisal of the performance</p>	 <p>Application Manager</p> <p>Automatically containerize applications</p> <p>TOSCA templates for complex & distributed applications</p> <p>Automate AI application performance profiling & design space exploration</p>
 <p>Infrastructure Provider & Sysops</p> <p>Naive autoscaling solutions</p> <p>Basic DL scheduling mechanisms</p> <p>Not optimized energy nor cloud operational costs</p>	 <p>Infrastructure Provider & Sysops</p> <p>Advanced resource allocation to cope with load variations</p> <p>Optimized energy or cloud operational costs and energy-aware runtime migration</p>
 <p>End User</p> <p>Need to trust the cloud and storage provider</p> <p>Need to manually create confidential docker images and recompile binaries</p> <p>Must manually encrypt files and seal libraries</p>	 <p>End User</p> <p>Trustable computing and storage environments even on untrusted providers</p> <p>Quantitative anonymization level</p>

Table 1.1 - AI-SPRINT end users and problems solved

2. AI-SPRINT Framework

AI-SPRINT overcomes current technological challenges for the design and efficient execution of AI applications exploiting resources in the edge-to-cloud continuum such as flexibility, scalability,

interoperability, security, and privacy. An AI-SPRINT application is mostly written in Python, and it makes intensive use of AI technologies. An application usually comprises multiple components which run across a computing continuum with some components allocated on the cloud, some on edge servers some on AI-enabled sensors (i.e., devices with intelligence and data processing capabilities).

The objective of the AI-SPRINT Design Tools packaged as the **AI-SPRINT Studio** is to provide a layer that abstracts the applications from the underlying computing resources, being these edge resources or cloud servers deployed and managed by OSCAR¹, so that the application developer only needs to focus on the actual algorithm and application logic. On the other hand, interfaces for easing the integration of developed applications with the runtime system are also available. The AI-SPRINT developer is provided with a framework to design AI applications using abstractions to specify quality parameters and to express the resource requirements of the components of AI distributed applications. Performance models are used to predict execution times for the different parts of AI applications in a given deployment and to drive the mapping of components on processing elements. Security policies are available in the definition of AI applications to run on different deployments as edge resources or cloud premises.

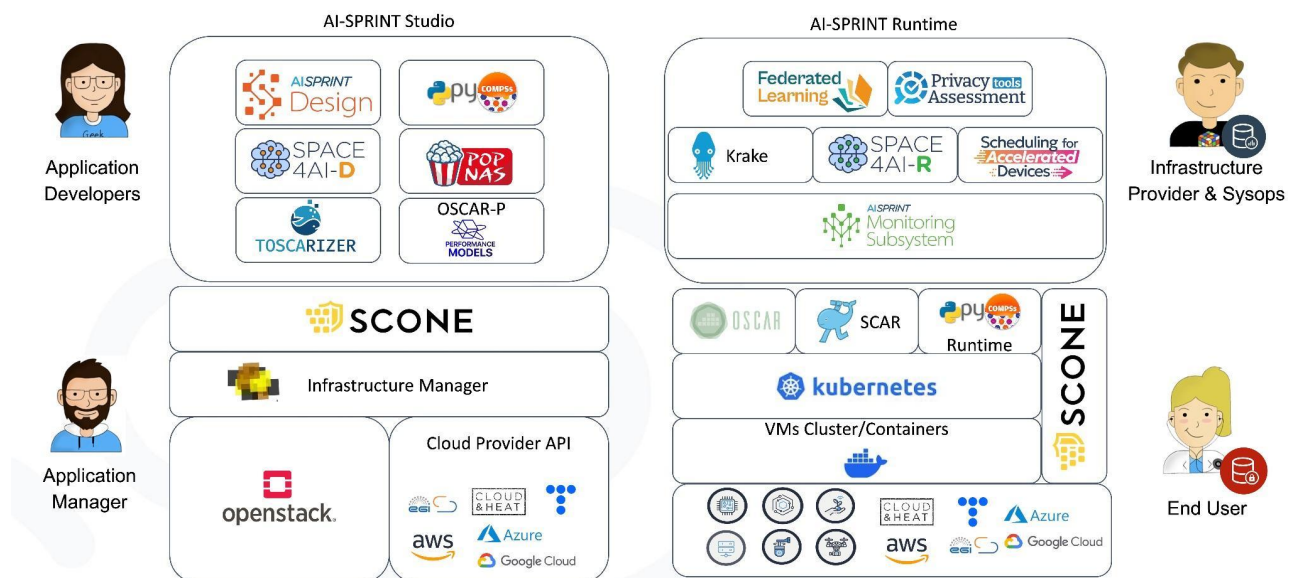


Figure 2.1 - AI-SPRINT Framework

The design environment, shown in Figure 2.1, includes the following main components:

- Design and Programming Abstractions,** to hide the communications across components and to transparently implement the parallelization of the compute-intensive part of the application, possibly exploiting specialized resources (e.g., GPUs and AI enabled sensors). The code developed in AI-SPRINT is enriched with high-level **annotations for QoS constraints** and code dependencies, with performance parameters for the allocation of tasks to computing continuum resources and with security annotations for data allocation and processing. **AI-SPRINT Design** parses the application code looking for the decorated functions, and generates a list of annotations together with the corresponding parameters. Additionally, annotations to guide the partitioning of specific components based on Deep Neural Networks, their early exit, and the support to multiple versions of the same function with different accuracy are also provided. **PyCOMPS** hides the communications across components and transparently implements the parallelization of the compute-intensive part of the application. Provides interoperability with several deep learning environments such as PyTorch and the European Distributed Deep Learning library (EDDL). **dislib** (part of PyCOMPS) implements several ML algorithms reducing the execution time of the training

¹ <https://oscar.grycap.net/>

and inference processes by exploiting the inherent data parallelism of the input data. The input dataset is distributed over a cluster, and each node performs the training on the local data. Models trained in isolation are iteratively combined together according to a stochastic gradient descent scheme. The distribution and execution of the operations are managed by the COMPSs runtime.

- The **TOSCARIZER**, which aims to provide TOSCA² documents for the optimal and base component placement to connect with the virtual infrastructure provisioning module. It generates the docker images of all the application components, the TOSCA documents for each component for the target resource, and the **OSCAR** FDL file for the whole inference pipeline. Finally, it triggers the virtual infrastructure provision module (i.e., the Infrastructure Manager³) to create and destroy the virtual infrastructures.
- The Infrastructure Manager - **IM** -, a TOSCA-based orchestrator that provisions customized virtual infrastructures on multi-Clouds. These include widely used on-premises Cloud Management Platforms (CMPs) such as OpenNebula and OpenStack; public Cloud providers such as Amazon Web Services, Microsoft Azure, and Google Cloud Platform; European Cloud infrastructures such as the EGI Federated Cloud and commercial providers such as Open Telekom Cloud and Orange.
- Tools that, once an AI application is designed, allow to automatically identify the *Performance Models* providing the highest performance prediction accuracy, before the production deployment or throughout revision cycles, under different configurations and deployment settings at the full computing continuum. The AI-SPRINT performance modelling approach is mainly based on ML, in particular, on the aMLLibrary⁴ library. For inference pipelines, Performance Models training data is automatically provided by **OSCAR-P**, a tool which automates the profiling of an OSCAR application by testing its full workflow on different hardware and node combinations, obtaining relevant information on the timing of the execution of the individual components.
- *Applications design space exploration* tools that evaluate multiple alternative candidate deployments for complex applications involving many components, and identify the resource selection and components placement maximizing resource efficiency while minimizing the cloud usage cost. **SPACE4AI-D** (*System PerformAnce and Cost Evaluation on Cloud for AI applications Design*) tackles this problem by leveraging a random greedy algorithm coupled with several heuristics, i.e., local search, tabu search, simulated annealing, and genetic algorithms.
- **POPNAS** which provides a machine-learning-as-a-service solution that automatically identifies the most accurate deep neural network from a training set of labelled training examples (images or time series) by trading off multiple metrics (training execution time, inference execution time, model performance).

The main artifacts that are produced by the design time and that are also used by the runtime tools are:

- The TOSCA description of the optimal application deployment identified by the SPACE4AI-D tool, which is used as input by the Infrastructure Manager to instantiate the required resources;
- The application components/partitions images that are built by TOSCARIZER;
- The application performance models that are used to predict the application performance on unseen configurations and to trigger adaptation/scheduling decisions at runtime.

The AI-SPRINT runtime is automatically deployed with the application transparently for the end-user. The application execution is continuously monitored at the full stack and, in case of performance violations and periodically, resource allocation is re-optimized by SPACE4AI-R.

² TOSCA is an OASIS standard language to describe the topology of cloud infrastructure.

³ <https://www.grycap.upv.es/im/index.php>

⁴ <https://github.com/a-MLLibrary/aMLLibrary>

In particular the runtime environment includes:

- **OSCAR**, a platform to support the serverless computing model for event-driven data-processing applications along the computing continuum. The use of a minified Kubernetes distribution running on ARM-based processors, such as those provided by Raspberry Pis, allows executing OSCAR clusters for the inference in the edge of previously-trained deep learning models.
- **SCAR**, a framework to create Docker-based serverless applications in AWS Lambda with automated delegation into AWS Batch. This approach combines the benefits of high elasticity provided by AWS Lambda with the unbounded computing capacity provided by AWS Batch, allowing the creation of data-driven serverless workflows typically aimed at file processing.
- The AI-SPRINT **Monitoring Subsystem (AMS)** which provides seamless collection, storage, forwarding and analysis of time series data and logs in complex hierarchical multi-layer deployments enforcing framework constraints and providing violation notifications. AMS enforces AI-SPRINT framework constraints and, based on the results of the application data flow analysis, in case of a constraint violation, sends a notification to the SPACE4AI-R REST API endpoint.
- **SPACE4AI-R** which determines a new optimal deployment for the production application, possibly changing the current assignment of application components to resources and their configuration according to the current load. In case of performance constraint violations or periodically, evaluates the resources allocated and updates the configuration of the running application accordingly.
- **Krake**, an orchestrator engine for containerised and virtualized workloads across distributed and heterogeneous cloud platforms using metrics and labels. It also provides the ability to create Kubernetes clusters with third-party tools.
- The **Privacy Preservation Component**, a tool for training deep learning models that trade-off accuracy and resilience to privacy attacks. It presents a distinct opportunity to ensure stringent model privacy by employing various techniques, including differential privacy, regularisation, and adversarial training
- **Drift Detector**, able to detect an AI model drift at runtime by identifying the drift of a user-defined metric. During runtime, the accuracy of a production AI model is continuously evaluated by monitoring a user-defined metric. If the metric undergoes drift during an observation period, as detected by the user-provided detection algorithm, a drift alarm is raised. After the drift has been detected, the AI-SPRINT user is notified, and additional data is gathered automatically and can be used to retrain the model to adapt to the new data distribution.
- *Federated learning solutions*, AI-SPRINT facilitates the joint training of ML models across various entities within the computing continuum, without necessitating the direct exchange of data. Within this framework, traditional federated average is implemented by PyCOMPSs and, moreover, Secure Generative Data Exchange - **SGDE** - has been developed to establish, amass, and disseminate generators of data that are cognizant of privacy concerns. SGDE tackles the challenge of data accumulation by enabling the training of data generators directly on edge devices, the primary sites of data collection. These generators are trained following stringent privacy-preserving protocols, ensuring that they are incapable of reconstructing any data that could compromise user privacy, even in the presence of a malicious agent. Moreover, SGDE introduces a protocol for sharing these data generators. This allows AI developers seeking data for specific tasks to access a repository of data generators, from which they can create an extensive synthetic dataset. Such datasets can be utilised to train machine learning models, circumventing direct access to sensitive real-world data.
- **Scheduling solutions for accelerated devices**, tools able to select the best VMs flavor, GPU types, and GPU partitions to support the training of Deep Learning jobs, minimising energy or execution costs, while meeting deadline constraints. The Scheduler manages the jobs, selects the nodes for their execution, and assigning the GPUs to them.

Finally, AI-SPRINT applications can be also secured by **SCONE**, a framework that allows to transparently run applications in Trusted Executions Environments (TEE) such as Intel SGX. Besides adding instrumentation to leverage TEEs, SCONE also provides transparent file system encryption as well as secure communications. Applications are attested to verify if the code is indeed executed in an enclave of a TEE and has not been tampered with. In case the attestation succeeds, SCONE provides the applications with configuration as well as reassurance that confidential information and private keys will never get into human hands.

3. AI-SPRINT Application Architecture

AI-SPRINT enables the design and deployment of ML-based applications through a well-defined set of consecutive development stages. On the user side, the main actors are the application programmer and the SysOp, which have to provide the code of the implemented application, as well as the set of configuration files needed to the AI-SPRINT tools to perform the design and deployment. In particular, AI-SPRINT applications must comply with the predefined structure reported in the diagram reported in Figure 3.1.

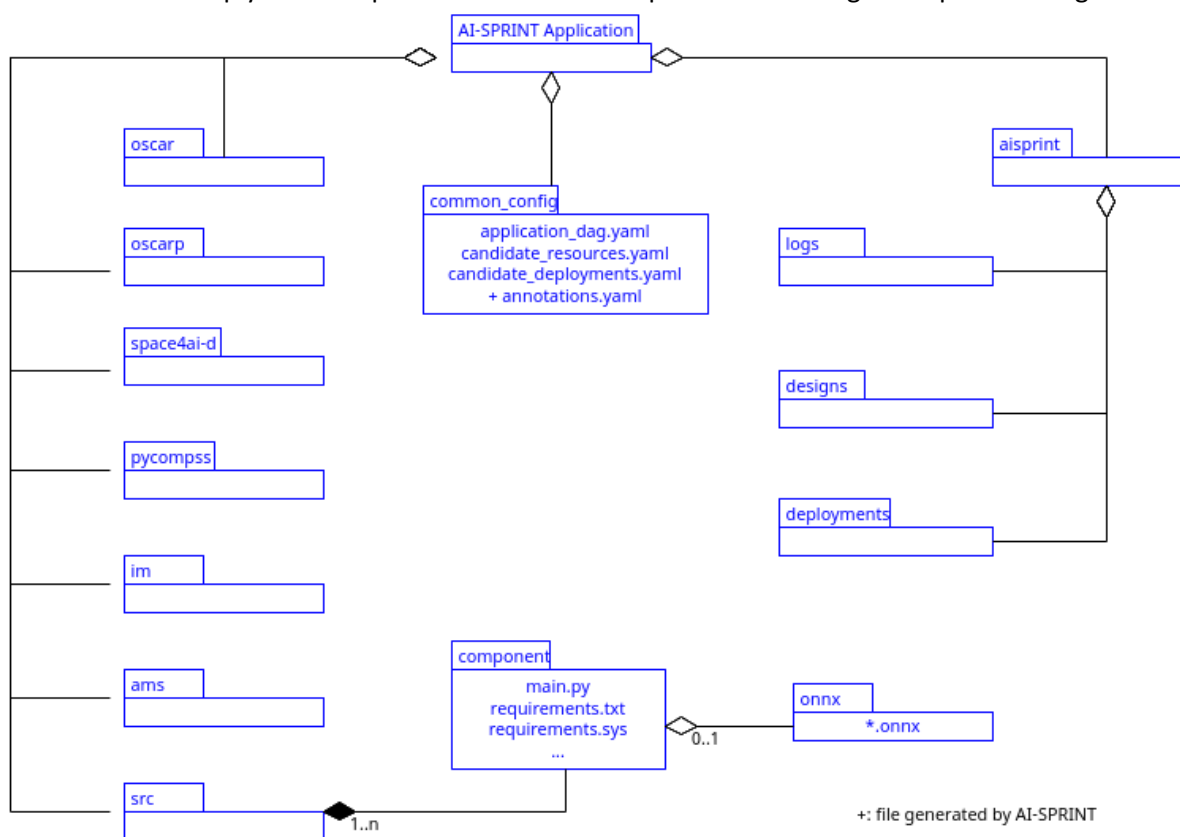


Figure 3.1 - AI-SPRINT Application Structure

3.1 Application DAG and Components

The dependency of components in our system is defined by the application Directed Acyclic Graph (DAG). The application DAG description includes a list of vertices which shows the application components and a list of edges that denotes the dependency between two application components. An edge is shown with a tuple that includes source component, destination component and transition probability between source and destination component. Consider an example application with two consecutive components. The **dag.yaml** file is defined as follows:

DAG YAML file

System:

```
name: Example Application
components: [component1, component2]
dependencies: [[component1, component2, 1]]
```

3.2 Candidate Resources and Deployments

AI-SPRINT allows defining the resources that are available for running the application components. Given the available resources, the user is enabled to explicitly associate each application component with a set of candidate resources. Both the two kinds of information are provided by two separate files: the *candidate_resources.yaml* file and the *candidate_deployments.yaml* file.

Candidate Resources

A system is characterised by a set of Network Domains that alternatively includes at least two computational layers consisting of one or more candidate resources. The *candidate_resources.yaml* file, which describes our system, includes a main section which is named `NetworkDomains` and it defines several network domains with different network communication properties connecting devices with each other and the technology of the underlying connection. All the computational layers located in the same Network Domain can communicate together under the Network Domain's connection properties.

In AI-SPRINT, the components of an application are deployed across different computing layers in a hierarchical way, following the OpenFog Reference Architecture (RA). Each computational layer (an item in `ComputationalLayers`) includes a list of candidate resources (`Resources`) such that one of them can be selected to run the application components.

A computational layer can be either `physical` (and then we distinguish among the ones that are already provisioned or have to be provisioned through IM), `virtual` (i.e., a public cloud VM image) or `NativeCloudFunction` (in case of AWS lambda). This is specified in the `type` field. This concept is used at deployment time by IM and at runtime by SPACE4AI-R such that if a layer is `physical`, the selected device at design time will not be replaced by a different one at runtime. Vice versa, in a `virtual` layer, if all resources in the layer are switched off (the layer is not in use), the SPACE4AI-R can select any of them and switch it on at runtime. On the contrary, if a `virtual` layer is in use, SPACE4AI-R can only scale in or out the number of resources. Usually, the edge and private cloud layers' type are `physical` while public cloud type is `virtual`.

The candidate resources are characterised by some attributes, such as `name`, `description`, a list of processors (`processors`), the total physical devices (in edge side) or the total number of VMs that can run on premises or in the cloud side (`totalNodes`).

Base resources include attributes common to every resource in the computing continuum and are then classified in `Resources` (further detailed in `VirtualMachines`, `PhysicalNodes`, and `EdgeNodes`) and `FaaS`. As an example, a `COMPSsNode` is characterised additionally by the type of processor, the number of cores for each processor, the internal cache and other user defined properties.

The field `operatingSystemImageId` defines the identifier of the Virtual Machine Image that will be used to deploy the resource in a specific cloud provider.

Furthermore, resources are annotated if they support secure boot, i.e., if the BIOS provides and is enabled for secure booting as well as if the operating system image used on the physical or virtual node supports measured boot in order to fulfil the security constraints laid out by the application developer or user.

Resources can have one or more accelerators which have some attributes such as GFLOPS, powerDraw and memorySize and each accelerator has its own processor(s). The attributes' description of resources and processors are brought to the following YAML input file structure.

Candidate Deployments

The *candidate_deployments.yaml* file allows to associate each component of the application with one or more of the resources defined in the *candidate_resources.yaml* file. This is done by defining the Components sections, where a list of Containers related to each component specified, as well as its compatible resources. Each component (partition) has two attributes called candidateExecutionLayers and candidateExecutionResources which denote a list of compatible computational layers and resources, respectively, on which the component can run. Each component can run on the available resources as a container characterised by some attributes like memorySize, computingUnits, to specify the required memory and the number of cores. Moreover, regarding the security part, we provide three layers of protection and devise some binary fields like trustedExecution, networkProtection and fileSystemProtection. trustedExecution ensures that data is stored, processed and protected in a secure environment by memory encryption that relies on Intel SGX or alternative technologies. networkProtection provides network shielding which means that if the application communicates through an unsecure protocol (e.g., http), the connection will be encrypted by enabling this field. Finally, if the storage provider cannot be trusted, by enabling the fileSystemProtection field, the files will be transparently and automatically encrypted.

As discussed in Section 4 - Partitionable Model Annotation, Deep Neural Networks (DNN) based components can be automatically partitioned by the SPACE4AI-D-partitioner tool in two segments, which might vary depending on the layer where the cut is performed. The naming convention is 'COMPONENT_NAME_partitionX_Y' where Y can be 1 or 2 denoting the first or the second half of the underlying DNN model. For what concerns the candidate resource specification the different versions of partitionX_1 and partitionX_2 will be associated with the same resources, while they have their own section defining the candidate execution layers in the *candidate_deployments.yaml* file.

The following Figure 3.2 reports an example of the information that is provided by the two *candidate_resources.yaml* and *candidate_deployments.yaml* files.

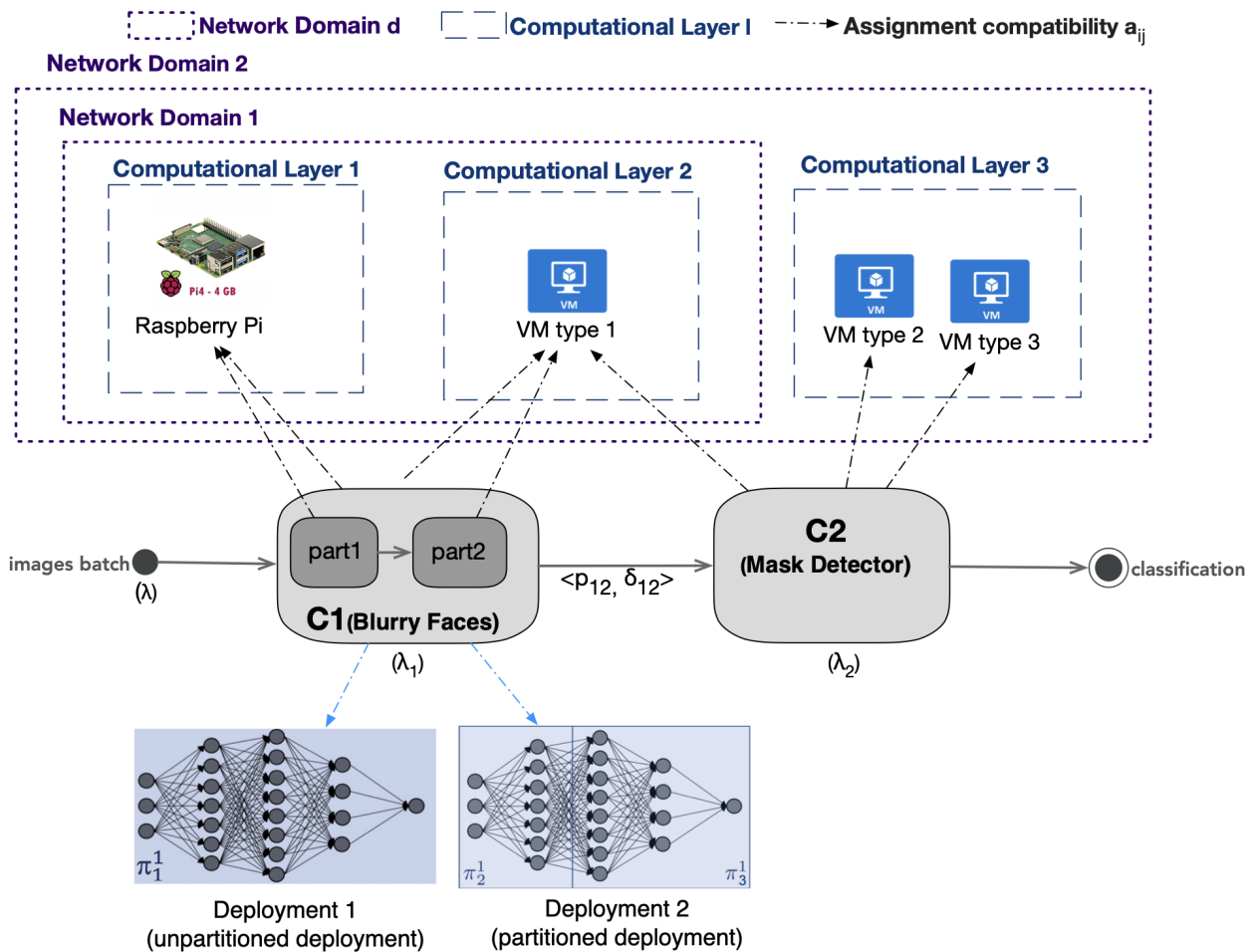


Figure 3.2 - Visual example of the information provided by the candidate resources and deployments files. On the top the candidate resources, as defined by the user in the `candidate_resources.yaml` file, while on the bottom the application components, each one having a candidate computational layer (represented by the arrows), as defined by the user in the `candidate_deployments.yaml` file. In the example the “Blurry Faces” component is partitioned into two parts, each one having its own candidate computational layer.

4. AI-SPRINT Design Abstractions

Quality of Service (QoS) annotations provide the AI-SPRINT application developer with the capability of specifying quality constraints to drive the deployment. Indeed, one of the objectives of the AI-SPRINT design environment is to provide a layer of abstraction between the application that must be deployed and the computing continuum, thus masking the management of the available resources to developers. At the same time, QoS Annotations give enough control to the developer to constrain, during the implementation, the allocation of resources in the cloud continuum. To this end, annotations are realised through Python decorators that enrich the user code and simplify the definition of quality constraints from the developer's perspective, which is only in charge of *annotating* the application components and, depending on the annotation, of providing additional configuration files. Indeed, an AI-SPRINT application is to be intended, in this context, as a set of Python components, for ML inference, whose execution is orchestrated by OSCAR. In AI-SPRINT, each component is defined as a Python application, which can be any complex Python package but must include a Python script with a `main` function. The latter is exactly the function that must be annotated in the case quality constraints are needed. As a convention, in order to automatically detect the main script of the component when running the AI-SPRINT tools, we require the script containing the

definition of the *main* function to be named as *main.py*. In the following, a description of the annotations involved in the tutorial (Section 8) is provided.

Component Name Annotation

The first annotation provided by AI-SPRINT is *component_name*. It has the simple objective of providing an easy way for the users to give a name to the components of an AI-SPRINT application. The name must be unique for the components and allows to identify the specific component throughout the framework. Indeed, all the AI-SPRINT tools identify the components based on their name. As for all the AI-SPRINT annotations, it is realised as a Python decorator, which is used to annotate the *main* function of an AI-SPRINT application component. The implementation is very straightforward since this annotation does not execute any additional code when wrapping the component, but it simply serves as pure annotation, which is parsed by the AI-SPRINT Design tool to associate to the component its user-defined name. In the following, the annotation header is reported, while in Figure 4.1 an example of its usage is provided:

```
def component_name(name)

from aisprint.annotations import component_name

@Component_name(name='example_component')
def main(args):
    ...
```

Figure 4.1 - Example of annotated component. The name of the component will be 'example_component' throughout the whole application and will be used by the AI-SPRINT tools to refer to this specific component.

Execution Time Annotation

The *Execution Time* annotation, whose decorator is named as *exec_time*, allows defining *time constraints*, i.e., upper bounds for the execution time of application components, which, besides guiding the design of the deployment, will be monitored at runtime to detect any violation. The set of candidate resources will be then skimmed at design time, by selecting (through SPACE4AI-D) those that should meet the desired time requirements according to the performance models. In particular, the user can use the annotation to specify a required execution time of a single or multiple components. In the AI-SPRINT perspective, this translates into annotating the component with a Python decorator named *exec_time*, which takes as an argument the maximum execution time required by the user. The decorator provides a wrapper of the annotated component, by allowing the execution of additional Python code each time the component is executed. Specifically, the *exec_time* decorator decollects runtime information and sends this information to the AI-SPRINT *Monitoring Subsystem (AMS)*, which stores the raw data and metadata, out of which execution times are being computed, computes the monitoring metrics based on the defined time constraints, and triggers SPACE4AI-R API functions in case of violation of such constraints.

The annotation header is the following:

```
def exec_time(local_time_thr=None,
              global_time_thr=None,
              prev_components=None) .
```

The *local_time_thr* is the maximum execution time, measured in seconds, of the single component and defines what is called a *local* time constraint. Supposing a component named 'blur_image_component' that

is required to run in no more than 20 seconds. The user is able to define this constraint by annotating the main function of the component as shown in Figure 4.2.

```
@component_name(name='blur_image_component')
@exec_time(local_time_thr=20)
def main(args):
    # assign model path and threshold
    model_path = args.model_path
    threshold = args.threshold
    ...
```

Figure 4.2 - The main function of the component named 'blur_image_component' has been annotated considering a local time constraint of 20 seconds.

The `global_time_thr` is instead the maximum execution time required for the consecutive execution of multiple components, which defines a *global* time constraint. Consider for instance an AI-SPRINT application of two components. The user may require that the two components are executed in at most a certain amount of time. This is shown in the example application in Figure 4.3. The figure highlights the two kinds of time constraints. Local constraints refer to the maximum execution time of the single components, while the global constraints refer to the maximum execution time of a group of consecutive components, that is, an AI/ML inference workflow. For instance, the user could require a maximum of 30 seconds to both anonymise the video and to provide the result of the mask detection algorithm. Furthermore, a distinction is made between the execution time of the pure component function and the execution time measured including the overhead due to the job creation.

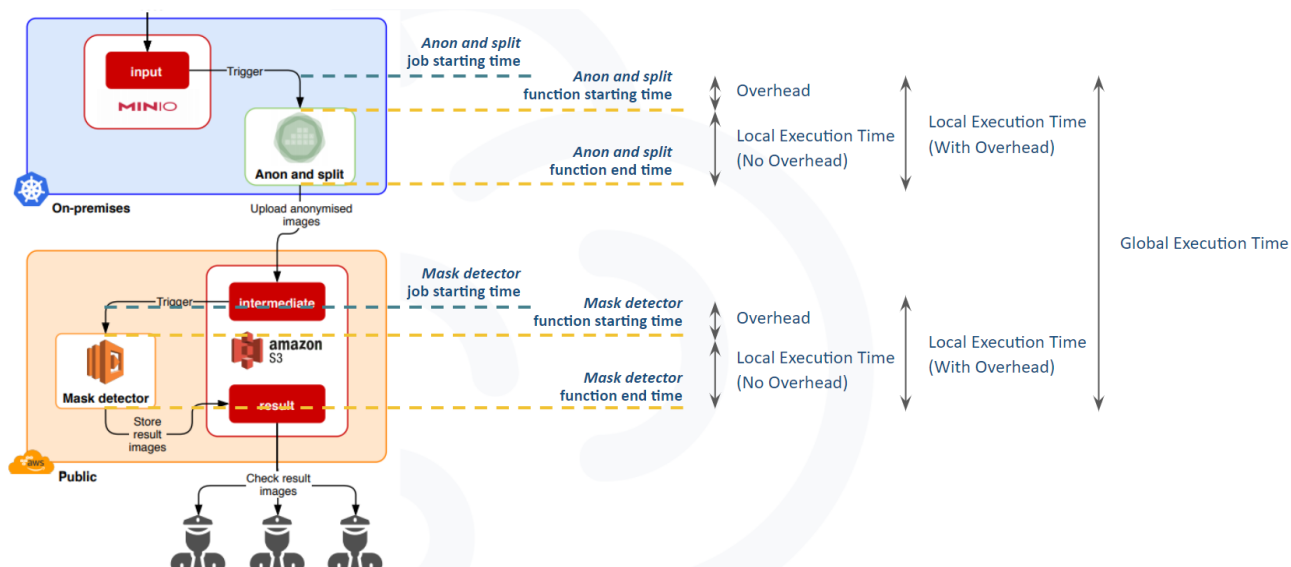


Figure 4.3 - Visual explanation of the difference between job and function starting times, as well as the difference between local and global constraints. The job starting time refers to the job creation time, while the function starting and end times refer to the execution of the pure Python function implementing the component.

In general, local and global constraints can be combined. A single component may be subject to a local constraint and, at the same time, be involved in a global constraint. Suppose an AI-SPRINT inference application with 3 components C1, C2, and C3. Then, it is possible, for instance, to define at the same time: a global constraint involving components C1, C2 and C3; a global constraint involving only components C1 and C2; three local constraints, one for each component (see Figure 4.4).

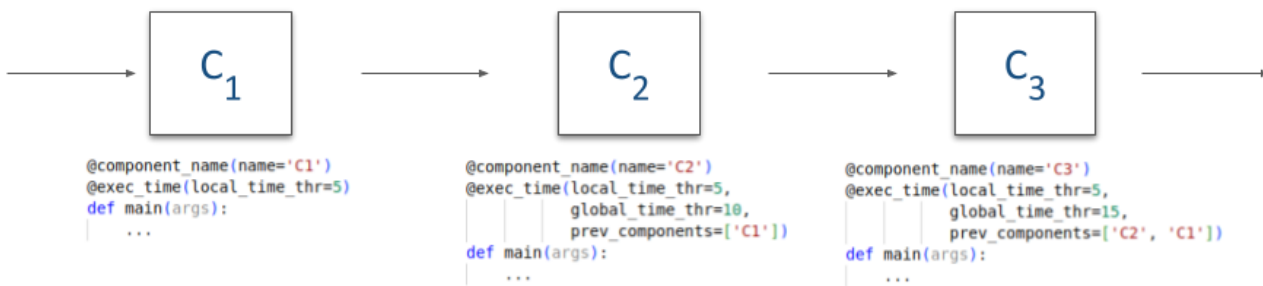


Figure 4.4 - Example of an AI-SPRINT inference application with 3 components C_1 , C_2 , and C_3 subject to two global constraints and three local constraints. The global constraints define a maximum execution time for both the $C_1+C_2+C_3$ (15s) and the C_1+C_2 (10s) computation paths. At the same time, the local constraints define a maximum execution time of 5s for each individual component.

5. PyCOMPSs and dislib

PyCOMPSs provides automatic parallelization of the application leveraging the user provided information through input dependency annotations in the code (see Figure 5.1). The application developer provides a sequential Python script whose functions are annotated through decorators; these annotations are used by the runtime to run those parts of code as asynchronous parallel tasks code.

These annotations describe the type of parameters and constraints on the resources. PyCOMPSs also provides a set of APIs to control the flow of the applications (fault tolerance and synchronisation points). PyCOMPSs processes the information provided by the user through Python decorators and generates a dependency graph.

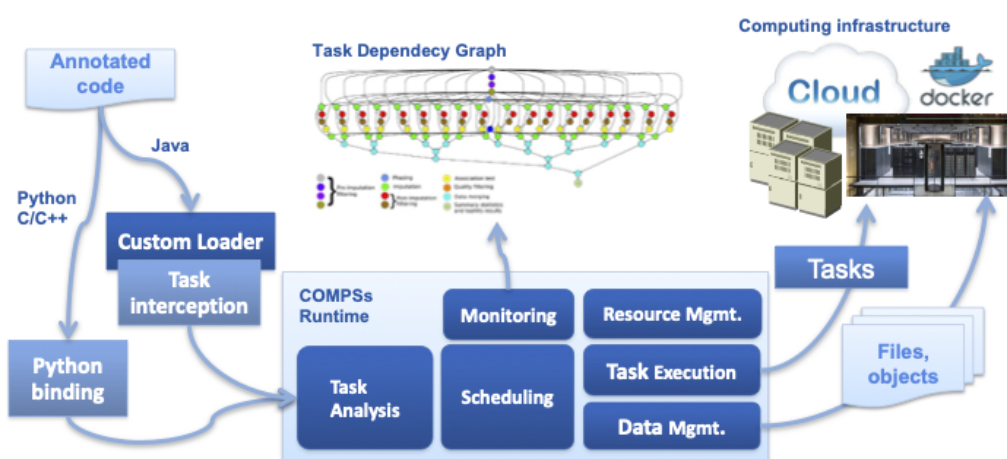


Figure 5.1 - Design and execution of a COMPSs application

The Distributed Computing Library (dislib) is a Python library built on top of PyCOMPSs that provides distributed mathematical and machine learning algorithms through an easy-to-use interface. dislib abstracts Python developers from all the parallelisation details and allows them to build large-scale machine learning

workflows in a completely sequential and effortless manner. dislib is a subcomponent of the design tools layer. dislib is a collection of PyCOMPSs applications that exposes two main interfaces to developers: 1) a distributed data structure called distributed array (ds-array), and 2) an estimator-based API. A ds-array is a 2-dimensional matrix divided in blocks that are stored across different computers. Ds-arrays offer a similar API to NumPy [Numpy2011], which is one of the most popular numerical libraries for Python. The difference between NumPy arrays and ds-arrays is that ds-arrays are internally parallelised and use distributed memory. This means that ds-arrays can store and process much larger data than NumPy arrays.

The typical dislib application consists of the following steps:

- 1) Load data into a ds-array
- 2) Instantiate an estimator object with parameters
- 3) Fit the estimator with the loaded data
- 4) Retrieve information from the estimator or make predictions on new data

As an example we consider the inference part of a dislib application; we have one component that in the figure is called classifier that is executed a COMPSs agent service through OSCAR.

```
from aisprint.annotations import annotation
import subprocess

@annotation({'component_name': {'name': 'classifier'},
            'exec_time': {'local_time_thr': 20},
            'model_performance': {'metric': 'average_f1', 'metric_thr': 0.5}})
def main(input=None, output=None):
    agent = start_agent()

    subprocess.Popen(["/opt/COMPSs/Runtime/scripts/user/compss_agent_call_operation",
                    "--master_node=127.0.0.1",
                    "--master_port=46101",
                    "--lang=python",
                    "--method_name=predict",
                    "--stop",
                    "rf_predict",
                    str(input),
                    str(output)])
```

The previous code has to be provided src folder of the implementation together with the specific implementation of the inference application and the trained model file.

```
from pycomps.api.task import task

import dislib as ds
from dislib.classification import RandomForestClassifier
from dislib.decomposition import PCA
@task()
```

```
def predict(in_file, out_file):
    start_time = time.time()
    print("Model was not loaded", time.time() - start_time)
    pca = PCA()
    pca.load_model("/opt/inference/pca_model", load_format="pickle")

    RF = RandomForestClassifier()
    RF.load_model("/opt/inference/rf_model", load_format="pickle")
    print("Model Loading Time", time.time() - start_time)

    x_test = load_n_preprocess(in_file)
    x_test = ds.array(x_test, x_test.shape)
    print("ECG Loading Time", time.time() - start_time)

    print("Running PCA on ECG " + in_file, flush=True)
```

6. OSCAR-P

OSCAR-P is built around OSCAR and its components, and it acts as a director, configuring and coordinating the profiling activities and collecting the required data once the profiling is completed. The aim of OSCAR-P is to simplify and fully automate the testing of specific OSCAR application workflows on different hardware configurations, and collect data to train machine learning performance models.

Specifically, OSCAR-P receives as input:

- a description of resources that needs to be tested, with a detailed overview of their hardware and software architecture (the number of available nodes, the memory amount and number of cores of every node);
- the components, their Docker images, and their hardware requirements (the needed memory amount and number of cores of every single instance);
- a set of parameters specifying how to test the application (which input files to use, the number of batches and their sizes, the time interval between uploads and their distribution). Moreover, for every component, the parallelism levels to be tested (i.e., the maximum number of parallel instances that a component is allowed to have) is also specified;
- the machine learning models to consider in the performance models training and their hyperparameters.

The tests are performed by varying the used resources, either by changing the number of active nodes or by changing one resource with another.

Once all the required resources are in place and correctly configured, the testing campaign is controlled by a single YAML configuration file containing the list of services, the description of the clusters and their worker nodes and information on how the application needs to be tested; this YAML file is paired with the results of its associated run, to simplify debugging and allowing replicability. Every combination of hardware and nodes is tested more than once to cope with execution time measurement noise and, after testing the full workflow, the individual components are also tested on their own.

The profiling activities have to follow a precise sequence of steps (illustrated in Figure 6.1), each managed by a separate OSCAR-P sub-component.

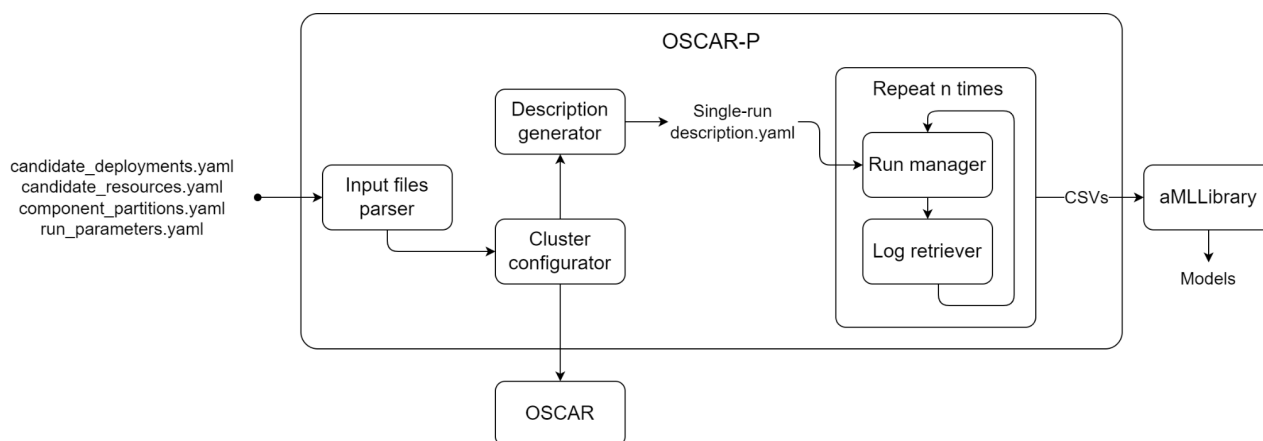


Figure 6.1 - Profiling activity steps and OSCAR-P sub-components.

Starting from the input files, OSCAR-P lists all the “testing units”, i.e., all the valid component / resource tuples; if a component is partitioned (OSCAR-P supports also the execution of partitioned Deep Neural Networks), all its partitions are considered as part of the same “testing unit”, thus ensuring that they are always tested together.

As a simple example (see Figure 6.2), we can consider an application including a single component (*Component 1*), which can be split in two partitions (*Component 1.1* and *Component 1.2*) available for different architectures (ARM64 and AMD64, respectively); *Component 1* is instead available for both architectures. The available resources include a cluster of Raspberry Pi (ARM64) and a cluster of Virtual Machines (AMD64).

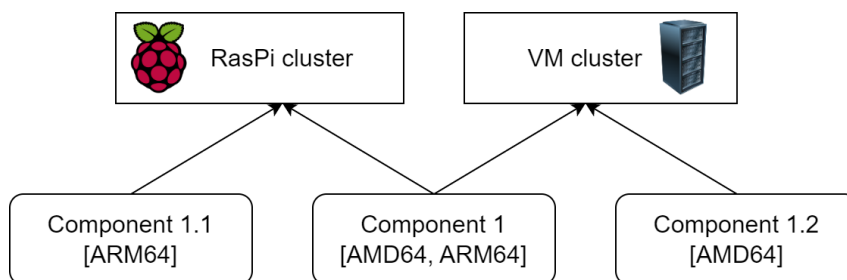


Figure 6.2 - Simple application example.

In this scenario, the testing units would be:

1. Component 1 on the RasPi cluster;
2. Component 1 on the VM cluster;
3. Component 1.1 on RasPi and Component 1.2 on VM cluster.

OSCAR-P then creates a list of all the possible deployments, that is all the possible combinations of the testing units that amount to the full applications (see Figure 6.3).

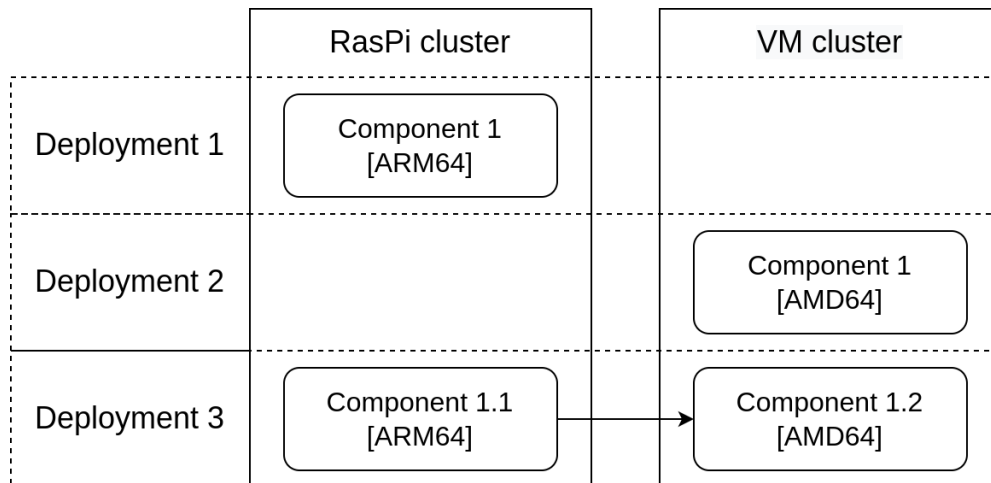


Figure 6.3 - Deployments example.

Finally, every deployment contains a list of “runs” to be tested: each run includes the same list of components, but the “parallelism” of each component (i.e., its maximum number of parallel instances) varies from one run to the other according to what has been specified into the `run_parameters.yaml` file. An example is shown for Deployment 3 in Figure 6.4.

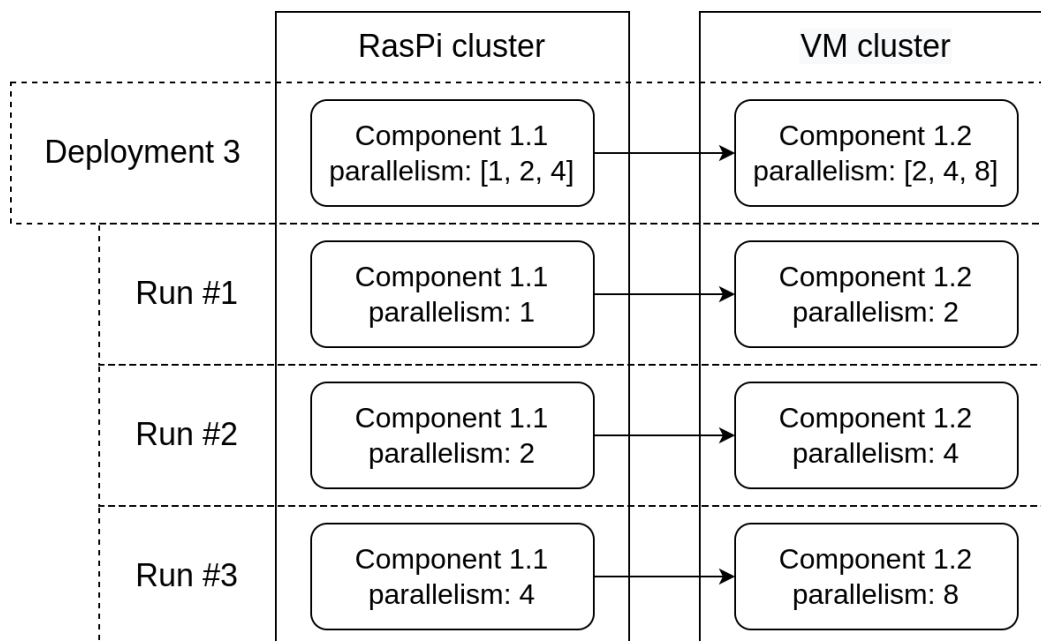


Figure 6.4 - Deployment 3 translated into runs, the amount of runs and their configuration is controlled by the parallelism field of each component.

Before testing a deployment, the involved clusters need to be setup and correctly configured for the first run. For configuring virtual clusters, this is done by interacting with IM, while for physical clusters, OSCAR-P connects via SSH to the front node of the Kubernetes (K8s for short) cluster, and then cordon or uncordon the worker nodes to reach the required number. Once the clusters are configured correctly, the profiling of a specific run can take place. In order to modify the configuration of the clusters to adapt it to the next runs, the cluster configurator can change the number of worker nodes.

After configuring the clusters to suit the requirements of a particular run, OSCAR-P creates a descriptive YAML file detailing all the information needed to run the test in a single location. This file contains the list of all the services, reporting for each one their requirements in terms of memory and cores, their input and output buckets and the associated Docker images. It also contains a description of the clusters in use, their endpoints, credentials and configuration, as well as information on the input files to be used to start the run, their number and the timing of their uploads. This descriptive file is updated with all the subsequent runs of a deployment, and in the end it serves as a detailed summary of the whole testing campaign.

The run description YAML file is parsed to extract all its relevant information before the run can start. OSCAR must also be cleaned by removing all the remnants of past executions (if any), such as buckets, old logs and services, to ensure that they will not interfere with the current run. OSCAR-P then generates an FDL file containing the information needed to build the new workflow, meaning the required services and buckets, and uploads it to OSCAR so that it can be applied.

The run is started by moving the required files in the input bucket of the first service, which triggers its execution. For the full application workflow tests, once the files are moved into the input bucket of the first component the workflow proceeds by itself, since every component output bucket is the input of another component (the input parser checks this assumption in the first steps), and OSCAR-P simply monitors the execution until its completion.

When testing single services instead, a component cannot write its output files in its assigned bucket or else it would trigger the execution of the next function. The solution adopted is connecting the tested components to a temporary empty input bucket, and to a temporary output bucket which does not trigger the execution of other components. The contents of the “real” input bucket are then copied to the dummy input bucket, just like at the start of the run with the storage bucket, triggering its execution.

After finalising each run, OSCAR-P proceeds to collect and process the logs. The logs are retrieved both from OSCAR and kubectl, and together they provide information on when a job (that is a single component execution) was scheduled, when its pod was created, and when it was actually started and finished; all this information is useful for checking delays, waits and overheads. The relevant logs information, also across multiple runs, are collected in a single CSV file which is used by the aMLLibrary to train a performance model for every service/resource pair.

7. SPACE4AI-D

The SPACE4AI-D (*System PerformAnce and Cost Evaluation on Cloud for AI applications Design*) tool tackles the component placement problem and resource selection in the computing continuum at design time, dealing with different AI application requirements in order to effectively orchestrate heterogeneous edge and cloud resources. Additionally, for each component multiple partitions may exist, since one of the features offered by the AI-SPRINT is the possibility of partitioning DNNs across different resources. The goal of the design tool is also to find the optimal DNN partitioning which guarantees the memory and QoS constraints. SPACE4AI-D receives following files as input:

- Resource description (candidate_resources.yaml) details hardware and software architecture
- Deployment descriptions (candidate_deployments.yaml) details the components such as its candidate execution layer and the detail of its container like the name of docker image, the memory size, compatible candidate resources, etc.
- Performance model (performance_models.json generated by OSCARP), includes the directory of pickle file to predict the performance of components
- Performance constraints (qos_constraints.yaml), details the local constraints of single components and global constraints related to the sequences of components.

- Application DAG (`application_dag.yaml`), details the components' sequence in the application modelled as a directed acyclic graph.
- Data transfer size (`components_data_size.yaml` generated by OSCARP), includes the data transfer size (expressed in KB) between every two consecutive components of application DAG
- Annotation (`annotations.yaml`), includes the expected throughput of the AI application.
- `SPACE4AI-D.yaml`, details the methods and parameters required to run SPACE4AI-D.

SPACE4AI-D finds the minimum cost solution while guaranteeing performance requirements (namely, requirements on the maximum admissible response times of single components or sequences of components). SPACE4AI-D uses a random greedy algorithm to find some feasible solutions and improve these solutions using a heuristic method. The heuristic methods implemented in SPACE4AI-D are as follows: *Tabu Search, Local Search, Simulated Annealing and Genetic Algorithm*.

After finishing the execution of the tool, it returns the optimal component placement, resource selection and the optimal number of nodes/VMs as "`production_deployment.yaml`" which helps the developer to find the optimal solution. The production deployment file can then be deployed through IM.

In the final release of SPACE4AI-D, the tool also finds the maximum arrival rate that still keeps the optimal solution feasible using a binary search. Therefore, the tool, in addition to the output optimal solution JSON file, generates an output JSON file with the same structure that includes the maximum arrival rate specified in the "Lambda" field, optimal resources, and the response time of all partitions under the maximum sustainable arrival rate. Since this output file is used directly by SPACE4AI-R optimizer, it is located in the AI-SPRINT Application/space4ai-r folder. In the case of multiple component versions (degraded-performance deployments), the maximum arrival rates are computed for each application configuration (corresponding to different *accuracy* levels) and, thus, the tool generates multiple output JSON files (one for each application configuration) including the maximum arrival rate corresponding to the application configuration, located in the AI-SPRINT Application/space4ai-r folder.

8. SCONE

SCONE is a framework to protect **data** (like training or inference data), and **code** (like Python programs, or AI models)

- **at rest** (i.e., on disk),
- **in flight** (i.e., on the network), and
- **in use** (i.e., in the main memory).

SCONE protects the confidentiality and integrity of the data and the code without needing to modify or recompile the application.

One novel aspect of **confidential computing** is that applications can be protected against privileged software like the operating system and the hypervisor. To certify that the data and code of a native application are adequately protected, we need to ensure that **all** software and hardware components are sufficiently protected. SCONE isolates each service individually. This reduces not only the attack's surface but also the effort to certify that an application is properly protected. The isolation enables us to securely outsource the management of components like the operating system and Kubernetes to a cloud/service provider. SCONE decouples the integrity and confidentiality of data and code from all other software components and from the entity that manages the software and hardware stack.

It is well known that many vulnerabilities are caused by using out-of-date software/hardware/firmware components. A second novel aspect of **confidential computing** is that one can **attest** all components that we need to ensure the confidentiality and integrity of our application. These components include the CPU, its firmware and the application code and its data itself. The **attestation** ensures that these components are up-to-date and no vulnerabilities are known for these components. In other words, one can establish trust in all components that are required to execute the application. In SCONE, this **attestation** is done transparently at each program start.

Using SCONE, one can transform - a.k.a. **sconify** - an existing AI application into a confidential application without needing to modify the application. Moreover, one can integrate this process into existing development and deployment workflows such as we have done with AI-Sprint Studio.

An AI application typically consists of one or more containers where a container usually hosts one service. A container is deployed with the help of a container image. Typically, a container image is created as part of a AI-SPRINT pipeline. One can transform a container image into a confidential container image with the help of container image **sconify_image**. This sconification is typically triggered as part of the pipeline. During development, one might **sconify** container images using a Linux, a Mac or a Windows machine. There is no need for a TEE environment.

We show how to build a container image such that the service deployed by this image runs automatically inside of an Intel SGX enclave. This workflow uses a native image as input. Typically, the native image is generated by an existing CI/CD pipeline. We translate this native image into a confidential image such that all files are protected and the service runs inside of an enclave. This is a single-step transformation using the command **sconify_image**:

```
$ sconify_image --from=new-application --to=new-application-confidential ...
```

Note that this step will be executed transparently within the Toscarizer pipeline based on the security annotations provided during design time. These security annotations are parsed by AI-SPRINT Design prior to the deployment and execution of the respective process and used to establish the appropriate configuration needed to ensure these properties. More details about the configuration, such as how these annotations will be used to form security policies is provided in D4.2 - Second release and evaluation of the AI-SPRINT security tools.

```

1 - applications:
2
3   - python3:
4     confProc: true
5     integrityProc: true
6     confRest: true
7
8     volumes:
9     - /mnt/data-in
10    - /mnt/data-out
11
12  - binary2:
13    confProc: true
14    integrityProc: true
15    confRest: true
16
17    volumes:
18    - /mnt/data-in
19    - /mnt/data-out
20
21    secureBoot:
22  - agent-0.8.0:
23    whitelist:
24    - 840f...72: /bin/agent
25  - AppArmor:
26    whitelist:
27    - 1e73...f6: /sbin/apparmor # hash of the executable
28
    
```

9. Getting Started

In the following, we provide a tutorial to demonstrate the use of the AI-SPRINT Studio. First, the instructions for the installation of the tools are provided. Then, we provide a guide on how to create a new AI-SPRINT application following the provided template. Finally, in Section 8, we provide a complete tutorial to execute the design of an AI-SPRINT application by considering the example *mask detection* application.

9.1 AI-SPRINT Studio Installation

Use the available Docker Image

A Docker image is publicly available to use the AI-SPRINT design tools. The image provides the AI-SPRINT design environment with the following tools:

- Design abstractions manager
- TOSCARIZER
- SPACE4AI-D
- SPACE4AI-D-partitioner
- OSCAR-P

Useful info

Get started with Docker here: <https://docs.docker.com/get-started/overview/>.

Here are the instructions to install Docker locally: <https://docs.docker.com/get-docker/>.

Execute the following commands from your terminal.

Step 1: Pull the AI-SPRINT Studio Docker Image

```
$ docker pull registry.gitlab.polimi.it/ai-sprint/ai-sprint-studio
```

Step 2a: Verification (AI-SPRINT Design)

```
docker run --rm registry.gitlab.polimi.it/ai-sprint/ai-sprint-studio aisprint --help
```

expected output

```
Commands:
  design
  new-application
  profile
  space4aid
```

Step 2b: Verification (TOSCARIZER)

```
$ docker run --rm registry.gitlab.polimi.it/ai-sprint/ai-sprint-studio toscarizer --help
```

expected output

```
Commands:
  delete
  deploy
  docker
  fdl
  outputs
  tosca
```

10. Code Examples

In the following, code examples are provided to familiarize yourself with AI-SPRINT concepts and modules. Learn how to build an AI-SPRINT application, to run the AI-SPRINT design tools, and to finally deploy the application.

10.1 Prerequisites

This demo is run on AWS services. In order to setup properly the Infrastructure Manager to deploy application components on AWS the following steps need to be performed:

1. Register a domain via Route 53, which should also already create a hosted zone
2. Add a random record to the newly created hosted zone

This is necessary as IM, when deleting an infrastructure, will also delete its created records; if none are left, it will also delete the hosted zone.

On the next deployment, a new hosted zone will be created, but with name servers different from the ones linked to the domain, and this will prevent access to OSCAR, MinIO and K8s. Adding a "fake" record to the initial hosted zone is a simple yet effective fix.

Note that, this step is not needed if the demo environment is already provided by the AI-SPRINT team (Polimi PhD students don't perform this step!).

10.2 Build a New AI-SPRINT Application

AI-SPRINT applications must comply with a predefined structure of folders and files, as defined in *Section 3 AI-SPRINT Application Architecture*. AI-SPRINT provides a project template in the Cookiecutter⁵ format (see Figure 10.1), which enables the automatic creation of such an application structure.

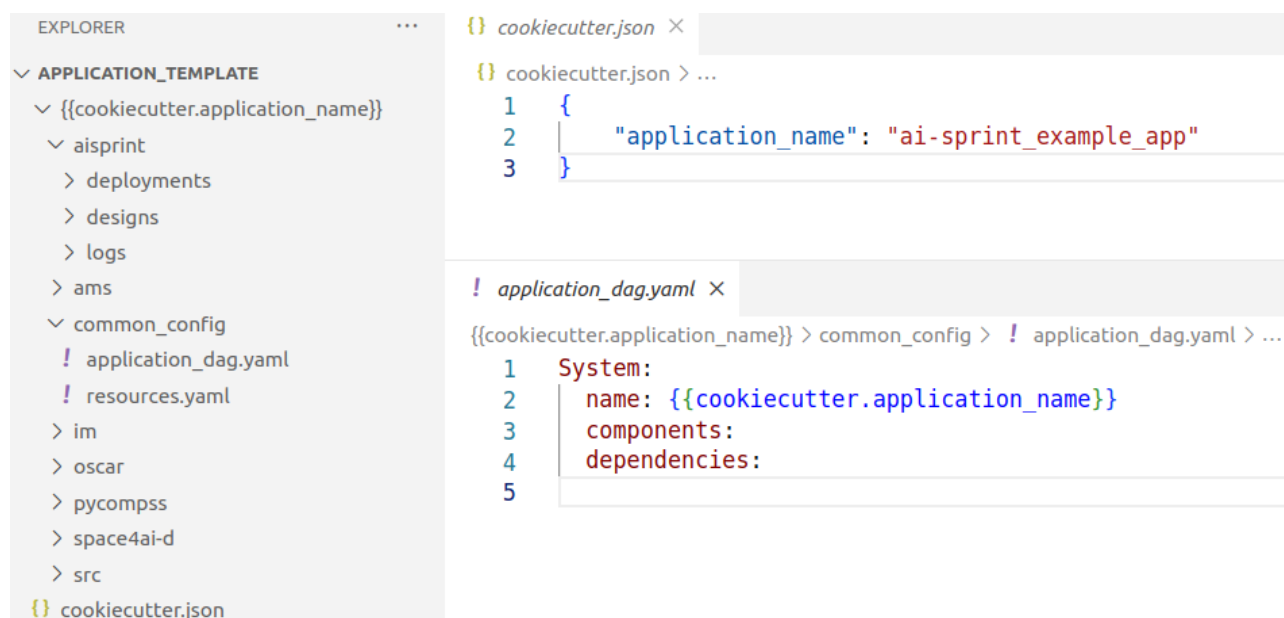


Figure 10.1 - Cookiecutter template used to automatically create the application with the structure expected by AI-SPRINT.

Create a new application with the AI-SPRINT

Step 1 Run *aisprint new-application*

```
$ docker run --rm \
-v $(pwd):/app_dir/ -w /app_dir \
--user $(id -u):$(id -g) \
registry.gitlab.polimi.it/ai-sprint/ai-sprint-studio \
aisprint new-application --application_name my_new_app
```

⁵ <https://github.com/cookiecutter/cookiecutter>

Step 2: Verification

```
$ cd /path/to/app_dir/my_new_app  
$ tree .
```

* tree can be installed by “*apt install tree*”

expected output

```
my_new_app  
├── aisprint  
│   ├── deployments  
│   ├── designs  
│   └── logs  
├── ams  
├── common_config  
│   ├── application_dag.yaml  
│   ├── candidate_deployments.yaml  
│   └── candidate_resources.yaml  
├── im  
├── oscar  
├── oscarp  
├── pycompss  
├── space4ai-d  
└── src
```

10.3 Execute AI-SPRINT Studio

In the following, an already prepared AI-SPRINT application is used as a base to demonstrate the AI-SPRINT design tools. The application is the Mask Detection application, whose execution workflow is reported in Figure 10.2. The application is composed of two main components: “*Anon and split*” and “*Mask detector*”. The former is in charge of anonymizing the video frames by blurring the detected faces, while the latter performs the detection of the masks. The original implementation of the application can be found at:

<https://gitlab.polimi.it/ai-sprint/scar/-/tree/master/examples/mask-detector-workflow>

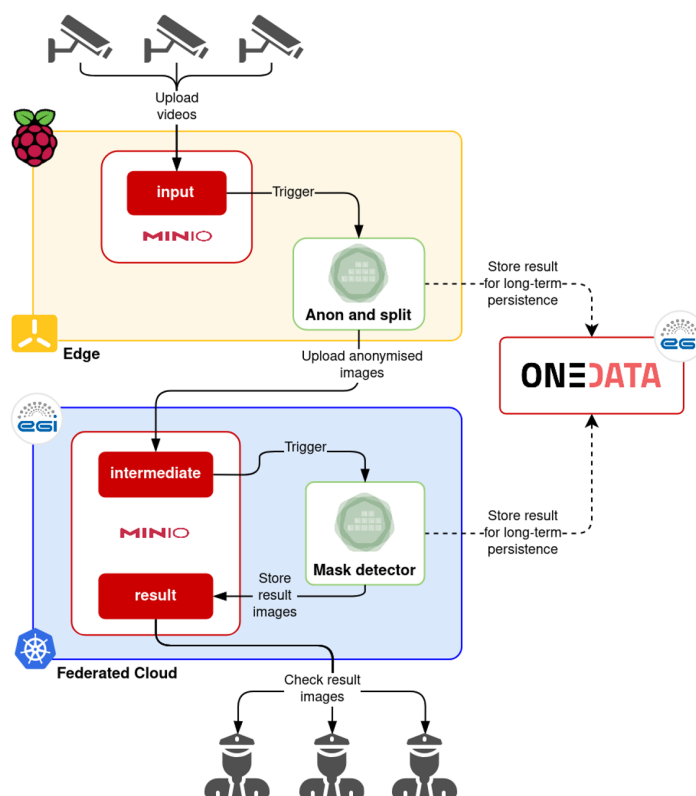


Figure 10.2 - Workflow of the mask-detection example application

Indeed, the goal is to determine if there are people not wearing face masks in crowds. The technological goal is to perform event-driven video processing involving anonymised mask detection based on pre-trained AI models. For this, video samples are captured through a set of cameras which perform a periodic upload to an OSCAR cluster running in a cluster of Raspberry Pis (closer to the edge of the network). For the sake of reproducibility, synthetic videos are being used instead of actual camera footage. These surveillance videos are preprocessed in order to periodically extract frames and blur the faces of people in the pictures using a Deep Learning inference application, thus ensuring that only anonymised data is sent to the Cloud, complying with established data protection regulations for Personally Identifiable Information (PII).

ONNX-Based Blur-Faces Component

In order to demonstrate the DNN-partitioning functionality introduced by AI-SPRINT (see *Section 5 AI-SPRINT Design Abstractions*), we developed an alternative mask-detection implementation in which the anonymization component, i.e., the “Anon and split”, relies on the use of the Open Neural Network Exchange (ONNX⁶) intermediate representation (IR). ONNX is a widely supported open format used to represent ML models that enables the interoperability between ML frameworks and tools, as well as the portability among different devices, by making it easier to optimise the computation depending on the hardware.

The ONNX-based blur-faces component has the same interface of the original one⁷, by taking as input an image and producing as output the anonymized image with the blurred faces. As in the original component, a DNN-based model is used to detect the faces in the image. In particular, the component uses the Receptive Field Block (RFB) Net Detector⁸, in which the top convolution layers of the Single Shot Detector

⁶ <https://onnx.ai/>

⁷ <https://gitlab.polimi.it/ai-sprint/scar/-/tree/master/examples/mask-detector-workflow/blurry-faces>

⁸ <https://arxiv.org/abs/1711.07767>

(SSD)⁹ are replaced by the RFB module. The Tensorflow implementation of this model, as well as its ONNX IR, can be found on GitHub: the repository¹⁰ contains different versions of the network, which differ for the number of parameters. We consider at this stage the *RFB-640* version, which the authors report as the one having the highest precision. We integrated the RFB Net Detector in the original “Anon and split” application, by preserving the input/output format. The inference of the ONNX model is performed through the ONNX Runtime¹¹, which allows accelerating models across several hardware platforms.

The predicted bounding boxes of the detected faces are filtered in the post-processing stage based on the confidence score. Finally, the detected faces are blurred using the *blur* function from OpenCV. An example of an anonymized frame from a short example video is provided in Figure 10.3.

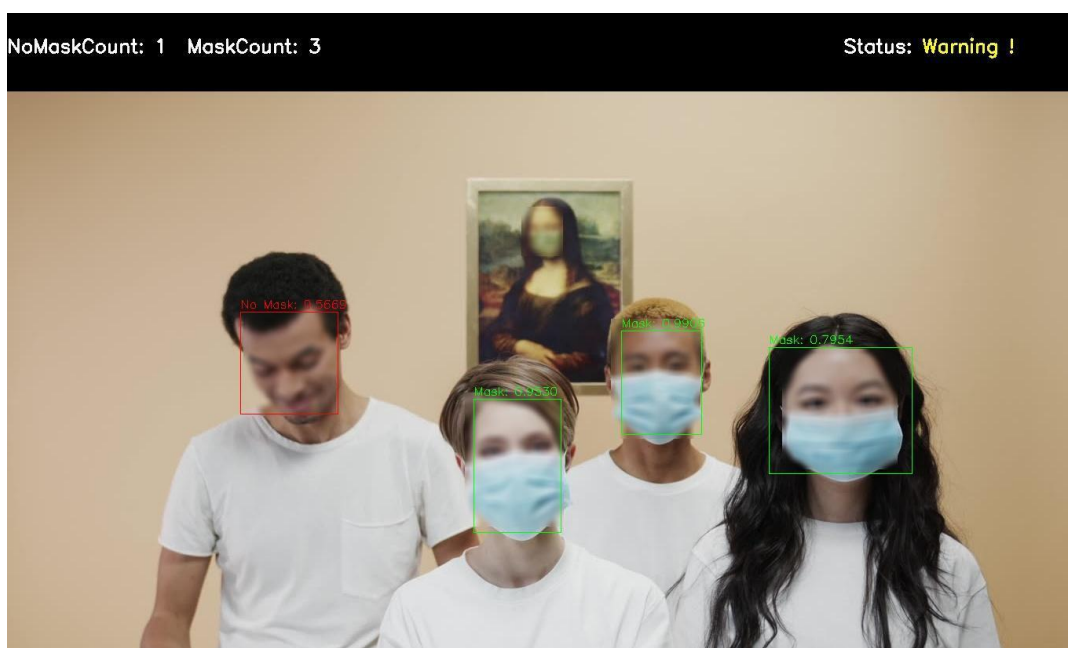


Figure 10.3 - Example of output of the Mask Detection application.

Mask Detection Component

The “mask detection” component is in charge of applying the mask recognition inference process on the output of the existing blur-faces component. The latter consists of a custom Deep Learning model based on YOLOv3, which takes as input the anonymised images and stores output images, including information about the detected masks, on the output bucket.

Candidate resource YAML file for Mask Detection application

Below we report the candidate resource file and candidate deployments for the application.

The file includes one network domain that connects two computational layers together with the identified access delay and bandwidth. The first computational layer includes AWS virtual machine, VM1 based on the t2.large flavour while VM2 is based on t3.xlarge.

⁹ <https://arxiv.org/abs/1512.02325>

¹⁰ <https://github.com/Linzaer/Ultra-Light-Fast-Generic-Face-Detector-1MB>

¹¹ <https://onnxruntime.ai/>

System:

name: Mask Detection Application PHD

NetworkDomains:

ND1:

name: Network Domain 1

AccessDelay: 0.00000277

Bandwidth: 40000

subNetworkDomains: []

ComputationalLayers:

computationalLayer1:

name: Public Cloud Layer

number: 1

type: Virtual

Resources:

resource1:

name: VM1

totalNodes: 5

description: t2.large

cost: 1.2

memorySize: 8192

storageSize: 450

storageType: SSD

operatingSystemDistribution: Ubuntu

operatingSystemType: Linux

operatingSystemVersion: 20.04

operatingSystemImageId: aws://us-east-1/ami-0149b2da6ceec4bb0

secureBoot: False

measuredBoot: False

onSpot: False

processors:

processor1:

name: Xeon

type: SkyLake

architecture: amd64

computingUnits: 4

internalMemory: 64

SGXFlag: False

computationalLayer2:

name: Public Cloud Layer

number: 2

type: Virtual

Resources:

resource1:

name: VM2

totalNodes: 5

description: t3.xlarge

cost: 1.8

memorySize: 16384

storageSize: 450

storageType: SSD

```
operatingSystemDistribution: Ubuntu
operatingSystemType: Linux
operatingSystemVersion: 20.04
operatingSystemImageId: aws://us-east-1/ami-0149b2da6ceec4bb0
secureBoot: False
measuredBoot: False
onSpot: False
processors:
  processor1:
    name: Xeon
    type: SkyLake
    architecture: amd64
    computingUnits: 4
    internalMemory: 64
    SGXFlag: False
```

Candidate deployments YAML file for Mask Detection application

Candidate deployment file, includes all the possible deployments of components. The first component (blurry faces), which is partitionable, has two alternative deployments. The first deployment has only one partition (full DNN) while the second one includes two partitions.

Each partition is a Python function running in a Docker container, thus, the image of docker has to be identified in the “image” field. Depending on the architecture of resources (ARM or AMD) we need to create different containers with different properties like memory size etc. for each partition. The candidate computational layers for the first and second component are 1 and 2, respectively, and the corresponding candidate resources for those components are VM1 and VM2, respectively.

```
Components:
component1:
  name: blurry-faces-onnx
  candidateExecutionLayers: [ 1 ]
Containers:
  container1:
    image: registry.gitlab.polimi.it/ai-sprint/toscarizer/blurry-faces-onnx_base_amd64
    memorySize: 1024
    computingUnits: 0.9
    trustedExecution: false
    networkProtection: false
    fileSystemProtection: false
    GPURequirement: false
    candidateExecutionResources: [ VM1 ]
component1_partitionX_1:
  name: blurry-faces-onnx_partitionX_1
  candidateExecutionLayers: [ 1 ]
Containers:
  container1:
    image: registry.gitlab.polimi.it/ai-sprint/toscarizer/blurry-faces-onnx_partition1_1_amd64
    memorySize: 1024
    computingUnits: 0.9
```

```
trustedExecution: false
networkProtection: false
fileSystemProtection: false
GPURequirement: false
candidateExecutionResources: [ VM1 ]
component1_partitionX_2:
name: blurry-faces-onnx_partitionX_2
candidateExecutionLayers: [ 1 ]
Containers:
container1:
image: registry.gitlab.polimi.it/ai-sprint/toscarizer/blurry-faces-onnx_partition1_2_amd64
memorySize: 1024
computingUnits: 0.9
trustedExecution: false
networkProtection: false
fileSystemProtection: false
GPURequirement: false
candidateExecutionResources: [ VM1 ]
component2:
name: mask-detector
candidateExecutionLayers: [ 2 ]
Containers:
container1:
image: registry.gitlab.polimi.it/ai-sprint/toscarizer/mask-detector_base_amd64
memorySize: 1024
computingUnits: 0.9
trustedExecution: false
networkProtection: false
fileSystemProtection: false
GPURequirement: false
candidateExecutionResources: [ VM2 ]
```

Let's start with the example!

Step 1: Download the Mask Detection application (*mask_detection_v1*)

```
$ cd /path/to/working/dir
$ wget -O mask_detection_v1.zip
"https://polimi365-my.sharepoint.com/:f/g/personal/10393616_polimi_it/EQKptxvhEftFk7RzyiRn1JcBu8
zRo-9URSGCph3geFN9g?e=hFFsRR&download=1"
$ unzip mask_detection_v1.zip
```

Let's check the structure of the application

```
$ tree mask_detection_v1
```


Expected output

```

mask_detection_v1
├── aisprint
│   ├── deployments
│   ├── designs
│   └── logs
├── ams
├── common_config
│   ├── application_dag.yaml
│   ├── candidate_deployments.yaml
│   └── candidate_resources.yaml
├── im
├── oscar
├── oscarp
├── pycompss
├── space4ai-d
│   └── SPACE4AI-D.yaml
├── src
│   ├── blurry-faces-onnx
│   │   ├── main.py
│   │   ├── onnx
│   │   │   └── version-RFB-640.onnx
│   │   ├── requirements.sys
│   │   ├── requirements.txt
│   │   └── utils.py
│   └── mask-detector
│       ├── cfg
│       │   ├── obj.names
│       │   ├── yolov3-tiny_obj_test.cfg
│       │   ├── yolov3-tiny_obj_train.cfg
│       │   └── yolov3-tiny_obj_train_tiny8.weights
│       ├── main.py
│       ├── requirements.sys
│       └── requirements.txt
    
```

Step 2: Run *aisprint design*

```

$ docker run --rm \
  -v $(pwd):/app_dir/ -w /app_dir \
  --user $(id -u):$(id -g) \
  registry.gitlab.polimi.it/ai-sprint/ai-sprint-studio \
  aisprint design --application_dir mask_detection_v1
    
```

Expected output

AI-SPRINT Design

```
[AI-SPRINT]: Starting the design of the AI-SPRINT application: mask_detection_v1

[AI-SPRINT]: Found 2 components in the DAG with the following dependencies:
- blurry-faces-onnx -> ['mask-detector']

[AI-SPRINT]: Parsing AI-SPRINT annotations..
[AI-SPRINT]: Done! Annotations parsed and stored in: mask_detection_v1/common_config/annotations.yaml

[AI-SPRINT]: Validating AI-SPRINT annotations..
[AI-SPRINT]: Done!

[AI-SPRINT]: Starting creating base components' designs..
[AI-SPRINT]: Done! Base designs created.

[AI-SPRINT]: Preparing for SPACE4AI-D-partitioner..
[AI-SPRINT]: Done! 'components_partitions.yaml' file initialized with base designs.

[AI-SPRINT]: Running SPACE4AI-D-partitioner..

[AI-SPRINT]: Running SPACE4AI-D-partitioner..
Maximum number of required partitions: 1
- Finding partitions of model: blurry-faces-onnx
67% ██████████ | 94/140 [00:13<00:06, 6.77it/s]

Done! Model partitioned at layers: ['266']

[AI-SPRINT]: Automatic generating the code of the found partitions..

- Generating code for the component: blurry-faces-onnx
Done! Code generated for segments: ['partition1_1', 'partition1_2']

[AI-SPRINT]: Starting creating base application deployment..
[AI-SPRINT]: Starting creating base application deployment..
Done! Current optimal deployment is 'base'

[AI-SPRINT]: Generating QoS constraints for the base deployment..
Done! QoS constraints generated.

[AI-SPRINT]: Done! Application designs and base deployment ready.
```

Verify the following files have been generated:

- “**annotations.yaml**” - It summarizes the parsed AI-SPRINT annotations. In particular, it is possible to observe from the summary that each component has a *component_name* annotation, which defines its name, and a local execution time constraint, which is of 20 seconds for the *blurry-faces-onnx* component and of 10 seconds for the *mask-detector* component. Finally, the *blurry-faces-onnx* component is annotated in order to be partitioned. Indeed, in the file there is the *partitionable_model* item with the *onnx_file* as value.

```
$ cat mask_detection_v1/common_config/annotations.yaml
```

Expected output

```
mask_detection_v1/src/blurry-faces-onnx/main.py:
  component_name:
    name: blurry-faces-onnx
  exec_time:
    local_time_thr: 20
  partitionable_model:
    onnx_file: version-RFB-640.onnx
mask_detection_v1/src/mask-detector/main.py:
  component_name:
    name: mask-detector
  exec_time:
    local_time_thr: 10
```

- “**designs**” - It contains the designs of the application components

```
$ tree mask_detection_v1/aisprint/designs
```

Expected output

```

mask_detection_v1/aisprint/designs
├── blurry-faces-onnx
│   ├── base
│   │   ├── main.py
│   │   ├── onnx
│   │   │   └── version-RFB-640.onnx
│   │   ├── requirements.sys
│   │   ├── requirements.txt
│   │   └── utils.py
│   ├── partition1_1
│   │   ├── main.py
│   │   ├── onnx
│   │   │   └── partition1_1.onnx
│   │   ├── requirements.sys
│   │   ├── requirements.txt
│   │   └── utils.py
│   └── partition1_2
│       ├── main.py
│       ├── onnx
│       │   └── partition1_2.onnx
│       ├── requirements.sys
│       ├── requirements.txt
│       └── utils.py
├── component_partitions.yaml
├── mask-detector
│   ├── base
│   │   ├── cfg
│   │   │   ├── obj.names
│   │   │   ├── yolov3-tiny_obj_test.cfg
│   │   │   ├── yolov3-tiny_obj_train.cfg
│   │   │   └── yolov3-tiny_obj_train_tiny8.weights
│   │   ├── main.py
│   │   ├── requirements.sys
│   │   └── requirements.txt

```

- “**deployments**” - It contains the base application deployment and the other additional deployments, together with the generated QoS constraints.

```
$ tree mask_detection_v1/aisprint/deployments
```

Expected output

```

mask_detection_v1/aisprint/deployments/
├── base
│   ├── ams
│   │   ├── qos_constraints_L1.yaml
│   │   └── qos_constraints_L2.yaml
│   ├── application_dag.yaml
│   ├── im
│   ├── oscar
│   ├── oscarp
│   ├── pycompss
│   ├── space4ai-d
│   │   ├── SPACE4AI-D.yaml
│   │   └── qos_constraints.yaml
│   ├── space4ai-r
│   └── src
│       ├── blurry-faces-onnx -> ../../../../designs/blurry-faces-onnx/base
│       └── mask-detector -> ../../../../designs/mask-detector/base
├── deployment1
│   ├── ams
│   │   ├── qos_constraints_L1.yaml
│   │   └── qos_constraints_L2.yaml
│   ├── application_dag.yaml
│   ├── im
│   ├── oscar
│   ├── oscarp
│   ├── pycompss
│   ├── space4ai-d
│   │   ├── SPACE4AI-D.yaml
│   │   └── qos_constraints.yaml
│   ├── space4ai-r
│   └── src
│       ├── blurry-faces-onnx_partition1_1 -> ../../../../designs/blurry-faces-onnx/partition1_1
│       ├── blurry-faces-onnx_partition1_2 -> ../../../../designs/blurry-faces-onnx/partition1_2
│       └── mask-detector_base -> ../../../../designs/mask-detector/base
├── multi_cluster_qos_constraints.yaml
├── optimal_deployment
│   └── production_deployment.yaml
    
```

- "**multi_cluster_qos_constraints.yaml**" - QoS constraints automatically generated for the AI-SPRINT AMS for all the possible deployments and layers.

```
$ cat mask_detection_v1/aisprint/deployments/multi_cluster_qos_constraints.yaml
```

Expected output

```

System:
  Deployments:
    base:
      ExecutionLayers:
        1:
          components:
            - blurry-faces-onnx
          local_constraints:
            local_constraint_1:
              component_name: blurry-faces-onnx
              threshold: 20
          global_constraints: {}
          throughput_component: {}
        2:
          components:
            - mask-detector
          local_constraints:
            local_constraint_1:
              component_name: mask-detector
              threshold: 10
          global_constraints: {}
          throughput_component: {}
      deployment1:
        ExecutionLayers:
          1:
            components:
              - blurry-faces-onnx_partition1_1
              - blurry-faces-onnx_partition1_2
            local_constraints: {}
            global_constraints:
              global_constraint_1:
                path_components:
                  - blurry-faces-onnx_partition1_1
                  - blurry-faces-onnx_partition1_2
                threshold: 20
            throughput_component: {}
          2:
            components:
              - mask-detector
            local_constraints:
              local_constraint_1:
                component_name: mask-detector
                threshold: 10
            global_constraints: {}
            throughput_component: {}
  
```

Step 3: Run TOSCARIZER

The TOSCARIZER

- automatically builds the Docker images of the components' designs generated in **Step2**
- automatically pushes the built Docker images to desired registry. It is required to log in the registry in order to push the images

Note: In this example we are going to use the dockerhub registry **docker.io** (see <https://hub.docker.com/>).

Ensure to be logged in the chosen registry. For dockerhub you can use **docker login**. You are required to use your username and password.

```
$ docker login
```

Run the TOSCARIZER

```
$ docker run --rm -t \  
-v $(pwd):/app_dir/ -w /app_dir \  
-v /var/run/docker.sock:/var/run/docker.sock \  
-v $HOME/.docker/config.json:/root/.docker/config.json \  
registry.gitlab.polimi.it/ai-sprint/ai-sprint-studio \  
aisprint toscarizer docker --registry docker.io --registry_folder registry_folder \  
--application_dir mask_detection_v1
```

Expected output

```
Building image: blurry-faces-onnx_base_amd64 ...  
Pushing image: blurry-faces-onnx_base_amd64 ...  
Building image: blurry-faces-onnx_partition1_2_amd64 ...  
Pushing image: blurry-faces-onnx_partition1_2_amd64 ...  
Building image: blurry-faces-onnx_partition1_1_amd64 ...  
Pushing image: blurry-faces-onnx_partition1_1_amd64 ...  
Building image: mask-detector_base_amd64 ...  
Pushing image: mask-detector_base_amd64 ...  
Docker images generated and pushed to the registry.  
DONE. mask_detection_v1/aisprint/designs/containers.yaml file created with new image URLs.
```

Check the “**containers.yaml**” file, which summarizes the built and pushed Docker images.

```
$ cat mask_detection_v1/aisprint/designs/containers.yaml
```

Expected output

```
components:  
  blurry-faces-onnx:  
    docker_images:  
      - docker.io/francescolattari/blurry-faces-onnx_base_amd64:latest  
      - docker.io/francescolattari/blurry-faces-onnx_partition1_2_amd64:latest  
      - docker.io/francescolattari/blurry-faces-onnx_partition1_1_amd64:latest  
  mask-detector:  
    docker_images:  
      - docker.io/francescolattari/mask-detector_base_amd64:latest
```

Step 4: Deploy Base Deployment

At the end of the AI-SPRINT design the *base* deployment is automatically generated and set as the current *optimal deployment*. The base deployment consists in the base solution which uses the original components’ implementations (not partitioned), which are assigned to the first available candidate

resources. Before running the OSCAR-P and SPACE4AI-D tools, the base deployment allows a ready-to-be-deployed version of the application, that can be used for testing purposes.

The base deployment is generated in the 'aisprint/deployments' folder of the application. Let's visualize it:

```
$ tree mask_detection_v1/aisprint/deployments
```

Expected output

```
mask_detection_v1/aisprint/deployments/
├── base
│   ├── ams
│   │   ├── qos_constraints_L1.yaml
│   │   └── qos_constraints_L2.yaml
│   ├── application_dag.yaml
│   ├── im
│   ├── oscar
│   ├── oscarp
│   ├── pycompss
│   ├── space4ai-d
│   │   ├── SPACE4AI-D.yaml
│   │   └── qos_constraints.yaml
│   ├── space4ai-r
│   └── src
│       ├── blurry-faces-onnx -> ../../../../designs/blurry-faces-onnx/base
│       └── mask-detector -> ../../../../designs/mask-detector/base
├── deployment1
│   ├── ams
│   │   ├── qos_constraints_L1.yaml
│   │   └── qos_constraints_L2.yaml
│   ├── application_dag.yaml
│   ├── im
│   ├── oscar
│   ├── oscarp
│   ├── pycompss
│   ├── space4ai-d
│   │   ├── SPACE4AI-D.yaml
│   │   └── qos_constraints.yaml
│   ├── space4ai-r
│   └── src
│       ├── blurry-faces-onnx_partition1_1 -> ../../../../designs/blurry-faces-onnx/partition1_1
│       ├── blurry-faces-onnx_partition1_2 -> ../../../../designs/blurry-faces-onnx/partition1_2
│       └── mask-detector_base -> ../../../../designs/mask-detector/base
├── multi_cluster_qos_constraints.yaml
├── optimal_deployment
└── production_deployment.yaml
```

The generated *production_deployment.yaml* file contains the summary of the resources selected for running the base deployment, as well as the assignment of the resources for each (base) component. The *optimal_deployment* is a symbolic link to the current optimal deployment folder that, in the current stage, is the base one. In order to use the base deployment the following steps must be executed:

- Run the TOSCARIZER to generate the TOSCA files for the two components of the base deployment

```
$ docker run --rm \
-v $(pwd):/app_dir -w /app_dir \
```



```
registry.gitlab.polimi.it/ai-sprint/ai-sprint-studio \
toscarizer tosca --application_dir mask_detection_v1 --optimal --domain domain_name
```

As output you will find two new generated files in “*aisprint/deployments/optimal_deployment/im*” that are the TOSCA files for the two components, i.e., *blurry-faces-onnx.yaml* and *mask-detector.yaml* files. The *domain_name* is needed to deploy on AWS and should be bought and configured on AWS Route 53

- The following is the template of the *auth.dat* file to be placed under *im/*:

```
id = im; type = InfrastructureManager; username = [IM username]; password = [IM password]
id = ec2; type = EC2; username = [access key ID]; password = [secret access key]
```

- Upload the TOSCA files to IM with the TOSCARIZER “deploy” command. The credentials for IM can be created by going to IM > Cloud credentials > New credentials > Infrastructure Manager:

```
$ docker run --rm -t \
-v $(pwd):/app_dir -w /app_dir \
registry.gitlab.polimi.it/ai-sprint/ai-sprint-studio \
toscarizer deploy --application_dir mask_detection_v1 --optimal
```

Step 5: Run OSCAR-P

OSCAR-P needs to interact with IM (<https://www.grycap.upv.es/im>) to deploy virtual infrastructure on AWS EC2. Copy the previously generated *auth.dat* file, if not already present, under *aisprint/deployments/base/im/*:

```
id = im; type = InfrastructureManager; username = [IM username]; password = [IM password]
id = ec2; type = EC2; username = [access key ID]; password = [secret access key]
```

Copy the URLs of the previously generated Docker images from the *aisprint/designs/containers.yaml* file in the “image” fields of the *common_config/candidate_deployments.yaml* file.

The testing campaign is controlled by the *run_parameters.yaml* file placed under *oscarp/*:

```
input_files:
  storage_bucket: "storage"
  filename: "Test-Video.mp4"
asynchronous:
  batch_size: 5
  number_of_batches: 2
  distribution: "deterministic"
  inter_upload_time: 20
```

```
synchronous:
  number_of_pre_allocated_pods: 2
  connect_timeout_seconds: 30
  request_timeout_seconds: 300
  worker_nodes: 4
  distribution: constant
  intervals:
    - throughput: 2
      number_of_threads: 1
      duration_seconds: 600
      ramp_up_seconds: 0
    - throughput: 5
      number_of_threads: 5
      duration_seconds: 600
      ramp_up_seconds: 10
  components:
    component1:
      parallelism: [ 10, 8, 6, 4, 2 ]
    component2:
      parallelism: [ 10, 8, 6, 4, 2 ]
      distribution: "deterministic"
  run:
    test_synchronously: False
    test_single_services: True
    train_models: True
    campaign_dir: "TEST"
    repetitions: 3
    cooldown_time: 30
  other:
    time_correction: 0
    domain_name: "polimi-demo.click"
    clean_infrastructures_before_testing: True
    clean_infrastructures_after_testing: True
```

According to this example, 10 copies of the input file will be used to trigger the test, and they will be uploaded in two batches of five, with an interval of 20 seconds between them. Two deployments will be tested, the first one involving blurry-faces and mask-detector and the second one involving the segments of partitioned blurry-faces and mask-detector. Each deployment will have 5 runs, with the parallelism level of every component going from 10 to 2. As each run will be repeated thrice, we'll have a total of 15 runs for both deployments.

The tables below describe in detail OSCAR-P input parameters.

Input_files

Field	Description
<i>storage_bucket</i>	<p>This is the name of the bucket where OSCAR-P will store the input files before starting the run. The bucket will be created on the OSCAR cluster hosting the first service of the application.</p> <p>Moving the input file into the input bucket from the storage bucket reduces latency to a minimum, which results in more accurate profiling data; if we were to upload the input files directly from the user computer, we'd have to take into account the network latency as well. Unless you have already created the storage bucket manually it'd be best to leave the default name.</p>
<i>filename</i>	The name of the file used for the tests. It needs to be either uploaded manually into the storage bucket, or stored under /OSCARP/input_files/

Asynchronous

<i>batch_size</i>	Number of files for every batch.
<i>number_of_batches</i>	Number of batches that will be uploaded.
<i>distribution</i>	Time distribution used to time the upload of the various batches, valid options are "deterministic" or "exponential".
<i>inter_upload_time</i>	<p>If the distribution used is "deterministic", "inter_upload_time" is plainly the time interval between the upload of different batches.</p> <p>If the distribution is "exponential", the time interval is defined as the maximum between inter_upload_time, and a sample from an exponential distribution where "inter_upload_time" is the scale.</p>

Synchronous

<i>number_of_pre_allocated_pods</i>	Number of hot pods on the OSCAR cluster, only applies to the first service
<i>ramp_up_seconds</i>	Thread creations will be divided on this interval, avoids bursts of requests
<i>number_of_threads</i>	Total number of threads to be used by Jmeter
<i>connect_timeout_seconds</i>	Connection timeout for each request
<i>request_timeout_seconds</i>	Response timeout for each request
<i>intervals</i>	A list of intervals that will be tested in sequence; each interval has a duration and a target throughput
<i>worker_nodes</i>	Number of nodes on which Jmeter is deployed. The more nodes, the greater the maximum workload that can be generated
<i>distribution</i>	Constant or exponential. Distribution of jobs generated by Jmeter

Parallelism

Field	Description
<i>parallelism</i>	A parallelism value defines the maximum number of instances that a component is allowed to have at the same time. A test will be performed for every value of the array, so the arrays of size three shown in the example above will translate to three tests. The parallelism arrays of all the components must have the same size.

Run

Field	Description
<i>campaign_dir</i>	<p>This is the name of the bucket where OSCAR-P will store the input files before starting the run. The bucket will be created on the OSCAR cluster hosting the first service of the application.</p> <p>Moving the input file into the input bucket from the storage bucket reduces latency to a minimum, which results in more accurate profiling data; if we were to upload the input files directly from the user computer, we'd have to take into account the network latency as well.</p> <p>Unless you have already created the storage bucket manually it'd be best to leave the default name.</p>
<i>repetitions</i>	
<i>cooldown_time</i>	

Other

Field	Description
<i>time_correction</i>	This is needed to align the times gathered from different sources, don't change it.
<i>domain_name</i>	The name of the registered domain associated with your AWS account, needed to correctly deploy the virtual infrastructures.
<i>clean_infrastructures_before_testing</i>	Delete all active machines before launching the entire simulation
<i>clean_infrastructures_after_testing</i>	Deletes all active machines after running the entire simulation

OSCAR-P can then be launched with the following command (note the `--dry_run` flag) to ensure that everything is configured correctly:

```
docker run -t \  
-v $(pwd)/mask_detection_v1:/mask_detection_v1 \  
--rm registry.gitlab.polimi.it/ai-sprint/ai-sprint-studio \  
aisprint profile --application_dir mask_detection_v1 --dry_run
```

The campaign can then be started with the following command:

```
docker run -it \  
-v $(pwd)/mask_detection_v1:/mask_detection_v1 \  
-v /var/run/docker.sock:/var/run/docker.sock \  
-v /tmp:/tmp \  
--rm registry.gitlab.polimi.it/ai-sprint/ai-sprint-studio \  
aisprint profile --application_dir mask_detection_v1
```

Expected output, recap of the whole campaign:

```
Deployments:  
  deployment_0: ['C1@VM1', 'C2@VM2']  
  deployment_1: ['C1P1.1@VM1', 'C1P1.2@VM1', 'C2@VM2']  
  
Testing deployment_0:  
Calculating hardware requirements...  
Component C1@VM1 can fit 4 container(s) on every node:  
  CPU needs to be decreased to (at least) 0.72 to fit more  
Component C2@VM2 can fit 2 container(s) on every node:  
  CPU needs to be decreased to (at least) 0.6 to fit more  
  
Deploying virtual infrastructures...  
Resource VM1 is being deployed...  
Resource VM2 is being deployed...  
  
Waiting for infrastructure deployment (this may take up to 15 minutes)...  
Done!  
  
Adjusting physical infrastructure configuration...  
Done!
```

Expected output, recap of the current deployment:

```
Starting Run #1 of 5 (full workflow)  
Workflow:  
  storage -> bucket0  
  bucket0 -> |blurry-faces-onnx| -> bucket1  
  bucket1 -> |mask-detector-onnx| -> bucket2  
  
Scheduler:  
  Run #1  
  Services:
```

blurry-faces-onnx cpu: 0.9 , memory: 1024 mb , parallelism: 10 , cluster: VM1
mask-detector-onnx cpu: 0.9 , memory: 1024 mb , parallelism: 10 , cluster: VM2

Clusters:

VM1 nodes: 3
VM2 nodes: 3

Run #2

Services:

blurry-faces-onnx parallelism: 10 -> 8
mask-detector-onnx parallelism: 10 -> 8

Clusters:

VM1 nodes: 3 -> 2
VM2 nodes: 3 -> 2

Run #3

Services:

blurry-faces-onnx parallelism: 8 -> 6
mask-detector-onnx parallelism: 8 -> 6

Clusters:

VM1 unchanged
VM2 unchanged

Run #4

Services:

blurry-faces-onnx parallelism: 6 -> 4
mask-detector-onnx parallelism: 6 -> 4

Clusters:

VM1 nodes: 2 -> 1
VM2 nodes: 2 -> 1

Run #5

Services:

blurry-faces-onnx parallelism: 4 -> 2
mask-detector-onnx parallelism: 4 -> 2

Clusters:

VM1 unchanged
VM2 unchanged

Repeated 3 time(s), 15 runs in total

Expected output, single run:

```
Removing services...
Removed service blurry-faces-onnx from cluster VM1
Done!

Removing buckets...
Removed bucket minio-VM1/bucket0/ from cluster VM1
Done!

Adjusting OSCAR configuration...
Checking correct OSCAR deployment...
Service blurry-faces-onnx deployed on cluster VM1
Service mask-detector-onnx deployed on cluster VM2
Done!

Moving input files...
Done!

Waiting for service blurry-faces-onnx completion...
100% | ██████████ | 5/5 [01:01<00:00, 12.21s/it]
Service blurry-faces-onnx completed!
Waiting for service mask-detector-onnx completion...
100% | ██████████ | 10/10 [00:31<00:00, 3.10s/it]
Service mask-detector-onnx completed!

Collecting OSCAR logs...
100% | ██████████ | 5/5 [00:03<00:00, 1.66it/s]
100% | ██████████ | 10/10 [00:04<00:00, 2.22it/s]
Done!

Processing logs...
Done!
```

The ML models will be generated at the end of all the runs of a service by using an automated library named aMLLibrary (<https://github.com/brunoquindani/aMLLibrary>). Its configuration file should be placed under *oscarp/*.

The following is an example of configuration file that uses sequential feature selection (SFS):

```
[General]
run_num = 1
techniques = ['LRRidge', 'DecisionTree', 'XGBoost', 'RandomForest']
hp_selection = KFold
folds = 5
validation = HoldOut
hold_out_ratio = 0.2
y = "avg_response_time"
hyperparameter_tuning = Hyperopt
hyperopt_max_evals = 10
hyperopt_save_interval = 0

[DataPreparation]
input_path = TEST/deployment_0/Full_workflow/results/Dataframes/blurry-faces-onnx_dataframe.csv
log = ["cores"]
inverse = ["cores"]
skip_columns = ["requested_throughput", "requested_parallelism", "warm_pods"]
product_max_degree = 2

[FeatureSelection]
method = "SFS"
max_features = 5
folds = 5

[LRRidge]
alpha = ['loguniform(0.01,1)']

[XGBoost]
min_child_weight = [1]
gamma = ['loguniform(0.1,10)']
n_estimators = [1000]
learning_rate = ['loguniform(0.01,1)']
max_depth = [100]

[DecisionTree]
criterion = ['mse']
max_depth = [3]
max_features = ['auto']
min_samples_split = ['loguniform(0.01,1)']
min_samples_leaf = ['loguniform(0.01,0.5)']

[RandomForest]
n_estimators = [5]
criterion = ['mse']
max_depth = ['quniform(3,6,1)']
max_features = ['auto']
min_samples_split = ['loguniform(0.1,1)']
min_samples_leaf = [1]
```


In the current configuration the library will train four models with techniques Ridge Regression, Decision Tree, XGBoost and Random Forest. 20% of the dataset will be used for validation and the hyperparameters will be tuned by using the Hyperopt framework. Moreover feature selection is enabled as well as automatic feature engineering, in the form of feature products/polynomial expansion up to the second degree. For additional details, please refer to the aMLLibrary official documentation.

Step 6: Run SPACE4AI-D

The parameters required to run Random Greedy and heuristics have to be provided in SPACE4AI-D.yaml. An example of SPACE4AI-D.yaml is shown below:

```
EdgeResources:
- computationalLayer1
CloudResources:
- computationalLayer2
- computationalLayer3
Methods:
method1:
  name: RandomGreedy
  iterations: 1000
  duration: 0
method2:
  name: GeneticAlgorithm
  startingPointNumber: 5
  iterations: 1
  duration: 1
  specialParameters:
    crossoverRate: 0.5
    mutationRate: 0.7
method3:
  name: BS
  upperBoundLambda: 2
  epsilon: 0.00001

Seed: 1
VerboseLevel: 1
Time: 1
```

Since the tool considers three main categories of resources: *Edge Resources*, *Cloud Resources* and *FaaS*, It must be specified that each computational layer belongs to *Edge Resources* or *Cloud Resources*. It is assumed that all *FaaS* configurations belong to a single computational layer, thus the last computational layer always belongs to the *FaaS* and it is not needed to be mentioned in the file.

In “Methods” section, the user specifies the methods. “method1” is always the Random Greedy method while “method2” can be one of heuristics. “method2” is not mandatory in the sense that the best result obtained by Random Greedy might be enough for the user. All the methods need two main parameters: “iterations” and “duration” which means how many iterations or how long (in second) the method must run. The tool runs the specified method based on the parameter that takes longer.

“method2” needs a general parameter called “startingPointNumber” which specifies the number of initial solutions that the heuristic methods explore. These initial solutions are found by Random Greedy. For

example, if the user set *startingPointNumber*: *k*, it means that *k*-best solutions of Random Greedy will be fed to the heuristics as the initial solutions.

Each heuristic method (except LocalSearch) needs some specific parameters to explore the solutions. The list of parameters are reported in the following table:

	Special Parameters	Description
Tabu Search	tabuSize	The memory size of the tabu list.
Genetic Algorithm	mutationRate crossoverRate	The probability of mutation. The probability of crossover.
Simulated Annealing	temperature scheduleConstant	Initial temperature. The initial temperature is gradually decreased according to "scheduleConstant". It must be between 0 and 1.

In order to find the maximum arrival rate that still keeps the solution feasible, SPACE4AI-D runs a binary search (mentioned as *method3*) to find the maximum arrival rate in the range of current arrival rate and an upper bound that can be reached for the application (*upperBoundLambda* field). The Binary Search is a mandatory method and another field, *epsilon*, is required to identify the difference between the minimum infeasible arrival rate and maximum feasible arrival rate to stop the binary search.

The rest Items, "Seed", "VerboseLevel", and "Time" identify the seed for generating random parameters of the methods, the verbose level for logging and the time horizon, respectively.

Run SPACE4AI-D

```
$ docker run -t \
-v {path to the project folder}/{folder name} \
--rm registry.gitlab.polimi.it/ai-sprint/ai-sprint-studio \
aisprint space4aid --application_dir {folder name}
```

After running SPACE4AI-D, it generates the optimal solution as a "production_deployment.yaml" file which includes all the information about the selected deployments as well as the selected resources assigned to the components. An example of *production_deployment.yaml* related to mask detection application is reported below:

```
System:
Components:
  blurry-faces-onnx:
    name: blurry-faces-onnx
Containers:
  container1:
    image: registry.gitlab.polimi.it/ai-sprint/blurry-faces-onnx:tag
    memorySize: 1024
    computingUnits: 0.9
    trustedExecution: false
    networkProtection: false
    fileSystemProtection: false
```

```
GPURequirement: false
selectedExecutionResource: VM1
executionLayer: 1
mask-detector:
  name: mask-detector
Containers:
  container1:
    image: registry.gitlab.polimi.it/ai-sprint/mask-detector:tag
    memorySize: 1024
    computingUnits: 0.9
    trustedExecution: false
    networkProtection: false
    fileSystemProtection: false
    GPURequirement: false
    selectedExecutionResource: VM2
  executionLayer: 2
Resources:
  name: Mask Detection Application PHD
NetworkDomains:
  ND1:
    name: Network Domain 1
    AccessDelay: 0.00000277
    Bandwidth: 40000
    subNetworkDomains: []
ComputationalLayers:
  computationalLayer1:
    name: Public Cloud Layer
    number: 1
    type: Virtual
    Resources:
      resource1:
        name: VM1
        totalNodes: 5
        description: t2.large
        cost: 1.2
        memorySize: 8192
        storageSize: 450
        storageType: SSD
        operatingSystemDistribution: Ubuntu
        operatingSystemType: Linux
        operatingSystemVersion: 20.04
        operatingSystemImageId: aws://us-east-1/ami-0149b2da6ceec4bb0
        secureBoot: False
        measuredBoot: False
        onSpot: False
        processors:
          processor1:
            name: Xeon
            type: SkyLake
            architecture: amd64
```

```
    computingUnits: 4
    internalMemory: 64
    SGXFlag: False
computationalLayer2:
  name: Public Cloud Layer
  number: 2
  type: Virtual
Resources:
  resource1:
    name: VM2
    totalNodes: 5
    description: t3.xlarge
    cost: 1.8
    memorySize: 16384
    storageSize: 450
    storageType: SSD
    operatingSystemDistribution: Ubuntu
    operatingSystemType: Linux
    operatingSystemVersion: 20.04
    operatingSystemImageId: aws://us-east-1/ami-0149b2da6ceec4bb0
    secureBoot: False
    measuredBoot: False
    onSpot: False
  processors:
    processor1:
      name: Xeon
      type: SkyLake
      architecture: amd64
      computingUnits: 4
      internalMemory: 64
      SGXFlag: False
```

The optimal configuration found by SPACE4AI-D can then be deployed through TOSCARIZER/IM by running the following commands.

First we need to generate the TOSCA templates for the recently generated optimal solution with the description of the full virtual infrastructure needed to deploy all the application components. For the correct OSCAR configuration a set of valid DNS names are assigned to the nodes to enable correct and secure external access to the services. A Route53 managed domain is required to make it work. You can set it with the “--domain” parameter (otherwise the default *im.grycap.net* will be used):

```
$ docker run --rm \
-v {path to the project folder}:/app_dir/ \
registry.gitlab.polimi.it/ai-sprint/ai-sprint-studio \
toscarizer toska --application_dir /app_dir \
--domain mydomain.my \
--optimal
```

Expected output:

```
DONE. TOSCA file /app_dir/aisprint/deployments/optimal_deployment/im/blurry-faces-onnx.yaml has
been generated for Component: blurry-faces-onnx.
DONE. TOSCA file /app_dir/aisprint/deployments/optimal_deployment/im/mask-detector.yaml has been
generated for Component: mask-detector.
```

To perform the infrastructure deployment, two additional files are required:

- The first one is “<app_path>/im/auth.dat” with the IM authentication file¹². The IM authentication file must contain the InfrastructureManager and all the cloud provider credentials that will be used in the deployment, including any already existing OSCAR cluster where a service has to be deployed. Moreover one line with an AWS credential (EC2 type) is needed to manage the DNS domain names used in the OSCAR TOSCA template. In case of using the default domain value (*im.grycap.net*) you should contact the authors to get a set of valid credentials. Otherwise you have to add some EC2 credentials to manage the specified domain (set in the TOSCA creation) in AWS Route53. The EC2 credentials must appear after any other cloud provider specified.

```
type = InfrastructureManager; username = user; password = pass
id = one; type = OpenStack; host = server:5000; username = user; password = pass; tenant = ten
id = oscar1; type = OSCAR; host = https://oscar.domain.com; username = user; password = pass
id = ec2; type = EC2; username = AK; password = SK
```

- The second one is “<app_path>/common_config/physical_nodes.yaml” with the information about the already deployed physical nodes. This file is only needed in case of using any type of resource “AlreadyDeployed”. This file should include all the needed data to contact with the set of already deployed resources. There are two different cases:
 - An already existing OSCAR cluster: In this case the needed data are the minio endpoint and credential (access key and secret key) and setting an oscar name.
 - A set of already existing machines: In this case the information about the SSH connection to access the nodes is required.

¹² <https://imdocs.readthedocs.io/en/latest/client.html#auth-file>

```
# Case of an already existing OSCAR cluster
ComputationalLayers:
  computationalLayer1:
    number: 1
    Resources:
      resource1:
        name: RaspPi
        minio:
          endpoint: https://minio.oscar.domain.com
          access_key: minio
          secret_key: pass
        oscar:
          name: oscar-test
```

```
# Case of already deployed nodes
computationalLayer2:
  number: 2
  Resources:
    resource1:
      name: PhysicalNode1
      fe_node:
        public_ip: 158.42.1.1
        private_ip: 192.168.1.2
        ssh_user: user
        ssh_key: |
          -----BEGIN RSA PRIVATE KEY-----
          ...
          -----END RSA PRIVATE KEY-----
    wns:
      - private_ip: 192.168.1.3
        ssh_user: user
        ssh_key: |
          -----BEGIN RSA PRIVATE KEY-----
          ...
          -----END RSA PRIVATE KEY-----
      - private_ip: 192.168.1.4
        ssh_user: user
        ssh_key: |
          -----BEGIN RSA PRIVATE KEY-----
          ...
          -----END RSA PRIVATE KEY-----
```

Then we use the `deploy` command to launch the infrastructure deployment. It will create one “infrastructure” per each application component. They will be created in order in case that some there are dependencies among the different components. This command will also wait for all the infrastructures to be correctly deployed.

```
$ docker run --rm \  
-v {path to the project folder}:/app_dir/ \  
registry.gitlab.polimi.it/ai-sprint/ai-sprint-studio \  
toscarizer deploy --application_dir /app_dir --optimal
```

Expected output:

```
Launching deployment for component mask-detector  
Launching deployment for component blurry-faces-onnx
```

Once deployed all the components resources we can get the TOSCA outputs where we can find all the needed information to access the deployed resources: OSCAR and MinIO endpoints, credentials, etc.

```
$ docker run --rm \  
-v {path to the project folder}:/app_dir/ \  
registry.gitlab.polimi.it/ai-sprint/ai-sprint-studio \  
toscarizer outputs --application_dir /app_dir --optimal
```

Expected output:

```
blurry-faces-onnx:  
  oscar_service_cred: service_token  
  oscar_service_url: https://oscar.domain.com/system/services/blurry-faces-onnx  
mask-detector:  
  dashboard_endpoint: https://oscar-cluster-1234.mydomain.my/dashboard/  
  oscarui_endpoint: https://oscar-cluster-1234.mydomain.my  
  minio_endpoint: https://minio.oscar-cluster-1234.mydomain.my  
  console_minio_endpoint: https://console.minio.oscar-cluster-1234.mydomain.my  
  admin_token: kubepass  
  oscar_password: oscarpass  
  minio_password: miniopass
```

Finally if the execution has finished and you do not need the resources you can delete them.

```
$ docker run --rm \  
-v {path to the project folder}:/app_dir/ \  
registry.gitlab.polimi.it/ai-sprint/ai-sprint-studio \  
toscarizer delete --application_dir /app_dir --optimal
```

Expected output:

```
Deleting infrastructure for component blurry-faces-onnx  
Infrastructure successfully deleted.  
Deleting infrastructure for component mask-detector  
Infrastructure successfully deleted.
```

Step 7: SCONIFYING a container

In order to SCONIFY docker containers used in an AI-SPRINT application,, developers can annotate tasks to receive certain security guarantees while executing. For this, we define the following constraints that can be used to annotate the main function of a Python script:

```
@security(trustedExecution=true,networkShield=true,filesystemShield=true)
```

We will now briefly recap the different properties and meaning of the above annotations.

The `trustedExecution` translates to the use of **trusted execution environments (TEE)**. Hence, the process that is executing the task will run in a TEE such as Intel SGX. This will also lead to the fact that such tasks will only be scheduled on processes that run on nodes that provide the necessary hardware support. If no such node is available, the task will run on a node that provides at least secure and measured boot mechanisms such that the used operating system and kernel can be trusted (for additional details, see AI-SPRINT Deliverable D4.3 -Final release and evaluation of the security tools).

The `networkShield` flag ensures that all TCP connections will be wrapped using SCONE's network shielding layer where all data that is read and written will be transparently encrypted/decrypted. Alternatively, service meshes such as Istio1 that provide secure communication will be deployed along the processes on a Kubernetes cluster.

Similar as with network encryption, the `filesystemShield` flag ensures that all files written and read by the process executing the tasks are encrypted. This is achieved by intercepting the system calls through SCONE in a similar fashion as for the network shielding.

All these flags will then be extracted from the source code files and put in a global annotations file which will then be considered during the **scnification** pass which is transparently executed in the pipeline of AI-SPRINT studio. Alternatively, this can be also triggered manually using the following command:

```
$ sconify_image_ai_sprint --from=dockerimage --to=dockerimage-confidential ...--application_dir /app_dir
```

The location of the application dir is important in order to extract the necessary information and perform the correct transformation of the native docker image into a secure one.

Expected output:

```
name:
mask-detector_base_amd64-8574/registry.gitlab.polimi.it-ai-sprint-toscarizer-mask-detector_base_amd64-sgx
version: "0.3"

# Access control:
# - only the data owner (CREATOR) can read or update the session
# - even the data owner cannot read the session secrets (i.e., the volume key and tag) or delete the session
```



```

access_policy:
  read:
    - CREATOR
  update:
    - CREATOR

services:
  - name: service
    image_name: service_image
    mrenclaves: [092a56e466d25aedadf1d185225421a168180215168af43a3c836c23ca05e6dd]
    command: "python3"
    environment:
      SCONE_MODE: hw

      PYTHON_VERSION: "3.8.14"
      PYTHON_SETUPTOOLS_VERSION: "57.5.0"
      LANG: "C.UTF-8"
      GPG_KEY: "E3FF2839C048B25C084DEBE9B26995E310250568"
      PYTHON_PIP_VERSION: "22.0.4"
      LD_LIBRARY_PATH: "/lib:/usr/lib:/usr/local/lib:"

      PYTHON_GET_PIP_SHA256:
"5aefe6ade911d997af080b315ebcb7f882212d070465df544e1175ac2be519b4"
      PYTHON_GET_PIP_URL:
"https://github.com/pypa/get-pip/raw/5eaac1050023df1f5c98b173b248c260023f2278/public/get-pip.py"
"
      PATH: "/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
      SCONE_HOST_PATH: "/etc/resolv.conf:/etc/hosts"
      pwd: /
      fspf_key: 893f70cfb76f1224e20b709246420c501e73194f20cd075bf3d9c4106261887d
      fspf_tag: b697ccb30cbd90b1b42c2bafbf9a149d
      fspf_path: /fspf/fs.fspf

images:
  - name: service_image

security:
  attestation:
    tolerate: [hyperthreading, insecure-igpu, outdated-tcb, software-hardening-needed,
insecure-configuration, debug-mode]
    ignore_advisories: "*"

Progress : [%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%] 100%: Finished
sconification
    
```

11. AI-SPRINT Runtime environment

The goal of the AI-SPRINT runtime environment is to support the automated deployment and monitoring of customized virtualized resources across the computing continuum. This encompasses the provision of computing resources to support the accelerated training of AI models as well as deploying the required services to support inference on the computing continuum together with the readaptation of components depending on varying conditions such as computing workload and network conditions.

This section will cover several components of the AI-SPRINT Runtime environment. First, the AI-SPRINT Monitoring Subsystem (AMS), which monitors various aspects of the system on various levels, such as cluster infrastructure resources, application performance, or any other application-specific metric. Second, the Secure Generative Data Exchange (SGDE), a synthetic data exchange tool based on privacy-preserving data generators. Third, the Privacy Preserving Component, a tool to train and retrain artificial neural networks providing high performance and security levels. Fourth, the AI-SPRINT Scheduling for Accelerated Devices, which aims to select the best scheduling and component allocation for deep learning training jobs on GPU-accelerated clusters, minimising the execution costs, while meeting deadline constraints. Finally, SPACE4AI-R (Runtime Tool for AI applications Resource Reconfiguration in Computing Continua) which periodically evaluates the resources allocation based on data gathered by the AI-SPRINT monitoring subsystem and updates the running components configuration (adding/removing/moving nodes, infrastructures, components and buckets) according to the varying conditions at runtime.

11.1 AI-SPRINT Monitoring Subsystem (AMS)

AMS provides the ability to collect and analyse various parameters describing the system state over time. The system is being monitored on multiple levels:

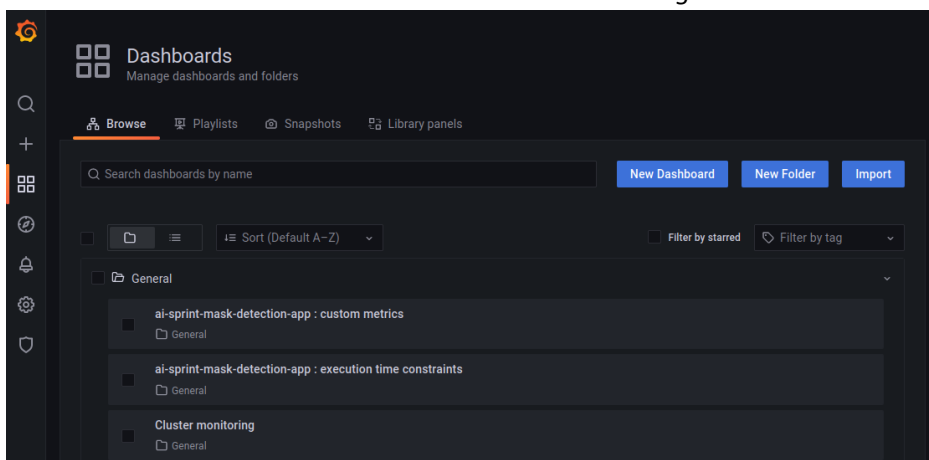
1. cluster level (infrastructure)
2. local and global AI-SPRINT (QoS) constraints
3. user-defined custom application-level metrics.

All this data is being collected while taking into account that some parts of the monitored system may be deployed on different clusters (layers) and that some elements may work in air-gap mode for prolonged periods of time, which is an important scenario for edge devices. Additionally, AMS raises alerts and emits notifications when a predefined set of constraints has been violated. The collected data can be analysed and visualised on automatically generated Grafana dashboards.

Furthermore, there is a component focused on application log gathering, based on the Elastic Stack and the AMS RESTful API is provided for other subsystems and components in order to communicate with AMS. This instruction focuses on metrics only, as the mask detector application does not integrate with the AMS log management framework.

11.1.1 Infrastructure monitoring

By default, AMS collects in the background various node metrics from every Kubernetes node. To analyse cluster data: in Grafana: *dashboards* → *cluster monitoring*.



Steps to change the scope of cluster monitoring and regenerate the dashboard:

1. Acquire the configuration file provided by the Infrastructure Manager, e.g., with:


```
kubectl -n ai-sprint-monitoring cp $(kubectl -n ai-sprint-monitoring get pods
            -l=app=ai-sprint-monit-manager -o
            jsonpath="{.items[0].metadata.name}"):templates/monitoring_setup.yaml monitoring_setup.yaml
```

```
monitoring:
  metrics:
    cpu: {}
    mem: {}
    swap: {}
    processes: {}
    system: {}
    kernel: {}
    disk:
      ignore_fs: ["hostfs", "tmpfs", "devtmpfs"]
      diskio: {}
      kubernetes: {}
  time_period: 20s
  parameters:
    performance_metrics_time_window_width: 120s
    default_notification_endpoint: http://10.96.234.
```

```
alerts: {}
```

- Edit the configuration file (`monitoring_setup.yaml`), e.g., remove all metrics but `cpu`,


```
monitoring:
  metrics:
    cpu: {}
  time_period: 20s
  parameters:
    performance_metrics_time_window_width: 120s
    default_notification_endpoint: http://10.96.234.
  alerts: {}
```
- Upload the updated configuration file with:


```
kubectl -n ai-sprint-monitoring cp monitoring_setup.yaml $(kubectl -n ai-sprint-monitoring get pods
            -l=app=ai-sprint-monit-manager -o
            jsonpath="{.items[0].metadata.name}"):templates/monitoring_setup.yaml
```
- Trigger re-configuration with:


```
kubectl -n ai-sprint-monitoring exec deployment/ai-sprint-monit-manager -- ./setup_monitoring.sh
```
- The dashboards update automatically in the background.

11.1.2 QoS monitoring

The default configuration depends on the application annotations and is provided by AI-SPRINT Design Tools. To analyse QoS data: in Grafana: *dashboards* → *execution time constraints*.



Normally, updates should follow the aforementioned AI-SPRINT design → TOSCARIZER → IM → OSCAR pipeline. However, manual changes are possible. Steps to manually amend QoS configuration:

- Edit the configuration file under `aisprint/deployments/XXX/ams/qos_constraints_YYY.yaml`.

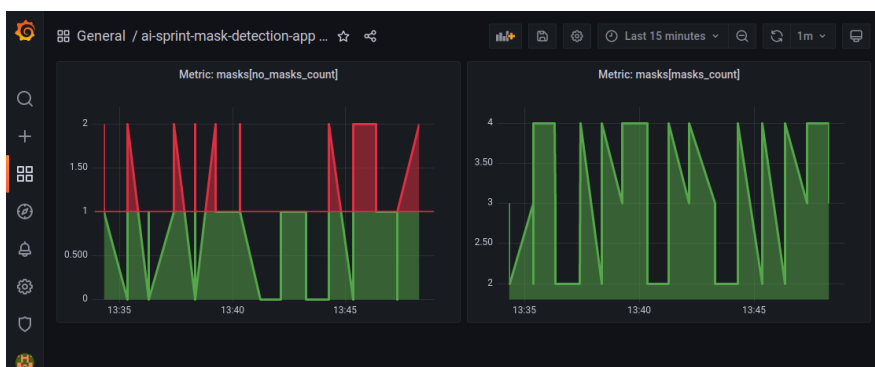
```
system:
  name: ai-sprint-mask-detection-app
  local_constraints:
    local_constraint_1:
      component_name: blurry-faces-onnx
      threshold: 15
    local_constraint_2:
      component_name: mask-detector-onnx
      threshold: 20
  global_constraints:
    global_constraint_1:
      path_components:
        - blurry-faces-onnx
        - mask-detector-onnx
      threshold: 20
  throughput_component: blurry-faces-onnx
```

- Update the configuration with:
`kubectl -n ai-sprint-monitoring cp qos_constraints_YYY.yaml $(kubectl -n ai-sprint-monitoring get pods -l=app=ai-sprint-monit-manager -o jsonpath="{.items[0].metadata.name}"):templates/qos_constraints.yaml`
- Trigger re-configuration with:
`kubectl -n ai-sprint-monitoring exec deployment/ai-sprint-monit-manager -- ./setup_app.sh -n oscar-svc`
- The dashboards update automatically in the background.

For the curious ones, internal calculations and constraints syntax have been thoroughly described in AI-SPRINT Deliverables *D2.3* and *D3.4*.

11.1.3 Custom metrics

AMS allows monitoring user-defined metrics and, as already mentioned, the mask detector application reports detected number of people with and without masks.



Steps to define a custom metrics dashboard and an alert:

- Create the following `custom_setup.yaml` file:

```
monitoring:
  name: ai-sprint-mask-detection-app
  metrics:
    masks:
      fields:
        - no_masks_count
        - masks_count
  alerts:
    no_mask_detected:
      condition:
        metric:
          name: masks
          field: no_masks_count
        threshold:
          type: '>'
          value: 1
      every: 30s
      notification_endpoint:
        url: ''
```

with app name, metric and field names matching the deployed application.

- Upload the configuration with:
`kubectl -n ai-sprint-monitoring cp custom_setup.yaml $(kubectl -n ai-sprint-monitoring get pods -l=app=ai-sprint-monit-manager -o jsonpath="{.items[0].metadata.name}"):templates/custom_setup.yaml`
- Trigger (re-)configuration with:
`kubectl -n ai-sprint-monitoring exec deployment/ai-sprint-monit-manager -- ./setup_app_custom.sh`
- The dashboards update automatically in the background. To analyse: in Grafana: `dashboards` → `custom metrics`.

Additionally, it is possible to integrate a custom metric alert with an external API. In case the metric value exceeds the defined threshold a HTTP post will be sent to the specified URL.

11.5 SPACE4AI-R

SPACE4AI-R is a set of runtime components in charge to adapt the deployment to the run conditions (network connectivity and congestion, CPUs load, etc.).

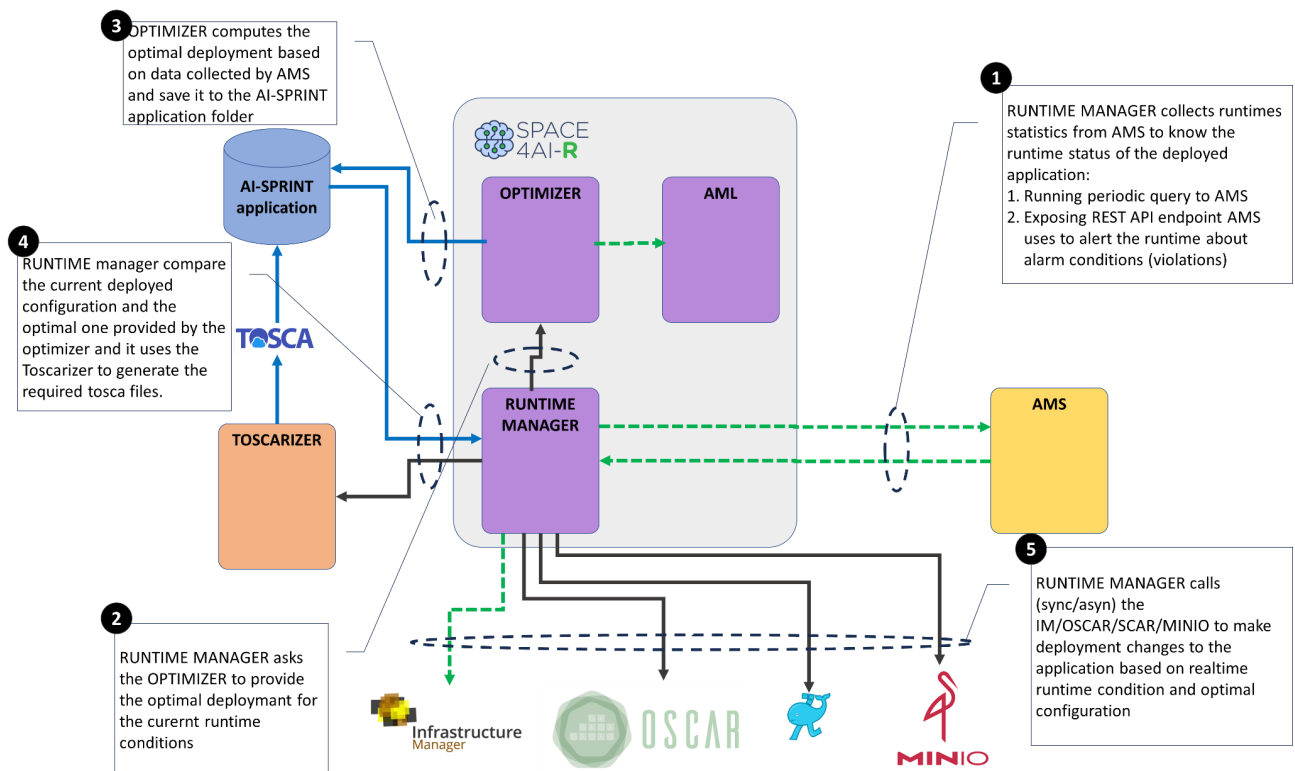
SPACE4AI-R consist of:

1. the **OPTIMIZER** that, based on component load and network congestion, computes the optimal application deployment;
2. the **AML** that is a library queried by the Optimizer to make the optimal deployment design;
3. the **RUNTIME MANAGER** that orchestrates all the runtime tools and seamlessly apply (without data loss) the optimal deployment.

SPACE4AI-R has both:

1. active monitoring: deployment status is checked every two minutes;
2. push notification: REST API called by AMS.

Runtime components architecture and interaction and are the following:



Installation

The components are deployed within the AI-SPRINT application and they do not require any user action.

The source code is public available here:

[ai-sprint-eu-project/space4ai-r-optimizer \(github.com\)](https://github.com/ai-sprint-eu-project/space4ai-r-optimizer)

[ai-sprint-eu-project/space4ai-r-runtime-manager \(github.com\)](https://github.com/ai-sprint-eu-project/space4ai-r-runtime-manager)

Output

SPACE4AI-R output is the optimal deployment configuration that is stored in the AI-SPRINT application deployment folder (aisprint\deployments). The folder hosts the initial deployment (aisprint\deployments)\base) the optimal deployment (aisprint\deployments\optimal_deployment), the current deployment (aisprint\deployments\current_deployment) as far as an history of the applied configuration (aisprint\deployments\current_deployment_DATE_TIME) as in the following example:

```
|-- aisprint
| |-- deployments
| | |-- base
| | |-- current_deployment
| | |-- current_deployment_2023-10-18-12-47-43-768529
| | |-- current_deployment_2023-10-20-14-17-37-636495
| | |-- current_deployment_2023-10-20-14-26-18-165259
| | |-- current_deployment_2023-10-20-15-20-57-707952
| | |-- current_deployment_2023-10-20-16-35-34-479136
| | `-- optimal_deployment
| |-- designs
| `-- logs
|-- ams
|-- im
|-- oscar
|-- oscarp
|-- pycompss
|-- space4ai-d
|-- space4ai-r
-- src
```