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

D3.1 - First release and evaluation of the runtime environment

Work Package	WP3 | Requirements & Architecture Definition
Task	T3.1| Continuous Deployment
Lead author	Germán Moltó (UPV)
Contributors	Miguel Caballer (UPV), Sebastián Risco (UPV), Hamta Sedghani (POLIMI), Matteo Matteucci (POLIMI), Giacomo Verticale (POLIMI), Federica Filippini (POLIMI), André Martin (TUD), Patrick Thiem (C&H), Francesc Lordán (BSC), Radosław Ostrzycki (7BULLS), Federico Silla (UPV), Danilo Ardagna (POLIMI), Eugenio Lomurno (POLIMI)
Peer reviewers	Daniele Lezzi (BSC), André Martin (TUD)
Version	V1.0
Due Date	31/12/2021
Submission Date	19/12/2021

Dissemination Level

X	PU: Public
	CO: Confidential, only for members of the consortium (including the Commission)
	EU-RES. Classified Information: RESTREINT UE (Commission Decision 2005/444/EC)
	EU-CON. Classified Information: CONFIDENTIEL UE (Commission Decision 2005/444/EC)
	EU-SEC. Classified Information: SECRET UE (Commission Decision 2005/444/EC)

AI-Sprint - Artificial Intelligence in Secure PRIvacy-preserving computing coNTinuum, has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement no. 101016577.

Versioning History

Revision	Date	Editors	Comments
0.1	24/10/2021	Germán Moltó	ToC definition
0.2	02/11/2021	Germán Moltó	Included sections 1, 2, 3 and structure of 4
	04/11/2021	Miguel Caballer, Sebastián Risco	Included sections 4.2.1 and 4.7.2
0.3	10/11/2021	Federica Filippini	Included section 4.5.2
0.4	12/11/2021	Francesc Lordán	Included section 4.7.1
0.5	14/11/2021	Radosław Ostrzycki	Included section 4.3
0.6	15/11/2021	Federico Silla	Included section 4.5.1
0.7	17/11/2021	Hamta Sedghani, Danilo Ardagna	Included section 4.8.1
0.8	26/11/2021	Patrick Thiem	Included section 4.8.2
0.9	29/11/2021	Eugenio Lomurno	Included section 4.4 and 4.6
1.0	28/11/2021	Matteo Matteucci	Revised contributions in section 4.4 and 4.6
1.1	29/11/2021	Germán Moltó	Included sections 5 and 6. Final review. Adjusted references and TOCs
1.2	08/12/2021	Daniele Lezzi	Review of the whole document
1.3	08/12/2021	André Martin / Mashid Mehrabi	Review of the whole document
1.4	15/12/2021	Germán Moltó	Changes from reviewers and final formatting

Glossary of terms

Item	Description
AI	Artificial Intelligence
CMP	Cloud Management Platform
DevOps	Software development (Dev) and IT operations (Ops)
DNN	Deep Neural Network
FaaS	Functions as a Service
GPU	Graphics Processing Unit
IaaS	Infrastructure as a Service
OS	Operating System
RBAC	Role Based Access Control
TEE	Trusted Execution Environment
TLS	Transport Layer Security
SRS	Software Requirements Specification
TOSCA	Topology and Orchestration Specification for Cloud Applications

Keywords

Artificial Intelligence; Edge Computing; Computing Continuum; Programming Models; Runtime management;

Disclaimer

This document contains confidential information in the form of the AI-SPRINT project findings, work and products and its use is strictly regulated by the AI-SPRINT Consortium Agreement and by Contract no. 101016577.

Neither the AI-SPRINT Consortium nor any of its officers, employees or agents shall be responsible, liable in negligence, or otherwise however in respect of any inaccuracy or omission herein.

The contents of this document are the sole responsibility of the AI-SPRINT consortium and can in no way be taken to reflect the views of the European Commission and the REA.

Executive Summary

This document describes the first release and evaluation of the runtime environment developed by the AI-SPRINT project, while the second release and evaluation of the runtime environment (D3.3) is due at M24. It describes the components involved in this first release that support the continuous deployment and programming framework runtime, as well as the results of the preliminary tests on the technologies employed that support the design decisions.

The technological choices are taken considering the requirements elicited from the analysis of the three use cases specified in the project proposal and detailed in AI-SPRINT Deliverable *D1.2 - Requirements Analysis*. In addition, an application based on face mask detection has been used as a “lead by example” approach to showcase an inference workflow consisting of images anonymised in the edge using resource-constrained computing resources, i.e. a cluster of Raspberry Pis, while face mask detection is performed on a dynamically provisioned elastic Kubernetes on top of an OpenStack-based cloud deployment, showing the initial integration among several components of the AI-SPRINT portfolio.

Overall, the first release of the runtime environment includes components in the following categories. For each one, the main tools are identified and a brief summary of the developments performed for the first release is included. First, deployment tools that provide custom virtualised computing resources from Cloud back-ends and resources located in the edge to support cloud-edge orchestration, enabling the automatic deployment of AI application models and components, without manual provisioning. The main tool involved is the Infrastructure Manager (IM), which has been extended in this first release to provision minified Kubernetes distributions such as K3S, to be used for edge-based resources, to include SGX support in its OpenStack connector, including its elasticity connector.

Second, monitoring tools, to gather infrastructure and application-level metrics based on NoSQL time series databases, responsible for storing and analysing the collected metrics, including visual dashboards. This includes the definition of synchronisation mechanisms among instances of the monitoring infrastructure to collect data at the edge to be stored in Cloud-based resources. The main tools involved are InfluxDB for metrics collection and analysis together with Telegraf for gathering metrics data and local buffering. Automated deployment procedures have been developed in the first release.

Third, scheduling for accelerated devices, including both local and remote GPUs, to jointly solve resource planning, in order to decide the appropriate number of GPUs to assign to jobs. This includes the shared usage of remote GPU-based computing. One of the tools involved is rCUDA, which has been extended in the first release to support Docker containers, newer versions of both TensorFlow and CUDA. Fourth, the programming framework runtime, to perform the execution of workflows along the computing continuum and to exploit the parallelism of the underlying computing resources. This ranges from detecting the data dependencies among the components and the allocation of parallel tasks to the available computing resources along the continuum, using also the FaaS (Functions as a Service) computing model. OSCAR, which provides event-driven file-processing serverless workflow execution along the computing continuum, is employed. For the first release, OSCAR was extended to be deployed on Raspberry PI clusters, used for AI inference at the edge, to support synchronous invocations and to include the initial support for GPUs. Also, COMPSs is used to orchestrate the execution of tasks on top of any distributed platform to exploit its parallelism, targeting the computing continuum. In the first release COMPSs has been extended to be compliant with the FaaS paradigm and to improve the resource management for dynamic addition and removal of resources and to facilitate the agent deployment to set hierarchies.

Fifth, application reconfiguration, to dynamically reconfigure the computational resources and execution workflow to consider changes in the performance of the underlying infrastructure, including the migration

of tasks. This involves generating optimal components placement to be adapted at runtime depending on the underlying state of resources. One of the tools employed is SPACE4AI-R, to provide optimal component placement, planned to be delivered by M24. Another tool is Krake, an orchestrator engine for containerised workloads used for rCUDA client migration, when the network to access remote GPUs becomes a bottleneck (support for stateful applications and QoS, performance and energy related scheduling is under development).

Finally, federated learning and privacy preserving continuous training tasks have also started, while software components will be made available in the second year of the project.

Table of Contents

1. INTRODUCTION 8

1.1 Scope of the document 8
1.2 Target Audience 8
1.3 Structure of document 8

2. EXAMPLE APPLICATION: MASK DETECTION 9

3. AI-SPRINT ARCHITECTURAL OVERVIEW FOR THE RUNTIME ENVIRONMENT 11

3.1 Runtime Environment 12

4. FIRST RELEASE OF THE RUNTIME ENVIRONMENT: COMPONENTS AND EVALUATION 18

4.1 Template description of components 18
4.2 Deployment Tools 19
4.2.1 Infrastructure Manager 19
4.3 Monitoring Tools 23
4.3.1 Storage and data analysis engine 23
4.3.2 Data gathering, preprocessing and delivery 25
4.4 Federated Learning 28
4.5 Scheduling for Accelerated Devices 30
4.5.1 rCUDA 30
4.5.2 GPU Scheduler 33
4.6 Privacy Preserving Continuous Training 40
4.7 Programming Framework Runtime 42
4.7.1 COMPSs / PyCOMPSs 42
4.7.2 OSCAR 48
4.8 Application Reconfiguration 53
4.8.1 SPACE4AI-R 53
4.8.2 Krake 54

5. TOWARDS THE INTEGRATED FRAMEWORK FOR THE RUNTIME ENVIRONMENT 59

5.1 Integration for the Example Application 59
5.2 Integration plan for the Runtime Environment 63

6. CONCLUSIONS 65

7. REFERENCES 66

List of Figures

Figure 2.1 - Anonymised mask detection workflow in the computing continuum 9
Figure 2.2 - Mask detection output 10
Figure 3.1 - AI-SPRINT Architecture Overview 11
Figure 3.2 - AI-SPRINT architectural relationship among the components of the runtime environment 15
Figure 4.1 - InfluxDB deployment integration in the Infrastructure Manager Dashboard 22
Figure 4.2 - GPU Scheduler: Randomised construction procedure 35
Figure 4.3 - GPU Scheduler: Path relinking procedure 36
Figure 4.4 - GPU Scheduler results: Comparison among the total costs obtained by the proposed algorithms under different workload scenarios 38
Figure 4.5 - GPU Scheduler results: Comparison among the average percentage gain obtained by the proposed algorithms under different workload scenarios 39
Figure 4.6 - GPU Scheduler results in a prototype environment 40
Figure 4.7 - Three-layer infrastructure managed by COMPSs 46
Figure 4.8 - Execution time (left) and efficiency (right) depending on the size of the infrastructure 49
Figure 4.9 - Evolution of the number of requests handled by each device 47
Figure 4.10 - Using COMPs in the application's algorithm 48
Figure 4.11 - Cluster of Raspberry PIs with four nodes to support OSCAR running on K3S for the edge. 51
Figure 4.12 - State of the nodes in the OSCAR cluster along the execution time 52
Figure 4.13 - Evolution of the RAM and CPU usage during the execution across the different nodes 52
Figure 4.14 - Openstack flavor configuration for VM instances 57
Figure 4.15 - Continuous testing of all implemented Krake features 58
Figure 5.1 - Steps involved in the example application (anonymised mask detection) 59
Figure 5.2 - Components involved in the implementation of the example application (anonymised mask detection) 60
Figure 5.3 - Screenshot of BLISS to compare the images before and after the mask detection 62
Figure 5.4 - Architecture for the monitoring integration between OSCAR and InfluxDB 63
Figure 5.5 - Milestones for components development 63

List of Tables

Table 3.1 -WP-level responsibilities for the runtime environment 12
Table 4.1 - Deployment time for the Infrastructure Manager to provision a Kubernetes cluster of several sizes in different Clouds. 22

1. Introduction

1.1 Scope of the document

The aim of the AI-SPRINT "Artificial intelligence in Secure PRIvacy-preserving computing coNTinuum" project is to develop a platform 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. This document describes the first release and evaluation of the runtime environment, including the software assets that have been produced in the first year of the project and their evaluation.

1.2 Target Audience

The release and evaluation of the runtime environment document (first and second) is intended for internal use, although it is publicly available. The target audience is the AI-SPRINT technical team including all partners involved in the delivery of work packages 2,3 and 4 but it also serves as a reference for the developers of the three use cases of the project.

1.3 Structure of document

This document includes four main parts:

- The **Example Application**, which introduces a sample application that performs mask usage detection using pre-trained AI models along the computing continuum (involving edge and cloud resources).
- The **AI-SPRINT Architectural Overview for the Runtime Environment** that provides the summarised updated description of the main components of the platform regarding the runtime framework together with its links with the deployable infrastructure and the security mechanisms amongst all the components.
- The **First Release of the Runtime Environment: Components and Evaluation** section provides details about the technological components involved and how they have been evolved in order to support the use cases needs. Interactions among the components are also covered together with the evaluation carried out for each component.
- The **Towards the Integrated Framework for the Runtime Environment** section describes the roadmap towards the integration among the different components for the runtime environment.

2. Example Application: Mask Detection

The AI-SPRINT deliverable *D5.2 Use case implementation plan* is due on M12, whereas *D5.3 Initial implementation and evaluation* is due on M24. Therefore, the use cases are not yet ready to be used in the platform under development. For this first release, we consequently decided to focus on an external use case that serves as a technology demonstrator, to guide the technical decisions that can later be extended to the actual AI-SPRINT use cases. This "lead by example" approach will potentially simplify the adoption of the technology by the use cases. Figure 2.1 provides the architectural description of the example application, which focuses exclusively on the inference process of the lifecycle of AI applications, that is, using pre-trained AI models on acquired data files.

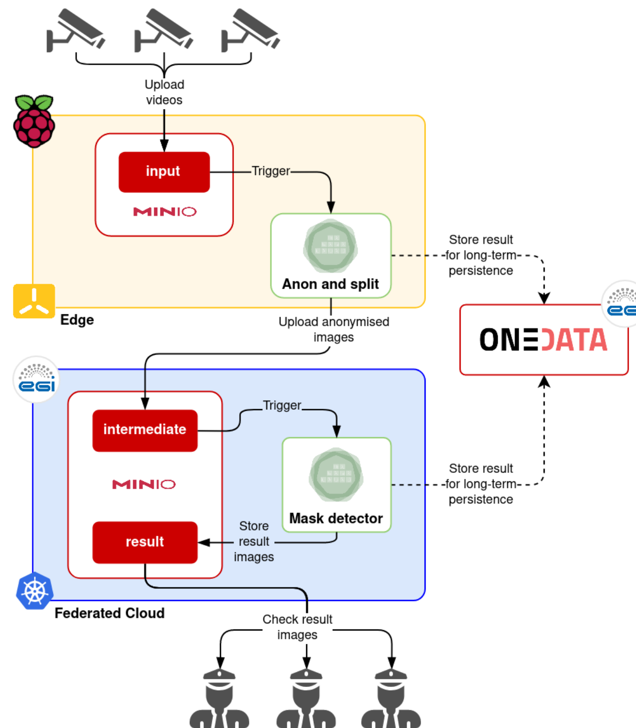


Figure 2.1 - Anonymised mask detection workflow in the computing continuum

The high-level 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 [2] 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 uploaded in the Raspberry Pi cluster (i.e., the Edge), where they are preprocessed in order to periodically extract frames and blur the faces of people in the pictures using a Deep Learning inference application [12], thus ensuring that only anonymised data is sent to the Cloud, complying with established data protection regulations for Personally Identifiable Information (PII). The face detection is performed by using a pre-trained single-shot multibox detection (SSD) MobileNet network, while the blur operation is computed on the detected faces by using the *blur* function provided by the OpenCV library. Uploading data to the storage MinIO [16] bucket of the cluster deployed in the Cloud will trigger the execution of the "Mask detection" OSCAR service in charge of applying the mask recognition inference process from an existing Face mask detector application [13], which consists of a custom Deep Learning model based on YOLOv3, on the anonymised pictures and generating output images that will be stored on the result bucket. As shown in

Figure 2.1, the resulting files can also be stored on external storage providers for long-term persistence, such as a Onedata provider [17] within the EGI DataHub [18].

Both AI-based applications, along with the weights of the pre-trained models, have been containerised to ensure the correct execution as OSCAR services in the computing architecture they are intended to run (including arm64 for "BlurryFaces" running in the Edge and amd64 for "face-mask-detector" running in the Cloud) [14].

A sample output of the inference workflow is shown in Figure 2.2, where colored bounding boxes are shown in the images depending on the proper wearing of a face mask (green) or not (red).

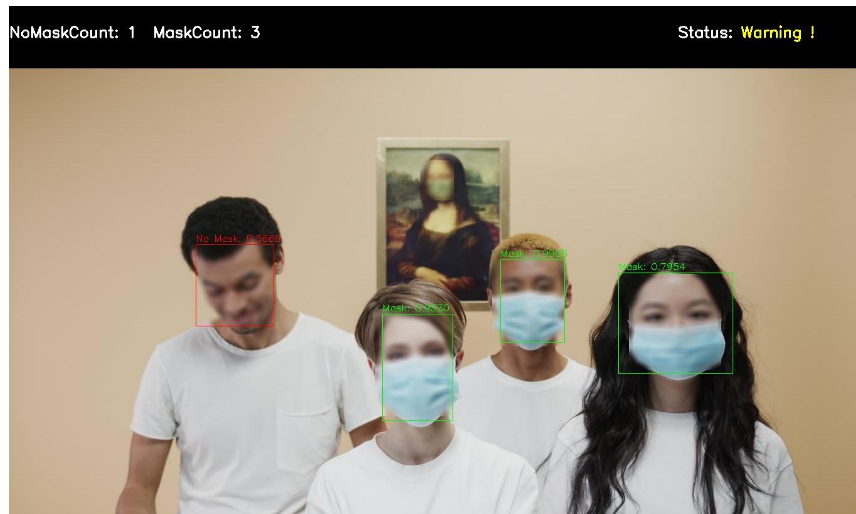


Figure 2.2 - Mask detection output

The following technological elements are available in this use case and can be later adopted by the project's use cases:

- The usage of **Docker** as the mechanism for encapsulating the AI model and the application required to perform the inference, in order to perform rapid application delivery across multiple platforms.
- The usage of **Kubernetes** as the underlying orchestrator of containers in order to simultaneously perform the inference process across multiple data files (images in this case).
- The usage of **OSCAR** as the event-driven serverless platform that triggers the execution of the Kubernetes jobs to perform the inference process in response to file uploads to the object storage system **MinIO**.
- The usage of **OpenStack** (not shown in the picture) as the underlying Cloud Management Platform (CMP) on which the Virtual Machines are dynamically deployed to create the virtualised Kubernetes cluster.

3. AI-SPRINT Architectural Overview for the Runtime Environment

The AI-SPRINT architecture overview was described in Deliverable *D1.3 Initial design of the architecture* [63]. This section summarises the components of the Runtime Environment. For each individual block, we describe its role and relation within the global architecture. An overview of the global AI-SPRINT architecture is shown in Figure 3.1.

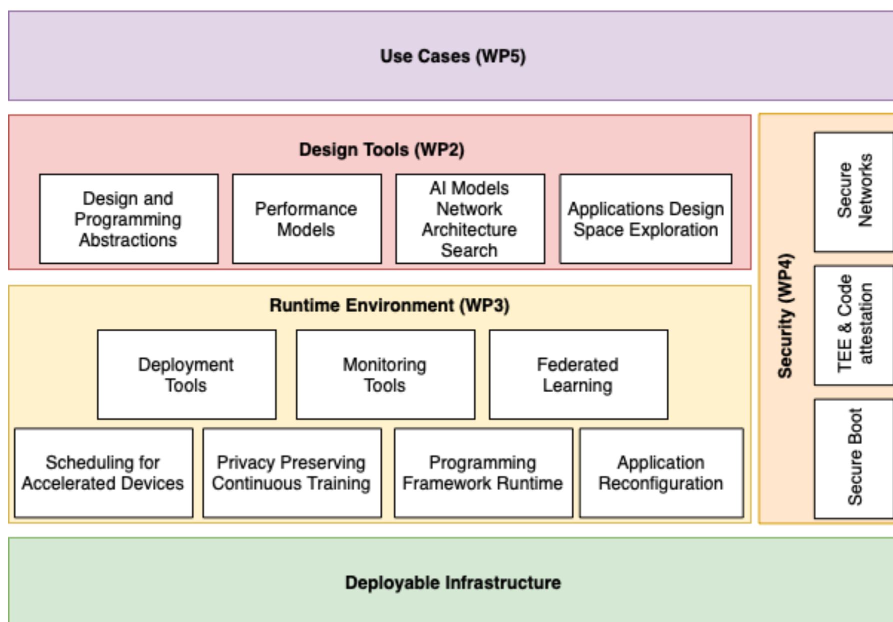


Figure 3.1 - AI-SPRINT Architecture Overview

The main responsibilities of each component of the Runtime Environment are the following:

- Deployment tools: automatic mechanisms to guide the process of configuring computing resources across the computing continuum and provide a cloud-edge orchestration enabling the automatic deployment of AI application models and components, without manual provisioning.
- Monitoring tools: gather performance and application-level metrics at every layer of the cloud-edge continuum.
- Federated learning: supports edge training and retraining through algorithms where different learners exchange only part of the information which can be extracted from the data in order to improve a model globally, but without making explicit or exposing the data used to train it.
- Scheduling for accelerated devices: solves the joint resource planning (i.e., how many GPUs to assign to a training job) and scheduling (i.e., when to run a training job) both for on-premises and public clouds.
- Privacy preserving continuous training: allows AI applications to update models in order to capture changes from the environment. Both changes in the weight matrices and internal model state (for, e.g., recurrent or LTSM based networks), or in the model structure by retriggering the AI model architecture search (developed in task T2.2) will be envisioned.

- Programming framework runtime: supports concurrent code execution, automatically detecting and enforcing the data dependencies among components and spawning parallel tasks to the available resources, which can be nodes in an edge cluster or cloud as well as edge devices such as mobiles.
- Application reconfiguration: periodically re-evaluates the initial resource allocation (identified by SPACE4AI-D developed in task T2.4) to consider load variations and increases in the components' response times. Application migration, to the extent that is possible, will be performed to counteract changes in the underlying performance of computing and network resources.

Table 3.1 summarises the WP-level responsibilities of the different components of the runtime environment, the lead maintainer and the major contributors. The complete table for the rest of the components is available in the AI-SPRINT deliverable *D1.3 Initial design of the architecture*.

Tool	WP	Task	Lead Maintainer	Major contributors
Deployment Tools	3	T3.1	UPV	UPV, TUD, POLIMI, BSC, BECK, C&H, 7BULLS
Monitoring Tools	3	T3.3	7BULLS	7BULLS, TUD, BSC, UPV
Federated Learning	3	T3.4	POLIMI	POLIMI, BSC, TUD, BSC, UPV, 7BULLS
Scheduling for Accelerated Devices	3	T3.5	UPV	UPV, POLIMI, 7BULLS, TUD
Privacy Preserving Continuous Training	3	T3.4	POLIMI	POLIMI, TUD, BSC, UPV, 7BULLS
Programming Framework Runtime	3	T3.2	BSC	BSC, POLIMI, UPV
Application Reconfiguration	3	T3.2	C&H	C&H, POLIMI, TUD

Table 3.1 -WP-level responsibilities for the runtime environment

D1.3 already described the motivation behind the adoption of the aforementioned tool categories together with the innovation required. Thus, the following sections directly dive into the runtime environment, briefly describing the specific tools in each category and providing relevant information concerning this first release.

3.1 Runtime Environment

The objective of the AI-SPRINT Runtime Framework is to support the automated deployment and monitoring of customised virtualised resources across the computing continuum infrastructure. This includes the orchestration of computations on that provisioned infrastructure in order to support the training of AI models and also exposing the trained models as a service using the FaaS (Functions-as-a-Service) paradigm for inference. Moreover, load variations may induce resource saturation or underutilization, which may have a strong impact on components' response times and execution costs. The Runtime Environment has therefore the goal of adapting the component placement to account for such events, by migrating component executions from edge to cloud and vice versa, by scaling the number of cloud resources and/or by changing DNN (Deep Neural Network) models partitions configuration.

Several software components have been adopted in AI-SPRINT in order to support the requirements identified within the use cases. The main ones are briefly described as follows:

- **Infrastructure Manager** (IM) [1], an open-source tool to deploy customised configurable application architectures described using the TOSCA (Topology and Orchestration Specification for Cloud Applications)[1] standard on multiple Cloud back-ends. These include widely used on-premises Cloud Management Platforms (CMPs) such as OpenNebula and OpenStack; public Cloud providers such as

[1] OASIS Topology and Orchestration Specification for Cloud. https://www.oasis-open.org/committees/tosca/

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.

- **SCAR** (Serverless Container-aware ARchitectures) [3], which supports compute-intensive functions packaged as Docker images on top of AWS Lambda, allowing also automated delegation of function execution into AWS Batch, a service that deploys elastic clusters on top of Amazon EC2, the IaaS managed service provided by Amazon Web Services (AWS). 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.
- **OSCAR** (Open-source Serverless Computing for Data-processing Applications) [2] supports the very same computing model offered by SCAR but within an IaaS Cloud or on edge resources. OSCAR consists of an elastic Kubernetes cluster, which can be dynamically deployed on all the Cloud providers supported by the IM. The cluster can grow and shrink in terms of the number of nodes, thanks to the **CLUES** [4] elasticity system, which inspects the status of the Kubernetes cluster to instruct the IM to provision or terminate the nodes to self-adapt to the current and expected workload. The use of a minified Kubernetes distribution running on ARM-based processors, such as those provided by Raspberry Pis, allows to execute OSCAR clusters for the inference in the edge of previously-trained Deep Learning models via a FaaS. These computing resources can be supplemented with on-premises CMPs and even public clouds to adapt to increased computing needs.
- **COMPSs / PyCOMPs** (COMPS Superscalar) [36] is a programming framework that aims to boost the productivity of parallel and distributed application developers. To ease the development of such applications, the programming model abstracts away from the developer all the details of the infrastructure and parallelism management. Application developers write their applications in a sequential fashion using a widely-used general-purpose language (Python, Java or C++) without references to the infrastructure where the code is meant to run. At execution time, a runtime system detects the parallelism inherent in the application and orchestrates the execution of the application component onto a set of available resources to achieve a shorter response time and an effective usage of the resources.
- **rCUDA** (remote CUDA) [35] is a middleware that provides remote GPU virtualisation, that is, it provides applications with virtual instances of a real GPU, which can therefore be concurrently shared. Furthermore, the real GPU being concurrently virtualised can be located in a node different from that executing the applications. The architecture of rCUDA follows a distributed client-server approach where the client part of the middleware is installed in the cluster nodes executing applications requesting GPU services, whereas the server side runs in the node owning the actual GPU. Communication between client and server is carried out through the network fabric in the cluster. rCUDA works in the following way: the client part of rCUDA, which is a library that resembles the CUDA library, receives a CUDA request from the accelerated application and appropriately processes and forwards it to the server middleware. On the server side, rCUDA receives the request and interprets and forwards it to the GPU, which completes the execution of the request and returns the execution results to the server middleware. Finally, the server sends back the results to the client middleware, which forwards them to the accelerated application.
- **KRAKE** [ˈkʀaːkə] is an orchestrator engine for containerised and virtualised workloads on distributed and heterogeneous cloud platforms. Often, these workloads consist of containerised applications (e.g., using Docker) running on Kubernetes clusters. Krake is able to schedule such workloads based on label constraints (e.g., a location constraint like Italy) and static or dynamically changing metrics (e.g., network latency) that can be fed through a metrics provider (e.g., Prometheus) to always find the most suitable environment for the applications to run. Krake is heavily inspired by the concepts of Kubernetes, and thus some similar internal functionality from Kubernetes has been abstracted and embedded into Krake. The idea behind Krake is to fluently and automatically schedule workloads

on instances such as Kubernetes clusters running in VMs of distributed cloud platforms. The architecture of Krake itself is loosely coupled, meaning that Krake's components can be replaced, extended, or abstracted. Besides that, Krake also comes with its own command line interface (CLI) called *rok* and supports security concepts, such as TLS, HTTPS and RBAC.

- **GPU Scheduler** determines the best scheduling and component allocation to minimise the costs related to the execution of Deep Learning training jobs, while meeting deadline constraints [43, 44]. The list of submitted jobs, in the form of Docker containers, together with information about their characteristics (expected execution times, collected through profiling, priorities and deadlines), together with a description of the system with all the available resources, are provided as input to the GPU scheduler. Based on this information, it determines which jobs should be run in the current time slot and the type and number of GPUs that should be assigned to them, to minimise energy costs (in the case of private GPU-accelerated clusters) or execution costs (in the case of public cloud), while meeting the deadline constraints. Disaggregated hardware architectures and remote GPUs can be accessed relying on rCUDA while Krake will be adopted to migrate rCUDA clients to alleviate network bottlenecks.
- **SPACE4AI-R** determines a new optimal deployment for the reference application, possibly changing the current assignment of application components to resources, their configuration according to the current load. SPACE4AI-R relies on Infrastructure Manager (IM) to deploy additional physical resources on cloud or edge and on OSCAR for component execution migrations. SPACE4AI-R is triggered by the monitoring tools (InfluxDB actions), which provide a snapshot of the current system status, in terms of execution time of the running components and resource consumption (e.g., CPU utilisations). According to this information, SPACE4AI-R checks whether QoS local and global constraints (namely, requirements on the maximum admissible response times of single components or sequences of components) are satisfied. If any constraint is violated, SPACE4AI-R determines a new optimal solution, adapting the current one to the actual load, and providing the updated deployment description to the Infrastructure Manager (IM) server. The new optimal solution may include changing the components configuration, and/or scaling the number of cloud VMs used to run specific components and migrating some components from edge to cloud or vice versa. Component migration will be supported by OSCAR/SCAR.

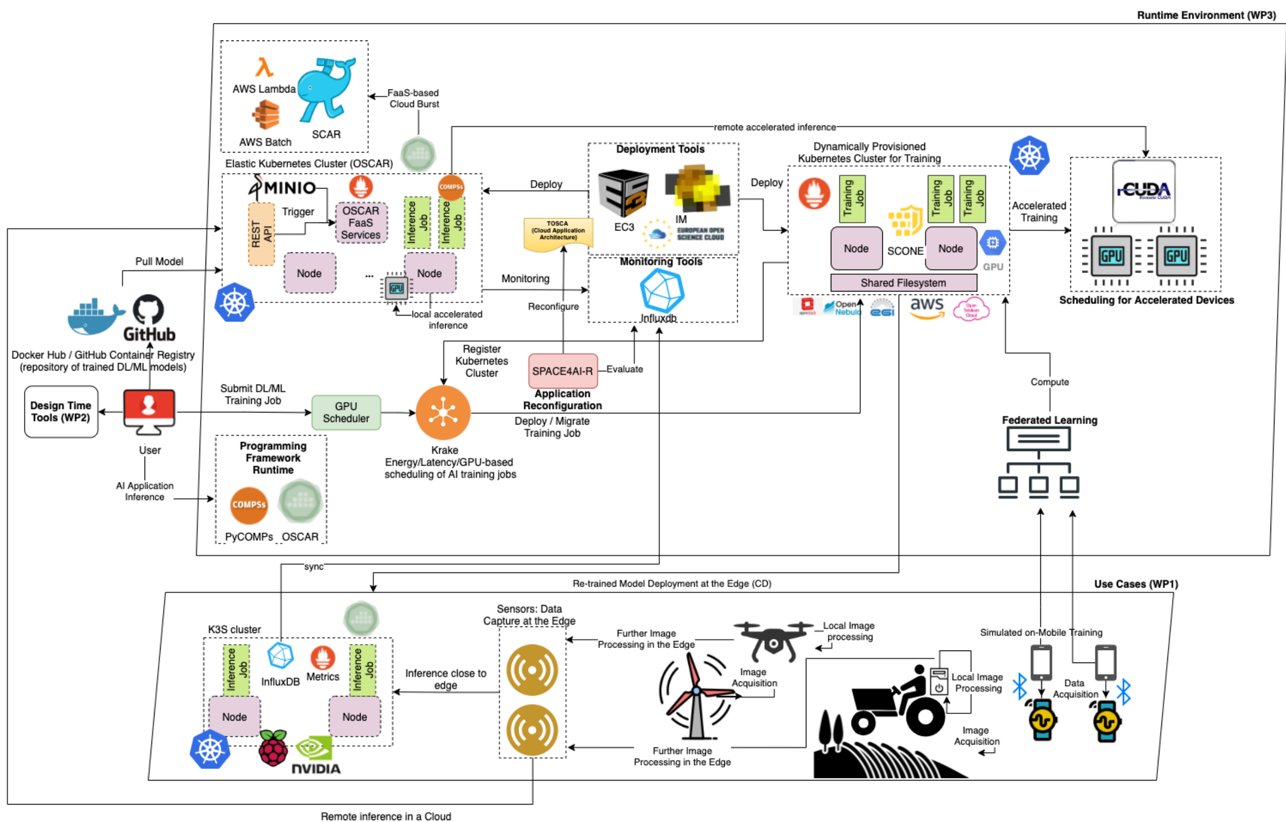


Figure 3.2 - AI-SPRINT architectural relationship among the components of the runtime environment

Figure 3.2 describes the main interactions among the different software components.

The runtime framework will provide several interfaces for provisioning virtual infrastructure to satisfy different user profiles. A REST API for programmatic access will facilitate application integration. A web-based GUI (Graphical User Interface) will allow end-users with limited technical skills to self-provision virtualised infrastructure for training models on specific infrastructures customised with certain software and hardware requirements. A CLI (command-line interface) will allow the savvy users to efficiently interact with the provided services in order to manage the lifecycle of dynamically provisioned virtual infrastructures.

The benefit of the runtime framework is to provide the "Design Tools" with the ability to deploy the required computing infrastructure and perform its automated configuration with the precise software artifacts to satisfy the user requirements. A wide range of customised virtual infrastructures are envisioned that include but are not limited to: Kubernetes clusters configured to support accelerated training using GPUs; Monitoring infrastructure, automatically deployed using a DevOps approach for the sake of repeatability; Virtual Machines with GPUs offered as a service using **rCUDA**[2] for remote acceleration of AI model training; OSCAR clusters to support FaaS for compute-intensive model inference.

AI-SPRINT also provides a programming framework runtime based on COMPSs and OSCAR that thanks to the integration with application reconfiguration solutions and to the continuous deployment service support the scheduling of dynamic workflows; these workflows can change their behaviour during the execution (i.e., according to the partial results of the application), to adapt applications and react to changes in the execution environment at large. Tasks execution can be migrated from edge to cloud servers simplifying the provisioning and management of AI applications lifecycle, including support (remote access) to edge and

[2] http://www.rcuda.net

accelerator devices. Developers can write AI/ML applications that will be orchestrated adopting a FaaS paradigm (which supports the execution of transient stateless functions) for the most effective and seamless use of the computing continuum resources.

For what concerns component application migration, **Krake** will be used. Krake provides a central point for managing component applications at each level of the computing continuum system. In this initial release of the runtime environment, Krake will be used to automatically migrate rCuda client instances. In this context, Krake will schedule two or more rCuda instances on at least two Kubernetes clusters based on predefined label constraints and latency metrics. Krake itself will run in a separate VM. Furthermore, in terms of security, connections will be secured through the usage of TLS and HTTPS. Intelligent decisions that will trigger component migration/cloud resource scaling/change in a DNN partition will be made by the **SPACE4AI-R** (*System PerformAnce and Cost Evaluation on Cloud for AI applications Runtime*) that will leverage OSCAR/SCAR or COMPSs runtime for enacting reconfiguration decisions.

Long running training jobs will be supported by the **GPU scheduler**. This receives job submissions, in the form of Docker containers, and determines the best scheduling and component allocation to minimise costs while meeting deadline constraints. The list of submitted jobs, together with their characteristics in terms of expected execution times (collected through profiling) and deadlines, and a description of the system with all the available resources, are provided as input to the GPU scheduler. Based on this information, the GPU scheduler determines which jobs should be run and the type and number of GPUs that should be assigned to them, in order to minimise energy costs (in the case of private GPU-accelerated clusters) or execution costs (in the case of public cloud), while meeting the deadline constraints. Disaggregated hardware architectures and remote GPUs can be accessed relying on rCUDA.

Data describing state and working parameters of all subsystems will be gathered by the **Monitoring** subsystem. It will preserve all collected time series metrics and allow system administrators and other subsystems to perform various actions by querying the time series. In particular, the Monitoring infrastructure will provide the possibility to create a self-healing system which will be able to adapt to changing environmental conditions or detected anomalies. Thanks to the advanced tools that will be developed, the Monitoring subsystem will also help analysing behaviour of the whole system in time and to find causes of bugs or failures. The Monitoring subsystem will also provide tools and mechanisms which allow to build hierarchical relationships among monitored systems. There will be the possibility to export collected or aggregated statistics from one monitoring instance to another one. Thanks to that feature the “upper” level of hierarchy will contain metrics from “lower” level. This will help analyse performance and working conditions of applications running on edge devices which may often work off-line (in network isolation).

Data privacy is a first-class citizen in the AI-SPRINT framework and thus tools for distributed computing and infrastructure management will be used to implement **Federated Learning** algorithms which are in charge of training models in a distributed fashion having data, errors, and gradients computed as close as possible to their source. The Monitoring infrastructure will be used to **trigger the retraining of models** and their update in the infrastructure which could go from simple weights update, if no major architectural changes are needed, to full application redeployment.

Already trained models will be exposed for inference using a serverless approach based on the **OSCAR** platform, which provides elastic Kubernetes clusters on which scalable AI inference can take place. These clusters can be deployed on low-powered devices such as Raspberry Pis to support AI inference at the edge. They can also be run on both on-premises and public Clouds to complex multi-step inference tasks. These inference workflows can be defined using its Functions Definition Language (FDL) [6] to perform fast local processing at the edge combined with compute-intensive tasks within a more powerful distributed infrastructure, even supporting GPU-based acceleration. If required, inference may take place within a FaaS service such as AWS Lambda [7] thanks to SCAR, to profit from the ultra-elastic capabilities of that service, supporting the concurrent execution of inference across multiple files in parallel. Cross-infrastructure

workflows ranging from edge, IaaS Clouds and FaaS (in AWS Lambda) can be defined in the aforementioned FDL combining the usage of OSCAR and SCAR.

4. First Release of the Runtime Environment: Components and Evaluation

This section describes the components of the runtime environment according to the Software Design Specification (SDS) standard (IEEE Standard 1016). Each component is described in terms of its external interfaces and dependencies with other components. In particular, this section focuses exclusively on the components employed in the runtime environment.

4.1 Template description of components

The following template is used as the structure to provide the information for each component involved in the runtime environment. The template is included here to make this document self-contained. A similar description is also reported in AI-SPRINT deliverables *D2.1 First release and evaluation of the AI-SPRINT design tools*, and *D4.1 Initial release and evaluation of the security tools.*

Identification	The unique name for the component and its location in the system
Type	A module, a subprogram, a data file, a control procedure, a class, etc.
Purpose	Function and performance requirements implemented by the design component, including derived requirements. Derived requirements are not explicitly stated in the Software Requirements Specification (SRS), but are implied or adjunct to formally stated SDS requirements.
Function	What the component does, the transformation process, the specific inputs that are processed, the algorithms that are used, the outputs that are produced, where the data items are stored, and which data items are modified.
High level Architecture	The internal structure of the component, its constituents, and the functional requirements satisfied by each part.
Dependencies	How the component's function and performance relate to other components. How this component is used by other components. The other components that use this component. Interaction details such as timing, interaction conditions (such as order of execution and data sharing), and responsibility for creation, duplication, use, storage, and elimination of components.
Interfaces	Detailed descriptions of all external and internal interfaces as well as of any mechanisms for communicating through messages, parameters, or common data areas. All error messages and error codes should be identified. All screen formats, interactive messages, and other user interface components (originally defined in the SRS) should be given here.
Data	For the data internal to the component, describe the representation method, initial values, use, semantics, and format. This information will probably be recorded in the data dictionary.
Needed improvement	Description of the needed improvements of this tool with regards the AI-SPRINT project, in order to fulfill the user requirements and to build the runtime environment

Implemented Improvements for the First Release	A description of the implemented improvements in the service to achieve the first release of the runtime environment.
Release Version & Repository	The software version released and the repository from where it can be downloaded.

4.2 Deployment Tools

The deployment tools provide automated mechanisms to guide the process of configuring computing resources across the computing continuum and provide a cloud-edge orchestration enabling the automatic deployment of AI application models and components, without manual provisioning.

4.2.1 Infrastructure Manager

Identification	Infrastructure Manager (IM)
Type	A service and a set of clients
Purpose	The IM is a service for the complete orchestration of virtual infrastructures and applications deployed on it, including resource provisioning, deployment, configuration, re-configuration and termination.
Function	The service manages the complete deployment of virtual infrastructures or individual components within them. The status of a virtual infrastructure can be: ● pending: launched, but still in initialisation stage; ● running: created successfully and running, but still in the configuration stage; ● configured: running and contextualised; ● unconfigured: running but not correctly contextualised; ● stopped: stopped or suspended; ● off: shutdown or removed from the infrastructure; ● failed: an error happened during submission;
High level Architecture	The following image describes the high-level architecture of the IM, including external dependencies.

	IM can optionally use catalogs for Virtual Machine Images (VMIs) such as the VMRC (Virtual Machine Image Repository and Catalog) [8] or EGI AppDB [9] to search for the most appropriate VMI according to the user requirement. Ansible [10] is used as the DevOps tool to perform the automated configuration out of a set of curated Ansible Roles available in Ansible Galaxy [11].
Dependencies	The IM service requires credentials to an IaaS Cloud on which to provision the virtual infrastructure. It supports multiple IaaS Cloud such as Amazon Web Services, Microsoft Azure, Google Cloud, OpenNebula, OpenStack, Open Telekom Cloud, Orange Cloud, EGI Federated Cloud. It can also provision resources directly from a Kubernetes cluster.
Interfaces	The IM service supports two APIs: ● The native one in XML-RPC ● A REST interface. It also includes a command-line Python client which interacts with the XML-RPC API. The IM supports both the native language, Resource Application & Description Language (RADL), and the OASIS standard TOSCA (Topology and Open Specification for Cloud Applications). A specific tool for the deployment of elastic virtual clusters through the Infrastructure Manager is also provided, named EC3 (Elastic Cloud Computing Cluster) [19], which provides both a command-line interface and a web portal.
Data	The IM uses three types of information ● Application descriptions following the OASIS TOSCA Simple Profile in

	YAML Version. ● Information about the cloud providers end-points and associated metadata to be used by the IM. ● Information about the deployed infrastructures (specifications, IDs, status, end-points, etc.) stored in a database. The client and the server exchange the data through the parameters of the API calls.
Needed improvement	In AI-SPRINT, the Infrastructure Manager is employed to perform customised deployment of virtual infrastructure across multiple infrustructures. This ranges from the deployment of minified Kubernetes clusters on top of Raspberry PIs, to automated configuration of OSCAR clusters across several IaaS Cloud back-ends. It also provides the users with simplified dashboards to self-provision the required virtualised infrastructure for training AI models, even with the usage of accelerated devices such as GPUs. The IM is also employed to perform infrastructure reconfiguration to satisfy increases in the workloads by deploying additional virtual machines.
Implemented Improvements for the First Release	The first release of the Infrastructure Manager regarding AI-SPRINT includes the following improvements: - Created Ansible Roles to deploy K3S (a minified Kubernetes distribution). - Created Ansible Roles and a TOSCA template to deploy InfluxDB as part of a Kubernetes cluster for the automated deployment of the monitoring infrastructure. - Add SGX support to the OpenStack connector. - Install Kubernetes SGX plugin. - Manage SGX resources in CLUES to manage elasticity.
Release Version & Repository	The first release version of the Infrastructure Manager for AI-SPRINT can be downloaded from the official repository: https://github.com/grycap/im/releases/tag/v1.10.6 and from the AI-SPRINT GitLab (which performs a daily mirror): https://gitlab.polimi.it/ai-sprint/im

4.2.1.1 Detailed Description of Activities for the First Release

This section provides additional details on the activities carried out within AI-SPRINT in IM for the first release:

Improvement of the Kubernetes Ansible role

The existing Ansible Role [45] created by UPV to deploy a Kubernetes cluster has been extended to enable the installation of the K3S minified Kubernetes distribution to enable the deployment of Kubernetes on top of edge devices.

Automated Deployment of InfluxDB for Monitoring

A TOSCA template has been created [46] to perform the deployment of InfluxDB on top of a Kubernetes cluster via the Infrastructure Manager using the developed Ansible Roles. This will help automate the deployment of the monitoring infrastructure using the Infrastructure Manager (IM). Additionally, support has been included in the IM Dashboard in order to facilitate this deployment using a web browser (see Figure 4.1).

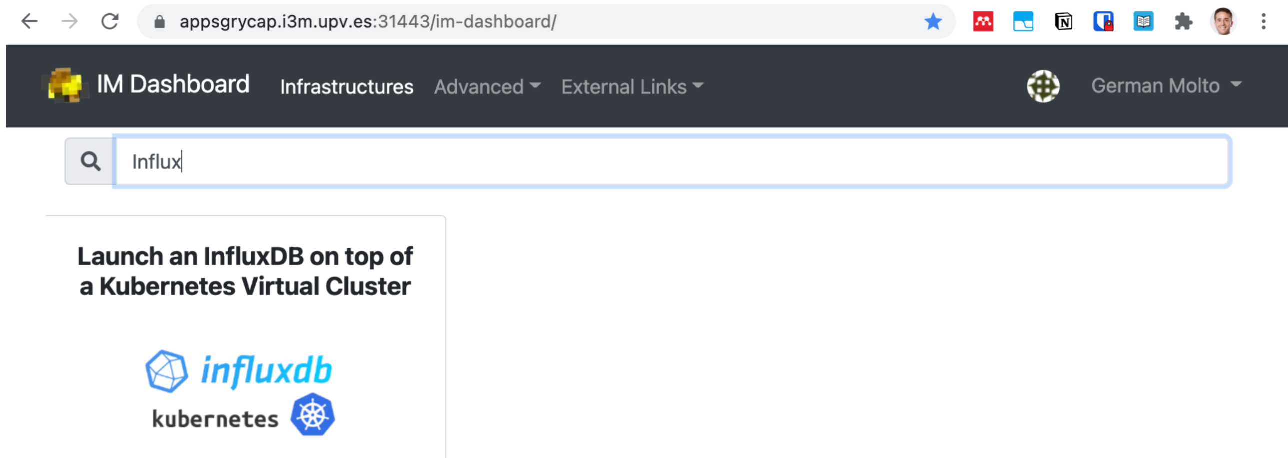


Figure 4.1 - InfluxDB deployment integration in the Infrastructure Manager Dashboard.

SGX improvements

First, SGX support was added to the OpenStack connector of the IM to correctly select the correct available flavors with SGX available. Next, support was introduced to install the SGX plugin in Kubernetes clusters to enable managing SGX resources at K8s level. Finally, support was introduced in CLUES (the internal IM scaling component) to manage the SGX resources that the K8s SGX plugin makes available in order to correctly manage the elasticity of SGX requesting pods.

4.2.1.2 Evaluation

To evaluate the IM, we updated the analysis made in [49]. In this case, a Kubernetes cluster has been selected, since it is the main container-based platform used in the project. This section shows the time needed to deploy a set of small or medium sized Kubernetes clusters on several IaaS Clouds: OpenNebula (ONE), EGI Federated Cloud and Amazon Web Services (AWS). It also shows the ability to manage the elasticity provided by the IM and the time needed for the addition and removal of working nodes of the cluster.

The time needed to deploy the infrastructure can be decomposed into the following steps:

- VMs accessible: Time needed to have the SSH port accessible in the master VM. This step requires the VM to be created and start running and the Operating System (OS) of the VMs to boot and start the SSH server.
- Ansible configured: Time needed to install and configure the Ansible contextualisation software in the master node. This is a relatively simple process that involves downloading the software and a small list of requirements, installing them and copying all the “recipes” needed to configure the infrastructure.
- Fully configured: This process involves the installation of all the needed software packages to install Kubernetes in all the nodes and the whole configuration process. It also applies a set of YAML files to configure a set of basic applications into the Kubernetes cluster (ingress controller, metrics-server, nfs-client-provisioner, …). This process is made simultaneously in all the nodes, so it may cause some bottlenecks in the network or in the disk access. Also, there are some points where some synchronisation is needed among the master and the worker nodes: e.g., the worker nodes cannot join the master node until it is properly initialised.

The node addition test includes the time needed to create the VM, the booting process to have the SSH server active, the configuration of the added node and the reconfiguration of the rest of the nodes. In the rest of the nodes, Ansible playbooks are executed again, but since the Ansible modules are idempotent it

only checks that the current configuration is correct, thus making no changes or just minor ones without any significant overhead.

The node removal test includes the time needed to terminate the VM, and the reconfiguration of the rest of the nodes. As in the previous case, this implies no changes or just minor ones without a significant overhead.

	5 Nodes (ONE)	5 Nodes (EGI)	5 Nodes (AWS)	10 Nodes (ONE)	10 Nodes (AWS)	20 Nodes (ONE)	30 Nodes (ONE)
VMs Accessible	1:06	1:10	1:02	2:16	1:09	2:34	1:50
Ansible Configured	2:01	1:44	1:56	2:00	1:58	2:16	2:27
Fully Configured	7:42	6:44	7:04	10:15	9:16	9:59	14:19
Total Time	10:49	9:38	10:02	14:31	12:23	14:49	18:36
VM Addition	5:38	4:26	6:55	5:54	6:50	7:07	6:18
VM Removal	2:06	2:27	3:15	2:41	4:00	4:31	7:27

Table 4.1 - Deployment time (in minutes) for the Infrastructure Manager to provision a Kubernetes cluster of several sizes in different Clouds.

Table 4.1 shows the time needed in each individual step. The shown times are the average values of three tests performed in each case. As shown in the results, the average time needed to deploy a small or medium sized fully functional Kubernetes cluster is about 10 or 20 minutes and the most time-consuming step is the configuration since it requires to install a relatively large list of packages in every node and perform the Kubernetes configuration. The time required to add a new node is slightly lower since just one node must be totally configured and the other nodes just need to add the new one to the configuration. Finally, the node removal is the quickest operation since terminating a VM can be done very quickly, and it only needs to remove the node from the configuration of the other nodes.

The time needed to add a node to the virtual infrastructure is reasonably long. If deployment time is an issue (e.g., dealing with an unpredicted workload), the configuration process could be reduced by using a pre-configured Virtual Machine Image (VMI) with some (or most) of the software requirements.

This analysis demonstrates improvements in the KPI *K3.1 Back-end resources supported by infrastructure deployment* (see AI-SPRINT deliverable *D7.6 IP Management plan)* by being able to deploy on several IaaS Cloud back-ends. In M12, the value of this KPI is 9 since the IM can deploy in Amazon Web Services, Microsoft Azure, Linode, Google Cloud Platform, EGI Federated Cloud, Orange, Open Telekom Cloud, OpenNebula, OpenStack.

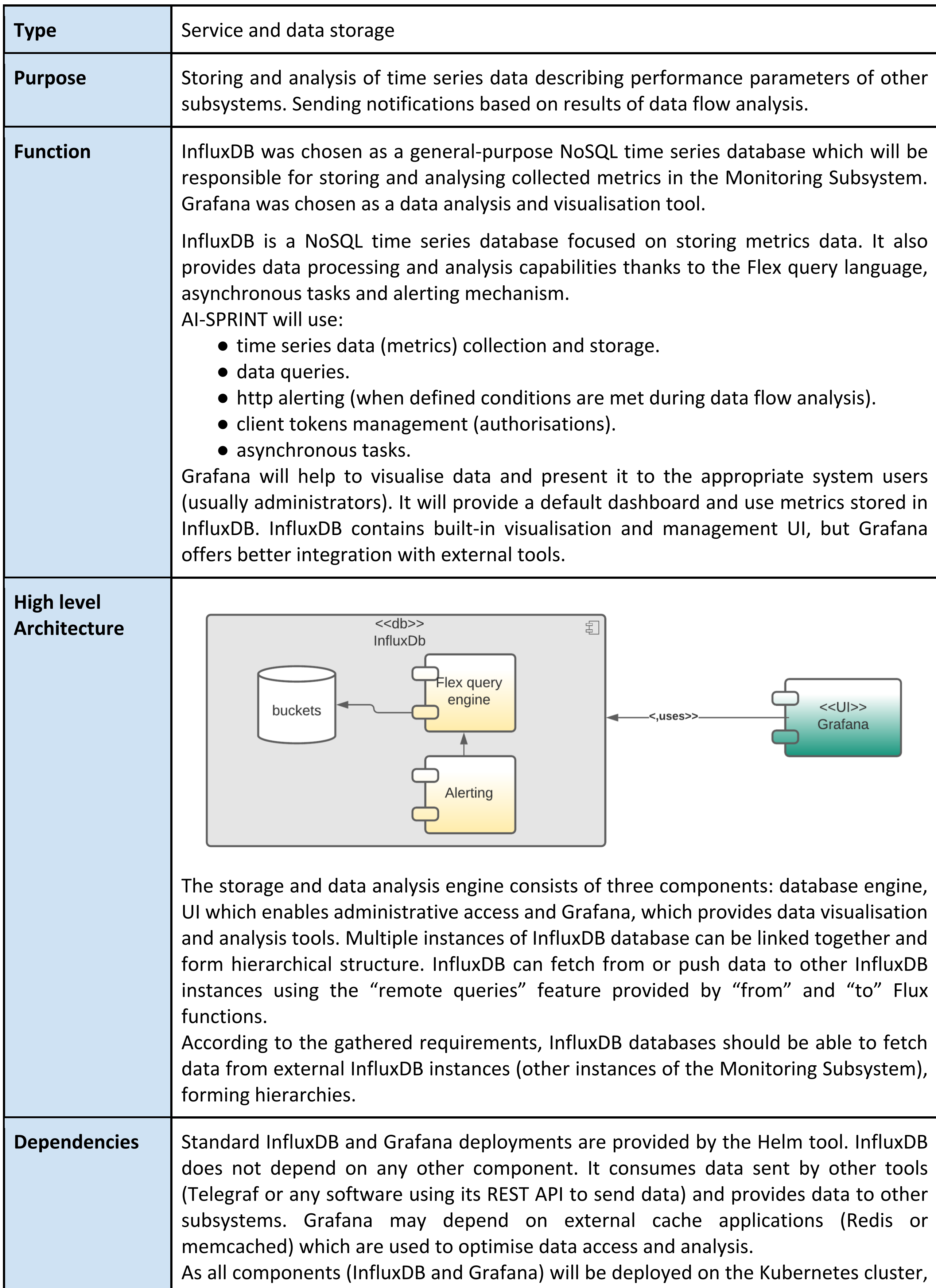
4.3 Monitoring Tools

The monitoring tools are thoroughly covered in *D3.2 First release and evaluation of the monitoring system*. However, since these tools are part of the runtime environment, at least the description tables are included in this section for the sake of reference.

4.3.1 Storage and data analysis engine

These sets of components will be responsible for data (collected metrics) storage and further processing and visualisation.

Identification	Monitoring/Data storage

Type	Service and data storage
Purpose	Storing and analysis of time series data describing performance parameters of other subsystems. Sending notifications based on results of data flow analysis.
Function	InfluxDB was chosen as a general-purpose NoSQL time series database which will be responsible for storing and analysing collected metrics in the Monitoring Subsystem. Grafana was chosen as a data analysis and visualisation tool. InfluxDB is a NoSQL time series database focused on storing metrics data. It also provides data processing and analysis capabilities thanks to the Flex query language, asynchronous tasks and alerting mechanism. AI-SPRINT will use: ● time series data (metrics) collection and storage. ● data queries. ● http alerting (when defined conditions are met during data flow analysis). ● client tokens management (authorisations). ● asynchronous tasks. Grafana will help to visualise data and present it to the appropriate system users (usually administrators). It will provide a default dashboard and use metrics stored in InfluxDB. InfluxDB contains built-in visualisation and management UI, but Grafana offers better integration with external tools.
High level Architecture	 The storage and data analysis engine consists of three components: database engine, UI which enables administrative access and Grafana, which provides data visualisation and analysis tools. Multiple instances of InfluxDB database can be linked together and form hierarchical structure. InfluxDB can fetch from or push data to other InfluxDB instances using the “remote queries” feature provided by “from” and “to” Flux functions. According to the gathered requirements, InfluxDB databases should be able to fetch data from external InfluxDB instances (other instances of the Monitoring Subsystem), forming hierarchies.
Dependencies	Standard InfluxDB and Grafana deployments are provided by the Helm tool. InfluxDB does not depend on any other component. It consumes data sent by other tools (Telegraf or any software using its REST API to send data) and provides data to other subsystems. Grafana may depend on external cache applications (Redis or memcached) which are used to optimise data access and analysis. As all components (InfluxDB and Grafana) will be deployed on the Kubernetes cluster,

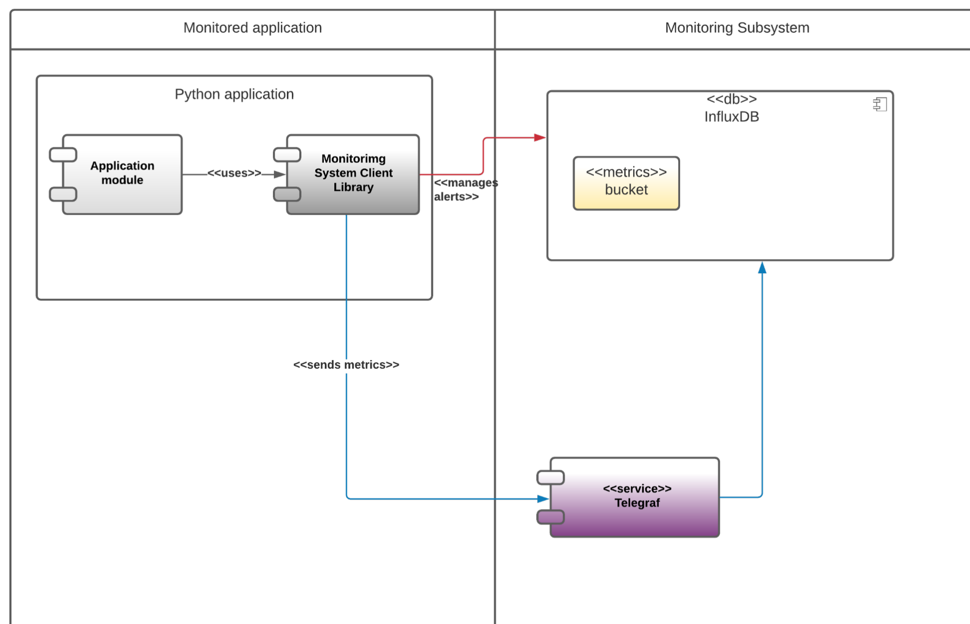
	they depend on the provided cluster functionalities. The deployment process depends on Helm and kubectl tools.
Interfaces	On the lowest level, InfluxDB provides a REST API which can be used to load and fetch collected data (time series) and also to perform analytics and administrative tasks. The API is described in: https://docs.influxdata.com/influxdb/v2.0/api/ Metrics data must be sent as text in "line protocol" format inside REST API call: https://docs.influxdata.com/influxdb/v2.0/reference/syntax/line-protocol/. Query results are returned in annotated CSV format: https://docs.influxdata.com/influxdb/v2.0/reference/syntax/annotated-csv/. InfluxDB also provides a web UI which allows users to do administrative tasks and data analysis. Grafana provides a web UI to create dashboards and present visualised data.
Data	Data is stored internally in "buckets". Buckets are grouped in "organisation". InfluxDB documentation suggests using as few buckets as possible. Each bucket can have a defined "data retention" parameter which describes how long data records will be preserved in that bucket (it can be infinite). Each data row consists of timestamp (nanosecond precision), tags and values. Timestamp and tags provide uniqueness constraints. Grafana is integrated with InfluxDB by a dedicated open-source plugin.
Needed improvement	InfluxDB does not have an "out of the box" mechanism providing hierarchical data connections between databases. This connection must not be fragile to any network issues and should allow synchronisation of data with subsystems which can work off-line.
Implemented Improvements for the First Release	It was needed to create a special periodic task in InfluxDB which will fetch data from dependent (remote) InfluxDB instances. The mechanisms provided in InfluxDB were not able to detect network issues during the data fetching process and it was needed to create an error-proof, proper solution. It was also needed to create scripts and documentation which will help automate InfluxDB and deployment processes on the Kubernetes clusters.
Release Version & Repository	GIT: git@gitlab.polimi.it:ai-sprint/monitoring-poc1.git in "automat" and "test_tasks" directories.

4.3.2 *Data gathering, pre-processing and delivery*

These set of components will be responsible for the first step of metrics gathering and preprocessing flow and data delivery to the storage engine.

Identification	Monitoring/Data delivery
Type	Service and library

Purpose	Components responsible for gathering metrics data, local buffering and sending it to InfluxDB.
Function	Telegraf was chosen as a component responsible for gathering metrics data from system and monitored applications and sending it to the data storage engine. It was also needed to create a Python library for developers, which provided basic client integration with the Monitoring Subsystem. Telegraf fetches or receives monitoring data (time series) in various formats and sends the collected data to InfluxDB. It decouples monitored applications from database and analytics software. It is a universal “metrics consumer” and adapter which buffers gathered data locally and sends it to the InfluxDB in well-defined format. Telegraf also allwos monitoring the whole Kubernetes infrastructure.
High level Architecture	This diagram presents the general architecture of the Monitoring Subsystem: The data gathering modules will be provided by Telegraf installed as “monitoring agents” on each node and (optional) Telegraf services responsible for receiving data from monitored applications. Also, a special Monitoring System Client library will be provided for Python developers. Telegraf is a single executable with a broad set of data input and output plugins. Plugins are configured in the external configuration file. They provide possibilities to fetch data in various formats and send data to various consumers (one of them is InfluxDB database instance). In a K8s cluster, Telegraf can be run as a sidecar container (Telegraf Operator) or as a separate K8s service. The Telegraf sidecar container is added automatically to

	annotated Pods by “telegraf.influxdata.com/*” annotations. In such configuration it provides easy integration of monitored applications with the Monitoring Subsystem. Monitored applications can then provide their metrics in the Prometheus format on a defined TCP port or they can send their metrics directly to a Telegraf instance running on “localhost”. Another solution is a Telegraf as a Service (TaaS) which may be a better solution in case of monitoring short lived jobs which send metrics to a Telegraf instance. 
Dependencies	When run as a separate service, Telegraf only needs access to the InfluxDB bucket where data should be sent and the InfluxDB access token. Telegraf can accept metrics data on a defined UDP port. When run as a sidecar container, Telegraf needs time to be fully initialised. In case of short living jobs, some data sent to Telegraf by a job may be lost because the Telegraf instance may not be ready. As all components (various Telegraf instances) will be deployed on the Kubernetes cluster they depend on provided cluster functionalities. The deployment process depends on Helm and kubectl tools.
Interfaces	Telegraf can receive data from very different sources. It can fetch data (periodic “data scraping”) and receive data sent by its clients. There are many available plugins which provide access to various software products’ metrics (including standard Prometheus data format). In the configuration used in the AI-SPRINT project, Telegraf service will provide a UDP or TCP port (using socket listener plugin) where external systems will be able to send data in one of well-defined data formats (https://docs.influxdata.com/telegraf/v1.20/data_formats/input/). The data format will be adjusted to the client library used in software which will be monitored (in most situations it will be “influx” or “prometheus” data formats). The Telegraf “monitoring agents” will use system statistics provided as files. And it will also fetch metrics from the local (node’s) kubelet instance.

Data	Telegraf only caches data for a short time. Accepted input data formats are described in : https://docs.influxdata.com/telegraf/v1.20/data_formats/input/ Output data format: influxdb line format: https://docs.influxdata.com/influxdb/v2.0/reference/syntax/line-protocol/
Needed improvement	Proper configuration procedures and configuration files must be prepared in order to provide automatic deployment in Kubernetes clusters. A new Python library must be developed in order to help programmers to integrate with the Monitoring Subsystem. It must support sending custom metrics and integrate with new Python decorators (annotations) developed in AI-SPRINT (see Deliverable *D2.1 First release and evaluation of the AI-SPRINT design tools)*. This library should provide automatic alerting functionality.
Implemented Improvements for the First Release	A set of proof-of-concept (PoC) small applications demonstrating possible approaches were created, including a basic client library. Also, documentation and a set of configuration files allowing deployment of Telegraf on the Kubernetes cluster were produced. Finally, a demonstration on how to automatically create alerts and consume alert messages by custom Python application was created.
Release Version & Repository	PoC projects: ● git@gitlab.polimi.it:ai-sprint/monitoring-poc1.git in “test_python” directory. ● git@gitlab.polimi.it:ai-sprint/monitoring-python-alert-receiver-poc-1.git ● git@gitlab.polimi.it:ai-sprint/python-metrics-job-poc1.git (contains 1st version of client library) ● git@gitlab.polimi.it:ai-sprint/python-metrics-job-poc2.git

4.4 Federated Learning

Identification	AI-SPRINT Federated Learning Framework
Type	A module
Purpose	This module allows distributed training among different data sources and different users in a private and secure environment without direct data sharing.
Function	This module allows the distributed training among different peers according to the privacy level to be achieved via an adequate Federated Learning protocol (e.g., traditional or generative, with or without differential privacy constraints, centralised vs. distributed, etc. see [58]). According to the classical federated learning paradigm, each user adheres to the protocol and builds her own local model. Then, the relevant information about the model is shared with the federated party according to the chosen protocol.
High level Architecture	The Federated Learning System is a separate component, and it will be provided via software stubs in the standard deep learning frameworks targeted by the project i.e.,

	PyTorch and TensorFlow. The current demo implementation is based on Python and relies on standard Python libraries and TensorFlow.
Dependencies	The Federated Learning System currently depends on the differential privacy module, on Keras, TensorFlow2, TensorFlow Federated and TensorFlow Privacy frameworks. Future porting to the PyTorch environment will require extending this list of libraries, but the framework will be tested in this version prior to the porting.
Interfaces	The first version (the reference task T3.4 started at M10) will be available via command line interface. From a user perspective, it will be completely hidden. In future versions code stubs for TensorFlow2 and PyTorch will be studied for the user in case a low-level API is required.
Data	The input data includes a list of hyperparameters and the dataset path, for users, and a list of hyperparameters and a queue of users, for the central server. A shared model architecture is available for users and the central server.
Needed improvement	The system is under development and must be currently tested and adapted to each scenario.
Implemented Improvements for the First Release	The current version of the tool has been developed from scratch within the AI-SPRINT project. It is currently a demo version. The first release is expected to be delivered in the second year of the project.
Release Version & Repository	N.A.

In the context of machine learning, having access to huge data amounts is crucial to build highly performing algorithms. Having multiple data producers, e.g., customers and users, the natural solution would be to collect their data into a centralised system and make them collaborate to the training of a central model.

Despite the ideal utility of such an approach, this has several drawbacks and violates many data privacy and confidentiality laws. The main idea at the basis of Federated Learning relies on what should be shared among users and a data center: from a user perspective, instead of sending personal information, that may be sensitive, and receiving inferences and suggestions, each user has her own local model and has to share only updates to receive global models updates [58].

Formally, each user that joins the federated party defines and accepts data, models and tasks policies with a central provider. It is thus possible to build local models and share weight updates, in principle, every time there is a new observation. These updates can be sent to the data center in clear, protected by noise or aggregated according to the different protocols. When this information reaches the data center, it is used to update global models, which in turn produce their updates to be broadcasted to all the party members. In this way, it is possible to share information without sharing data, protecting users and their privacy, and at the same time making local and global models learn and generalise.

Despite the robustness of this paradigm, it has been demonstrated that Federated Learning, as it is commonly intended nowadays, suffers from other security issues rather than learning privacy. In fact, in the last few years, several works have been explaining how it is possible to reconstruct original data from users' updates and how to poison the entire federated party [59, 60, 61]. In detail, most of the attacks provide for the

presence of the attacker among the regular users, which is very probable, especially in huge federated systems.

In order to prevent these threats, novel experimental approaches rely on generative deep learning and differential privacy techniques [55,62]. These techniques aim to preserve the benefits of traditional Federated Learning and guarantee higher security levels against potential attackers.

The idea proposed by some authors is to share data generators instead of weight updates. Each user shall adhere to a policy to define her local model like in traditional Federated Learning, but in this scenario, the local model is a differentially private data generator. Each user will train it on her own data, send it to the data center, and have access to all the generators of the federated party. This technique makes it impossible to reconstruct private data from updates because of the absence of an updates flow. If a generator is poisoned, it can be discarded without affecting any model or any user.

Moreover, each user will be free to exploit the received generators creating an arbitrary number of synthetic samples without violating any privacy constraint. Unlike traditional Federated Learning scenarios, users will not be forced to adhere to any predefined list of tasks.

The first prototype developed within the AI-SPRINT Federated Learning Framework, namely [55], leverages this latter approach and uses generative models to exchange dataset surrogates protected by training the generative models via differential privacy training procedures.

4.5 Scheduling for Accelerated Devices

This component aims to solve the joint resource planning (i.e., how many GPUs to assign to a training job) both for private and public clouds. It includes the GPU scheduler described in section 4.5.2 and rCUDA, described in the following section, to access remote GPUs. The rCUDA suite includes the rCUDA remote GPU virtualisation middleware and the rCUDA scheduler which partitions the available memory in the remote GPUs.

4.5.1 rCUDA

Identification	rCUDA scheduler
Type	A GPU scheduler service to be used along with rCUDA
Purpose	The rCUDA scheduler allows a user to request a set of virtual remote GPUs across the cluster.
Function	Users of GPU services across the cluster use the rCUDA scheduler to specify the remote GPU requirements of the accelerated job (amount of available GPU memory, type of GPUs, etc) and the rCUDA scheduler tries to satisfy the request from the user by providing the set of remote GPUs that best suits the requirements. The rCUDA scheduler provides the set of remote GPUs to be used by the job by providing the exact value of the rCUDA environment variables to be used by the job before it is executed.
High level Architecture	The rCUDA scheduler deploys a small daemon at each of the cluster nodes owning GPUs. These small daemons collect information about the GPUs in the node (available memory, GPU utilisation, etc) and forward that information to the rCUDA scheduler, which is running in one of the nodes of the cluster. In this way, the rCUDA scheduler gets a global view of the status of the GPUs in the cluster and, therefore, can efficiently perform the requested scheduling from the demanding jobs.

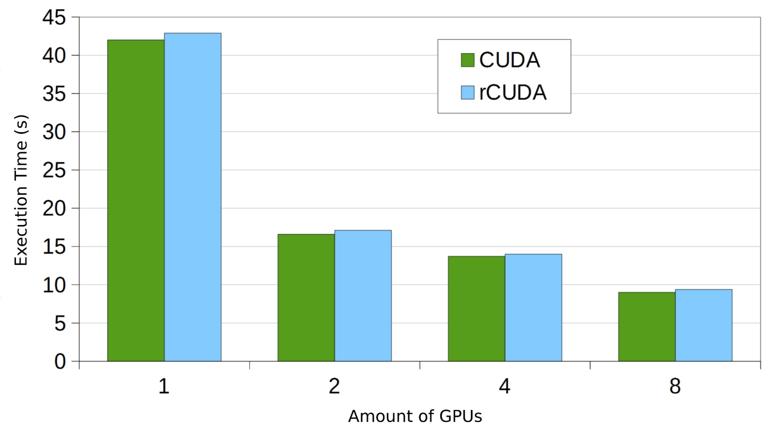
Dependencies	The rCUDA scheduler only depends on the NVML library provided by NVIDIA. This library is used to access the GPUs and collect data about their current state. The rCUDA scheduler is independent from the rCUDA remote GPU virtualisation middleware. Both components are connected by the output from the rCUDA scheduler, consisting of the environment variables that the job must use when executed to use the rCUDA middleware.
Interfaces	The current interface to the rCUDA scheduler is based on the *srun* and *sbatch* commands of the SLURM job scheduler. The rCUDA scheduler provides its version of these commands, which separate CPU requirements from GPU requirements and forward GPU requirements to the rCUDA scheduler. Accessing the rCUDA scheduler is done by connecting to the TCP port used by the scheduler. The scheduler provides the result from the scheduling across this port. Communication among the rCUDA scheduler and the demons deployed across the cluster is also carried out by using TCP-based communications.
Data	The rCUDA scheduler internally stores the status of the GPUs of the cluster in a table where available GPU memory, GPU utilisation, etc. is stored for each of the GPUs provided to the scheduler.
Needed improvement	Within AI-SPRINT, the main action to take regarding the rCUDA scheduler is adapting its interface so that integration with the rest of the system is optimal. In this regard, the current interface of the rCUDA GPU scheduler is oriented to the Slurm scheduler. However, this interface might not be the best one in the context of the AI-SPRINT project. For that, the rCUDA GPU scheduler will incorporate the interface required in AI-SPRINT. Also, as the rCUDA scheduler was only used in the cluster of the rCUDA developers, some bugs may arise because of moving the scheduler to another scenario.
Implemented Improvements for the First Release	rCUDA has been adapted to the reference deployment considered within the project based on Docker containers. It was necessary to address some issues with the virtualisation layer provided by Docker regarding parameters internally used by rCUDA to support multithreaded applications. Moreover, rCUDA has been adapted to properly execute the TensorFlow benchmarks used in the project (a-GPUBench, see AI-SPRINT deliverable D2.1 Appendix D) by fixing some bugs that raised up when executing those benchmarks: (1) some CUDA functions used by TensorFlow in these benchmarks were not virtualised yet in rCUDA, and (2) some race conditions in memory updates from server to client appeared inside rCUDA when executing these benchmarks. On the other hand, rCUDA was adapted to a more modern version of CUDA (CUDA 10.0). Also, TensorFlow had to be recompiled with CUDA 10.0 dynamically. Finally, a bug was fixed in the rCUDA scheduler consisting of increasing the stability of the scheduler.
Release Version & Repository	The binaries of the rCUDA scheduler are available upon request to the rCUDA developers. The source code of the rCUDA scheduler is not available.

4.5.1.1 Detailed Description of Activities for the First Release

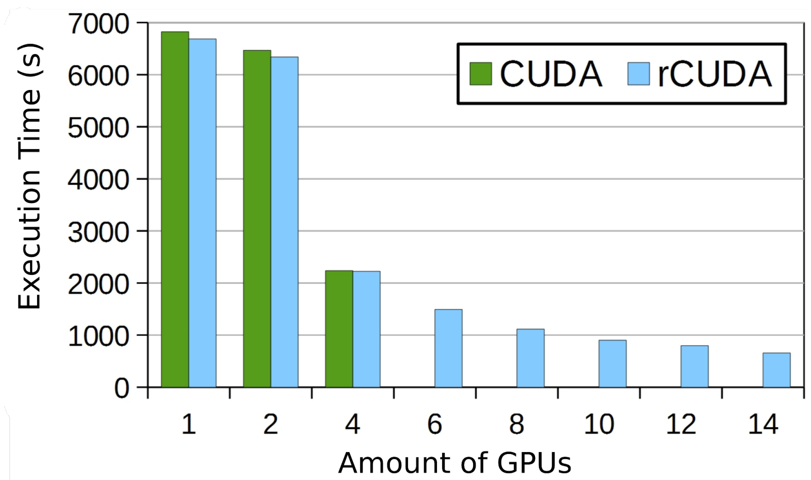
In addition, to trigger the usage of the rCUDA middleware within dockers and TensorFlow, some initial testing has been carried out in the context of the AI-SPRINT project to get familiar with the rCUDA scheduler.

4.5.1.2 Evaluation

Some initial performance evaluation has been carried out with rCUDA. First, the LAMMPS [47] application, which is a particle dynamics simulator, has been used. In this case we use a small cluster made up of 8 nodes, each of them equipped with one GPU. Note that this is one of the possible configurations used today in data centers with accelerators, in which all nodes have their own GPU. In this example, we first run the LAMMPS simulator in a distributed manner across the 8 nodes, so that we have 8 processes, one per node, and each process uses the GPU of the node where it is running (there is one GPU per node). This would be the usual configuration. In a second stage, we setup the system in a way the 8 processes remotely access the 8 GPUs through the interconnection network, based on the InfiniBand technology. In this case, the 8 processes are running on the same node, using 8 cores. If we compare both execution times (local GPUs versus remote GPUs) we observe that the use of rCUDA increases the execution time by only 4%. However, this small increase in execution time buys significant benefits. On the one hand, it provides the flexibility for applications to run on any node in the cluster, whether GPUs are installed. On the other hand, given that GPUs cannot be safely shared when CUDA is used because it does not provide a safe way to share a given GPU, only one process can safely own the GPU. This makes that in cluster nodes where a single GPU is installed, only one process can use the GPU. Therefore, for a set of jobs using the GPU, this translates into an under usage of the CPU cores because only one of the jobs can be scheduled for execution at a time. With rCUDA it is possible to safely share a given GPU and thus is it possible to use all the cores of the node.

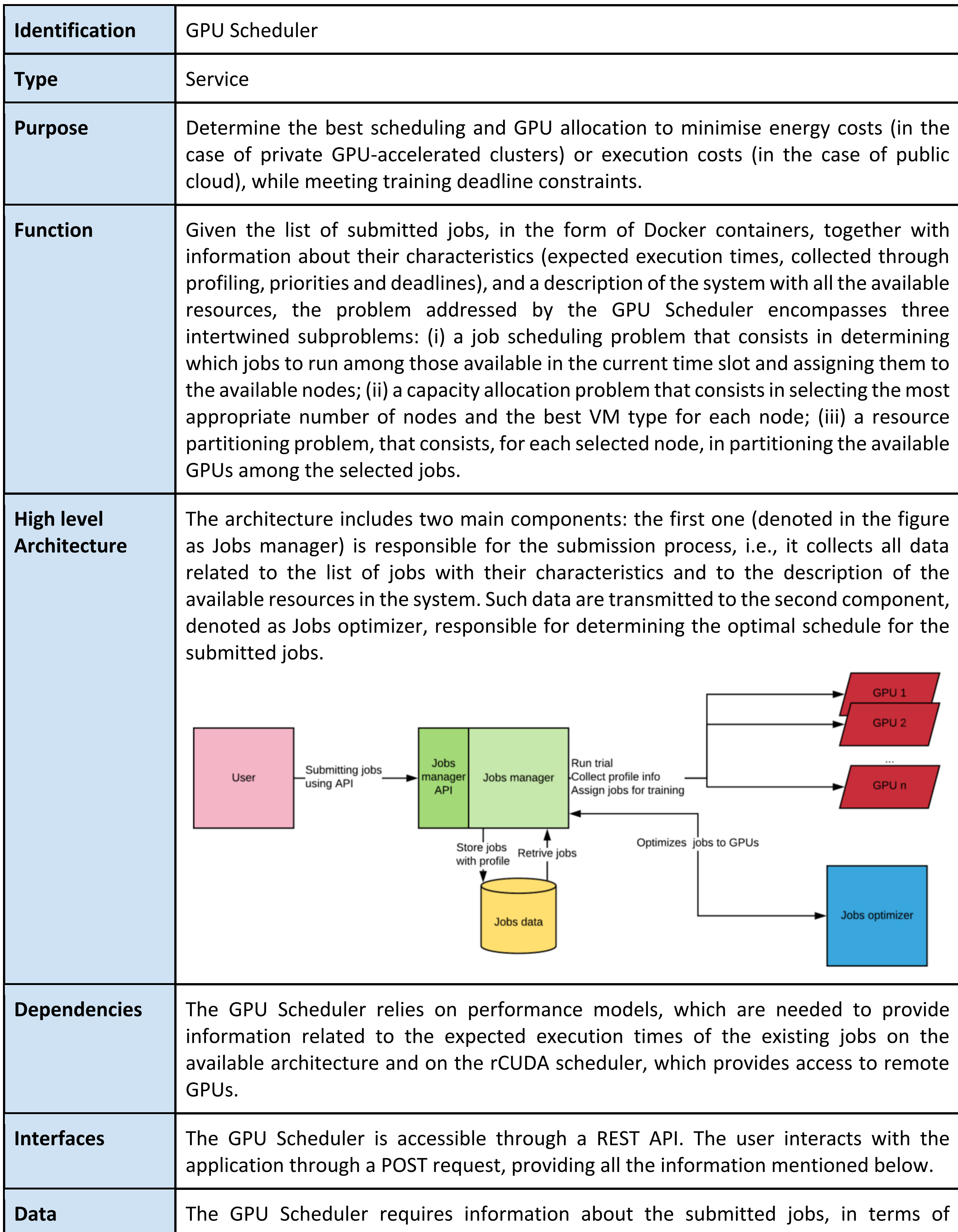


Another performance evaluation used the MontecarloMultiGPU NVIDIA Sample [48]. This sample performs a Monte Carlo simulation to calculate the price of shares on the stock market. In this case, we provide multiple GPUs to the same application, beyond those that typically physically fit in a single node. In this case, using local GPUs with CUDA we could provide the application with up to 4 GPUs, reducing the execution time to 31% of the time that the application would need with a single GPU. However, if rCUDA technology is used to provide the application with many more GPUs, its execution time is reduced to 9% of the initial time with CUDA locally, when assigned 14 remote GPUs with rCUDA.



4.5.2 GPU Scheduler

<table>
<tr><td>Identification</td><td>GPU Scheduler</td></tr>
<tr><td>Type</td><td>Service</td></tr>
<tr><td>Purpose</td><td>Determine the best scheduling and GPU allocation to minimise energy costs (in the case of private GPU-accelerated clusters) or execution costs (in the case of public cloud), while meeting training deadline constraints.</td></tr>
<tr><td>Function</td><td>Given the list of submitted jobs, in the form of Docker containers, together with information about their characteristics (expected execution times, collected through profiling, priorities and deadlines), and a description of the system with all the available resources, the problem addressed by the GPU Scheduler encompasses three intertwined subproblems: (i) a job scheduling problem that consists in determining which jobs to run among those available in the current time slot and assigning them to the available nodes; (ii) a capacity allocation problem that consists in selecting the most appropriate number of nodes and the best VM type for each node; (iii) a resource partitioning problem, that consists, for each selected node, in partitioning the available GPUs among the selected jobs.</td></tr>
<tr><td>High level Architecture</td><td>The architecture includes two main components: the first one (denoted in the figure as Jobs manager) is responsible for the submission process, i.e., it collects all data related to the list of jobs with their characteristics and to the description of the available resources in the system. Such data are transmitted to the second component, denoted as Jobs optimizer, responsible for determining the optimal schedule for the submitted jobs.


</td></tr>
<tr><td>Dependencies</td><td>The GPU Scheduler relies on performance models, which are needed to provide information related to the expected execution times of the existing jobs on the available architecture and on the rCUDA scheduler, which provides access to remote GPUs.</td></tr>
<tr><td>Interfaces</td><td>The GPU Scheduler is accessible through a REST API. The user interacts with the application through a POST request, providing all the information mentioned below.</td></tr>
<tr><td>Data</td><td>The GPU Scheduler requires information about the submitted jobs, in terms of</td></tr>
</table>

	expected execution times (collected through profiling and performance models), deadlines and priorities. Moreover, it requires a description of the available resources.
Needed improvement	The integration with rCUDA, that provides access to remote GPUs, is ongoing as well as the implementation of the submission system.
Implemented Improvements for the First Release	The optimizer was implemented, providing three different heuristic methods for determining a schedule, guaranteeing different levels of optimality in terms of costs and scalability performance.
Release Version & Repository	The first release version of the GPU Scheduler for AI-SPRINT can be downloaded from: https://gitlab.polimi.it/ai-sprint/GPUScheduler

4.5.2.1 Detailed Description of Activities for the First Release

Formally, the reference system considered by the GPU Scheduler includes a set $N = \{n_1, n_2, ..., n_N\}$ of job processing units (referred to as nodes); each node can be configured with a Virtual Machine (VM) of type v taken from a cloud provider's catalogue denoted by the set V, characterised by a specific type and number of available GPUs, other than by a time-unit cost c_v. The number of GPUs available on each VM is identified by a set $G_v = \{1, 2, ..., G_v\}$. The set of submitted jobs (which includes the jobs in execution and in the waiting queue) at a given time instant is denoted by J. Each job is characterised by its release time, its deadline d_j and by an estimated processing time t_{jvg} that depends on the VM type v and the number g of GPUs assigned to the job. Jobs are never rejected, but they can be preempted, and their execution can be postponed to a following time period if a job with higher priority is released and the available resources are insufficient to run both (job preemption is allowed since the training code periodically saves some snapshots of the DNN weights and of the training optimizer if it is stateful). The priority of a job depends on the remaining processing time, the due date and the associated tardiness penalty. The tardiness of job $j \in J$ is denoted by τ_j whereas a weight ω_j is used to compute the tardiness penalty, $\omega_j\tau_j$.

The job selection and resource allocation processes aim at executing all jobs in J before their due date and, at the same time, to maximise the utilisation of the selected nodes avoiding idle resources, thus reducing the overall cost, which is given, as reported in Equation (1), by the sum of the execution costs of all jobs and of the tardiness cost of jobs whose deadline is violated. Since this penalty term is always more significant than the execution cost per time unit, the system tends to avoid delays as much as possible, according to the available resources.

$$f_{OBJ} = \sum_{j \in J}\left(\omega_j\tau_j + \rho\omega_j\hat{\tau}_j\right) + \sum_{j \in J, n \in N} \alpha_j\pi_j \qquad (1)$$

To solve this joint Capacity Allocation and Job Scheduling problem, we have developed a heuristic algorithm based on randomised greedy and path relinking strategies [37]. The proposed method is based on the following assumptions:

- Jobs should be scheduled according to their pressure, which measures how close they are to the deadline when executed with the fastest configuration (i.e., a given VM type and number of GPUs).
- The optimal configuration $(v,g) \in V \times G_v$ for each selected job is:
  - the cheapest configuration such that the job is executed before its deadline, if such a configuration exists,
  - the fastest available configuration if, independently from the selected setup, it is not possible to execute the job before its deadline.

This latter assumption is justified by the fact that, in our scenario, the time unit deployment cost of a job on any available configuration is always lower than the penalty incurred if the job deadline is violated. Therefore, it is reasonable to choose the cheapest configuration as long as the deadline can be met, while, as soon as it is violated, it is more convenient to complete the job execution as fast as possible in order to reduce the corresponding penalty.

- Deployment costs increase linearly in the number of GPUs, as demonstrated by Cloud providers pricing models [38, 39].
- The speedup of job execution is sublinear in the number of GPUs, as observed in GPU-based application benchmarks [40].

The proposed method includes three main phases: in the *preprocessing* step, candidate jobs are sorted according to their pressure. Then, the *scheduling* step implements a randomised construction procedure, followed by a step of path relinking, where the best available configuration is selected for all the jobs in the queue. Finally, the *postprocessing* step aims at reducing the amount of idle resources by modifying or reallocating the configurations selected on the different nodes. The overall costs of the proposed solutions are computed according to the formula reported in Equation (1), denoted as $f_{OBJ.}$, while all choices made by the algorithm across in each step are evaluated according to a proxy function f_P. Details about f_P are provided at the end of this section. The scheduling step consists of two sub-steps that are executed sequentially: the randomised greedy construction procedure builds a set *S** of good-quality candidate solutions by optimizing a proxy function f_P. The path relinking procedure starts from the best candidate solution in *S** and improve it by iteratively identifying and combining features of the other candidates.

The algorithm of the randomised construction procedure is reported in Figure 4.2: at each iteration, a new candidate solution S is built through a randomised greedy method. If it has a better f_P value than any other solution currently stored in *S**, the set is updated, possibly removing the worst-valued solution to keep its total number of elements under a fixed value σ. Each candidate solution S ∈ *S** represents a data structure carrying information about the running jobs and the relative configuration. Specifically, each element of S is a tuple (j,n,v,g), where j is the current job and n,v, g are the node, VM type, and number of GPUs assigned to it, respectively.

Algorithm 1 Randomized construction procedure

```
1: function RANDOMIZED_CONSTRUCTION(𝒥, MaxIt_RG)
2:     iter = 0
3:     𝒮* ← ∅
4:     while iter < MaxIt_RG do          ▷ MaxIt_RG: maximum number
                                          of random iterations
5:         S ← empty schedule                      ▷ S: current schedule
6:         𝒥_s ← SORT_JOBS_LIST(𝒥, Δ)    ▷ Δ : pressures of all jobs
7:         for all j ∈ 𝒥_s do                      ▷ 𝒥_s : sorted queue
8:             D*_j = {(v, g) s.t. t_jvg + T_c < d_j}
9:             (v*, g*) ← SELECT_BEST_CONFIGURATION(j, D*_j)
10:            assigned ← ASSIGN_TO_EXISTING_NODE(j, (v*, g*), 𝒩_O)
11:            if not assigned then
12:                if |𝒩_O| < N then
13:                    Select ν' with VM type v* and G_v* GPUs
14:                    𝒩_O ← 𝒩_O ∪ {ν'}
15:                    S ← S ∪ (j, ν', v*, g*)
16:                else
17:                    ASSIGN_TO_SUBOPTIMAL(j, S, 𝒩_O)
18:                end if
19:            else
20:                S ← S ∪ (j, ν*, v*, g*)
21:            end if
22:            𝒥_s ← 𝒥_s \ {j}
23:        end for
24:        if f_P(S) is better than f_P(S') for any S' ∈ 𝒮* then
25:            𝒮* ← 𝒮* ∪ {S}
26:            𝒮* ← 𝒮* \ {S'} if |𝒮*| > σ
27:        end if
28:        iter ← iter + 1
29:    end while
30:    return 𝒮*
31: end function
```

Figure 4.2 - GPU Scheduler: Randomised construction procedure.

The path relinking procedure, whose algorithm is reported in Figure 4.3, receives as input the set of solutions *S** (*elite solutions*) returned by the randomised construction procedure. It extracts from it the candidate solution S_s with best f_P value, generates and explores paths connecting S_s to the other elite solutions in *S**, to find better solutions by combining features of good-quality candidates. The procedure is based on the concept of move, which denotes any atomic change that can be performed to move from the source solution S_s in the direction of a target solution $S_t \in S^*$ different from S_s. The new candidate solution obtained by applying a move m to the current solution S_s is denoted as $S_s \circ m$. In our context, a move from the source S_s to the target S_t is defined as any pair (j,n) such that job j is assigned to node n in S_t, while it is assigned to a different node n' in S_s. In the algorithm, a sequence of moves is performed, either until S_t is reached or until a maximum number of moves is performed.

As reported in the next section, we have evaluated the performance and the costs obtained by exploiting the GPU Scheduler in three different variants, obtained by enabling or disabling different aspects of the scheduling step. Specifically, we denote as Path Relinking method the one whose scheduling process consists, as described above, in the randomised construction procedure coupled with the path relinking procedure. We obtained a first variant, denoted as Random Greedy, by disabling the path relinking procedure and considering only the randomised construction. Finally, we further simplified the process, and obtained the Greedy algorithm, by disabling all randomisations in the construction procedure and taking all choices in a purely greedy manner. Both the Random Greedy and the Greedy variants adopt as proxy function f_P the objective function f_{OBJ} reported as Equation (1). The Path Relinking method, instead, adopts as function f_P a different function f_{OBJ-PR} so as to favour the schedules that use the resources most efficiently, pursuing a trade-off between maximising the number of completed jobs and minimising the operational costs. Such f_{OBJ-PR} is defined as:

$$f_{OBJ-PR} = \sum_{j \in J} \frac{M_j^t}{\pi_j + \omega_j \tau_j} \qquad (2)$$

where M^t_j denotes the maximum processing time of job j on the slowest configuration and π_j denotes the execution cost of the job itself.

Algorithm 2 Path relinking procedure

```
1:  function PATH_RELINKING(S*, MAXIT_PR)
2:      S_s ← solution in S* with best f_P(S_s)      ▷ S_s: source solution
3:      for all S_t ∈ S*, S_t ≠ S_s do                ▷ S_t: target solution
4:          iter = 0
5:          while S_s ≠ S_t and iter < MaxIt_PR do
6:              M ← GET_MOVES(S_s, S_t)
7:              M_E ← ∅                                ▷ M_E: set of explored moves
8:              (m*, c*) ← (empty move, f_P(S_s))
9:              for all m ∈ M do
10:                 if m ∉ M_E then
11:                     (m*, c*) ← EXPLORE_STEP(S_s, S_t, m)
12:                     M_E ← M_E ∪ {m}
13:                 end if
14:             end for
15:             if m* is not empty then
16:                 S_s ← (S_s ∘ m*)
17:             end if
18:             iter ← iter +1
19:         end while
20:     end for
21:     return (S_s, f_P(S_s))
22: end function
```

Figure 4.3 - GPU Scheduler: Path relinking procedure.

4.5.2.2 Evaluation

Results in a simulated environment

The GPU Scheduler has been evaluated first of all in a prototype environment, designing a set of simulations that allowed us to measure the obtained performance in terms of execution costs and scalability. The experimental scenarios have been generated randomly, considering heterogeneous neural networks training tasks for image and speech recognition (i.e., Alexnet, Resnet, VGG, and DeepSpeech), implemented with different DL frameworks (i.e., PyTorch and Tensorflow). Such training jobs were characterised by varying epochs and batch sizes and, therefore, different expected execution times. We considered scenarios with increasing size, characterised by a number of available nodes $|N|$ varying between 10 and 100 and a number of submitted jobs $|J| = 10|N|$. Each node could be configured with a Virtual Machine selected from a catalogue including 8 different types: six of them (NC6, NC12, NC24, NV6, NV12, NV24) are based on Nvidia K80 and M60 and are available on Microsoft Azure. The remaining two (NC48* and NV48*) are hypothetical VM types obtained from the NC24 and NV24, doubling the number of available GPUs and their hourly costs, in line with the current cloud providers pricing models. Jobs inter-arrival times have been generated, in a first scenario denoted as *exponential*, according to an exponential distribution with mean equal to 75,000s / $|N|$. Moreover, following the approach proposed in [41], we have considered three alternative settings, denoted as *high*, *low*, and *mixed*, respectively, where arrivals were sampled from a Poisson distribution, considering three possible rates. The *high* rate is set to $\boldsymbol{\varepsilon} k_{max} \boldsymbol{\lambda}$ and the *low* rate is $\boldsymbol{\varepsilon} k_{max} \boldsymbol{\lambda} / 4$, where $\boldsymbol{\lambda}$ is a base rate defined as the reciprocal of the minimum expected completion time given the configurations available in the catalogue, k_{max} is the number of nodes in the system multiplied by the maximum number of GPUs that can be assigned to each job, and $\boldsymbol{\varepsilon}$ was tuned to match the peak load of the system to real-life scenarios, of nearly 135 job submissions per hour in a system involving few thousands of GPUs [42]. Finally, the *mixed* rate is obtained by alternating *high* and *low* distributions approximately every 10 submissions. For each value of cluster size and each arrival rate, we generated three problem instances, and performed the experiments for each instance with 10 different random seeds, for a total number of nearly 700 tests.

Note that the operational costs depending on the chosen VMs and the penalty costs for deadline violations are evaluated at the end of the simulation, when all jobs have been completely executed. The results obtained by the GPU Scheduler, both implementing the Path Relinking algorithm and two simplified versions denoted as Random Greedy and Greedy algorithms, respectively, are reported in Figure 4.4. They are

compared with the outcomes of the Earliest Deadline First method, which is a first-principle method often used as benchmark and adopted in industrial schedulers, with a Hierarchical Method proposed in [43] and with Dynamic Programming-based methods adapted from [41].

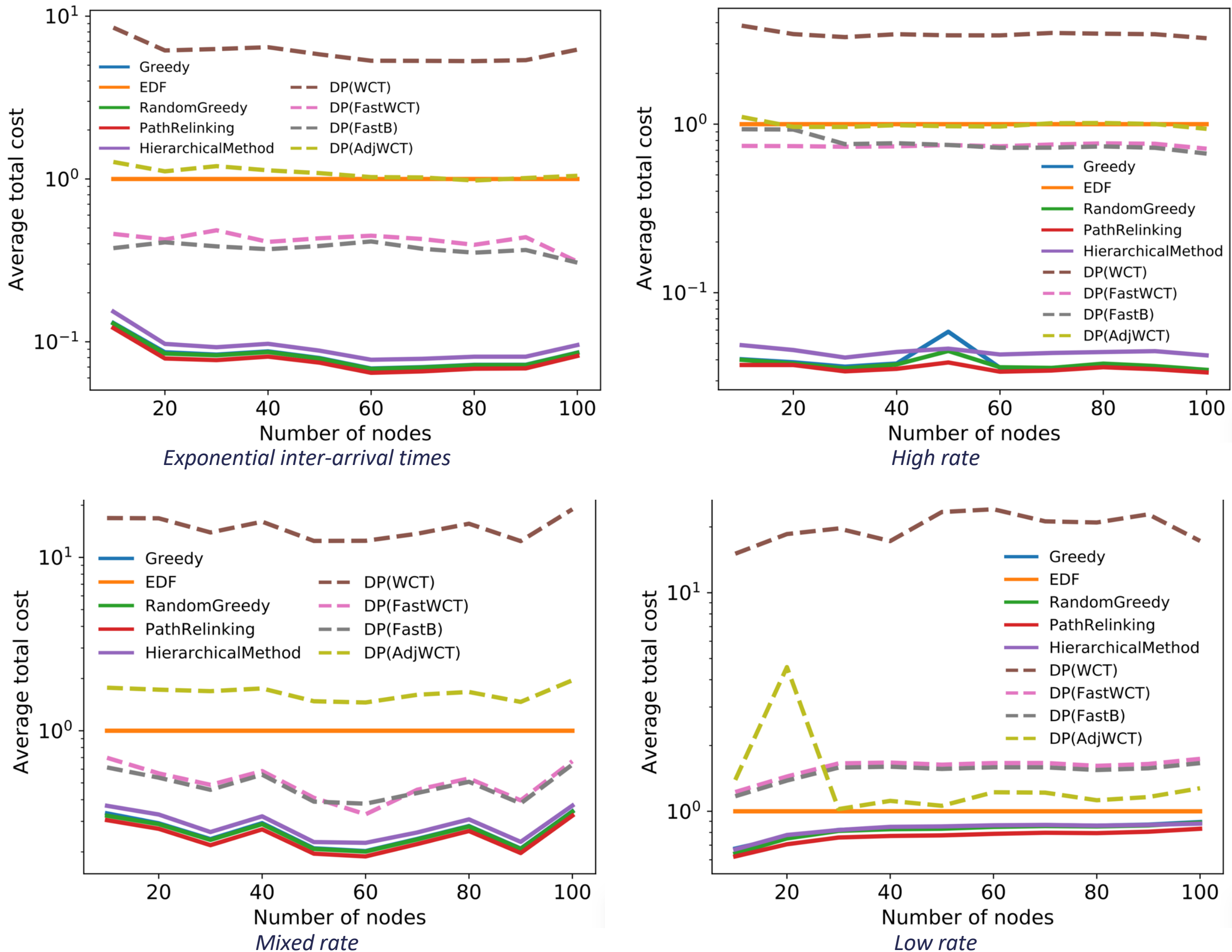


Figure 4.4 - GPU Scheduler results: Comparison among the total costs obtained by the proposed algorithms under different workload scenarios.

The average percentage gain obtained by the GPU Scheduler variants over the EDF method and the other methods considered for comparison is reported in Figure 4.5. First of all, the Greedy algorithm, which is the simplest among the GPU Scheduler variants, is compared against EDF. Then, we show the average percentage cost reduction obtained by exploiting more sophisticated algorithms, comparing the Randomised Greedy method against Greedy and the Path Relinking method against Greedy and Randomised Greedy. Finally, we compare the Path Relinking method with the Hierarchical Method and the four Dynamic Programming-based algorithms. It is worth noting that the Path Relinking delivers the lowest costs in all analysed scenarios. In particular, it achieves a significant cost reduction on EDF, between 23% and 97% on average, and an average percentage gain between 7 and 20% compared with the Hierarchical Method, and between 43 and 95% against the best among the Dynamic Programming-based versions (DP(AdjWCT) in the *low* rate scenario, DP(FastB) in all the others). The fact that, in the *low* rate scenario, all GPU Scheduler variants obtain a less significant cost reduction when compared to EDF (of about 23% on average for Path Relinking) is motivated by the fact that, in this context, the system load is reduced; consequently, it is easier to meet the due dates

even with simple algorithms. The results achieved by relying on simulation during the first year are then already inline with the KPI we defined for the GPU Scheduler (costs savings with respect to first principle methods, i.e., earliest deadline first, first in first out, priority scheduling >= 20%, see AI-SPRINT deliverable *D7.6 IP Management plan*).

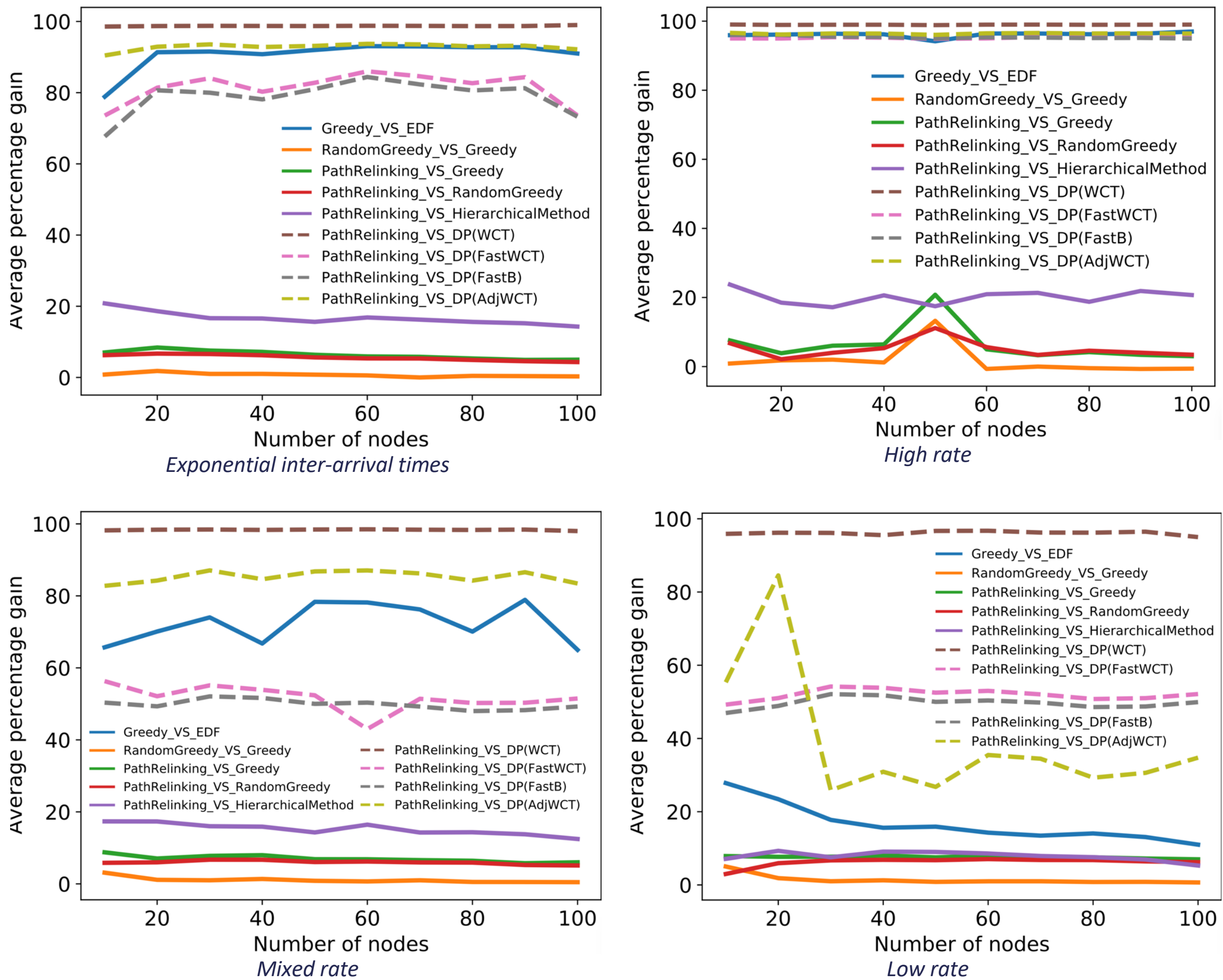


Figure 4.5 - GPU Scheduler results: Comparison among the average percentage gain obtained by the proposed algorithms under different workload scenarios.

Results in a prototype environment

To validate the results obtained by the GPU Scheduler exploiting the Path Relinking algorithm, we deployed on Microsoft Azure a prototype system including six different applications, continuously submitted with an inter-arrival time of 300 s. The considered scenario is accelerated, in terms of submission frequency and average execution times of applications, to limit cloud operations costs. This does not affect the solution effectiveness since the workload assigned to the available nodes is comparable to practical situations.

The results provided by the Path Relinking algorithm are reported in Figure 4.6. The schedule obtained at each step has been implemented and run on a real system to compare the expected performance with the actual one. A discrepancy in execution and completion times of jobs, lower than 5%, is observed because the time required to deploy and boot the VMs on the system and the time required to migrate applications from one VM instance to another are not negligible due to the accelerated time setting of the experiment.

Figure 4.6 - GPU Scheduler results in a prototype environment.

4.6 Privacy Preserving Continuous Training

Identification	Differential Privacy Component
Type	Subprogram
Purpose	Guarantee strict confidentiality levels for individual users in a machine learning context.
Function	The Differential Privacy Component has to mitigate the amount of information that can be extracted from a machine learning model via noise injection at training time. To achieve this, the threat level of deep learning models has to be defined by measuring, layer by layer, the privacy level and the amount of information leaked in case of an attack (e.g., membership inference and model inversion attacks [64], [65]).
High level Architecture	The Differential Privacy Component is a separate functionality. The current implementation is based on Python and relies on standard Python libraries. It aims to quantify the differential privacy of network layers by means of layer level differential privacy attacks. By this quantification it is possible to provide components such as the Neural Architecture Search engine and the Federated Learning Framework insights on the differential privacy level of some architectures (NAS) or training algorithms (Federated learning).
Dependencies	The Differential Privacy Component will rely on TensorFlow2, Keras and Tensorflow Privacy.
Interfaces	The software is currently accessible via command line interface.
Data	The input data includes hyperparameters to set the required privacy level, and the path to the target model to train/analyse.
Needed improvement	The first implementation, which is still under development (the reference task T3.4 started at M10), will include the following features:

	• A function to compute the differential privacy level for deep learning models • A function to inject differential privacy in deep learning models via DP-Stochastic Gradient Descent • A function to compute the threat of model inversion and membership inference attacks for each layer of deep learning models • A function to compute the privacy guarantees for each layer of a deep learning model • A function to increase privacy and security levels at layer level
Implemented Improvements for the First Release	The first version of the tool will be developed from scratch within the AI-SPRINT environment. The core functionalities are actually under development.
Release Version & Repository	N.A.

Privacy plays a fundamental role in an increasingly connected society, especially while talking about data privacy. From Wikipedia definition, “Privacy is the ability of an individual or group to seclude themselves or information about themselves, and thereby express themselves selectively.” Furthermore, privacy must also be taken into account in automatic training scenarios. In these cases, it is possible to exploit private information related to the individual or predict behaviours or actions of single users. In order to avoid these unwanted scenarios, it is necessary to adopt strict, modern and severe countermeasures.

In the last decade, the concept of differential privacy has spread in the machine learning world intending to guarantee users data safety. From Wikipedia definition “Differential privacy (DP) is a system for publicly sharing information about a dataset by describing the patterns of groups within the dataset while withholding information about individuals in the dataset.”

Deep learning models are based on data and related information. Their goal is to learn patterns and correlations to solve a task in the best possible way. In addition to the possible malicious uses of these algorithms, it has been proven that these models contain information related to the learned data, even if in a non-in-clear representation.

In order to cope with this issue, Abadi *et al.* [51] aimed at improving and measuring differential privacy by introducing Gaussian noise during different stages of neural networks training. In information theory, differential privacy [52, 53] can be considered as a privacy model providing strong guarantees against the disclosure of sensitive information related to individual samples.

The most adopted and considered model is (ε, δ)-differential privacy, i.e., the version proposed by Dwork *et al.* [52]. Differential privacy is defined as a randomised mechanism $M : D \rightarrow R$ with domain D and range R which is satisfied if for any two adjacent inputs $d, d' \in D$ and for any subset of outputs $S \subseteq R$ the following condition holds:

$$\boldsymbol{Pr}[\boldsymbol{M}(\boldsymbol{d}) \in \boldsymbol{S}] \leq \boldsymbol{exp}(\boldsymbol{\epsilon}) \ \boldsymbol{Pr}[\boldsymbol{M}(\boldsymbol{d}') \in \boldsymbol{S}] + \boldsymbol{\delta}$$

having that ε is the privacy budget with the goal of balancing the Accuracy of the mechanism with sensitive disclosures. The δ parameter, instead, relaxes the constraints of ε-DP introduced in [54] by allowing violations with probability δ.

Thanks to its properties of composability, robustness to post-processing, and degradation in presence of highly correlated data, differential privacy offers solid guarantees for the privacy of the individual. Its formulation has to be intended as a measure of the privacy level of a query (M): the maximum distance between the same query on a database (d) and a parallel dataset (d'), so that $||d - d'|| \leq 1$, is equal to ϵ. δ represents the accepted leak probability and must be a value smaller than $1/||d||$.

In the deep learning field, many approaches have been proposed to integrate differential privacy guarantees within training and inferential procedures. In order to protect training data sensitivity, Abadi *et al.* [51] designed a Differentially-Private version of the Stochastic Gradient Descent algorithm, i.e., the DP-SGD, for training deep neural networks with non-convex objectives.

In the last few years, many authors have introduced new techniques to exploit differential privacy in generative deep learning [55, 56, 57]. Empowering generators with differential privacy properties has been demonstrated to bring to an even higher level of security for the individual, making data sharing more feasible even in scenarios characterised by a high level of confidentiality. These techniques find their natural application in Federated Learning scenarios as proposed by the preliminary work of the AI-SPRINT consortium [55] that will be further analysed and improved within the project.

4.7 Programming Framework Runtime

These components support concurrent code execution, automatically detecting and enforcing the data dependencies among components and spawning parallel tasks to the available resources, which can be nodes in an edge cluster or cloud, including their execution under the FaaS model along the computing continuum.

4.7.1 COMPSs / PyCOMPSs

Identification	COMP Superscalar (COMPSs)
Type	Programming model, supporting runtime engine and algorithm libraries
Purpose	COMPSs/PyCOMPSs and disLib aim to ease the development of AI applications targeting the Cloud-Edge-IoT Continuum. COMPSs is a task-based programming model that orchestrates the execution of such tasks in a serverless manner on top of any distributed platform. PyCOMPSs is an enhanced version of the programming model exploiting the benefits of the Python programming language. disLib provides application developers with a set of built-in AI algorithms that leverages on PyCOMPSs to distribute the computation.
Function	Service developers code their application's logic using the methods provided by the disLib library or directly following the COMPSs programming model. The COMPSs runtime will receive an external request to compute something coming either from a manual petition by an end-user or automatically triggered as a response to a change on the data or the infrastructure related to the service. Upon the reception of such a request, the runtime engine divides the application into several tasks, detects the data dependencies among them and orchestrates the execution of such tasks across all the resources within the infrastructure aiming to achieve a shorter execution time and an efficient usage of the nodes belonging to the underlying infrastructure.
High level Architecture	The following image depicts the architecture of the programming model runtime. Each node belonging to the infrastructure runs a daemon process, known as Agent, that handles the different computation requests that arrive through the Agent API

	(described later on in this table). Upon the reception of this request, the API submits to the Runtime engine a task that encapsulates the execution of the main code of the computation. Such an engine is composed by four main components: ● Resource Manager: keeps track of the computing resources currently available (embedded on the device or available as agents on remote nodes). ● Task Scheduler: picks the resources and time lapse to host the execution of each tasks while meeting dependencies among them and guaranteeing the exclusivity of the assigned resources ● Data Manager: stores locally data values and establishes a data sharing mechanism across the whole infrastructure ● Executor Engine: handles the execution of tasks on the resources embedded on the local device (CPU, GPU, FPGA or any other accelerator) When the local computing devices execute a code programmed following the COMPSs/PyCOMPSs programming model, new tasks are spawned and submitted back to the runtime engine so that the runtime handles their execution in the same way as it was done for the main task.
Dependencies	Currently, COMPSs interacts with two open-source components: ● Kubernetes which deploys the agents in a virtualised, isolated and secure environments ● OSCAR, which triggers COMPSs function executions in a FaaS manner
Interfaces	COMPSs has two APIs: the Agent API and the Runtime engine API. External users and other architecture components interact with COMPSs through the REST implementation of the Agent API. This Agent API provides a method to request function execution indicating the values of the arguments or a source URI from where the agent can fetch the value. In addition, the API provides methods to query/modify the resource pool. There is another implementation of the Agent API leveraging a proprietary protocol building directly on TCP sockets. This version of the API is meant to be used internally among agents pursuing higher-performance communications. The Runtime engine API is meant to be used internally by each agent. As mentioned above, the runtime engine's purpose is to handle the execution of asynchronous tasks while guaranteeing the sequential consistency of the tasks. This interface is available natively in Java and there are bindings for C and Python based on JNI. Besides, other processes can interact with the runtime engine with a glue software that publicly offers

	this API through system pipes.
Data	To achieve its purpose, COMPSs requires the following data: - Service code. The runtime engine will invoke one of the functions of the code when the task scheduler subcomponent decides to run a task locally on the embedded computing devices. Generally, it is embedded on the container running the agent. - Resource configuration. Each agent needs to be set up indicating all the available resources embedded in it and the resources available on its neighbouring agents. This information is usually provided in a XML file at runtime; however, it can also be updated dynamically at run time calling the Agent API. - Service data. Any information required to run the service needs to be accessible to the COMPSs runtime. To provide the service, the data needs to be stored locally on the device or, if the data is in a remote store, COMPSs requires the device to have the appropriate software and credentials to access it.
Needed improvement	In AI-SPRINT, COMPSs is used for distributing the workload of a computation across several nodes in the Cloud-Edge-IoT Continuum. Scheduling systems are crucial for the performance of the deployed services, they decide which node will compute each application component and which data needs to be transferred. The current scheduling mechanism was designed for a master-worker approach where a master node has complete information - i.e., the master knows all the tasks to be computed, the features of all the devices within the infrastructure and the current workload of each node. This is no longer the case since COMPSs deploys an autonomous agent in each device able to start a workflow execution. To achieve better performance, new scheduling approaches should be explored.

Implemented Improvements for the First Release	The tasks performed throughout this first year of project are grouped into five big objectives which are detailed in depth in the next section
	● Creation of the agent to wrap the COMPSs runtime
	● Improve the COMPSs' resource management to :
	    ○ enable the dynamic addition/removal of resources and to handle abrupt resource disconnection,
	    ○ create resource hierarchies matching the natural layout of the Compute Continuum
	● Creation of utilities to deploy an agent and generate hierarchies
	● Integration of the Agent with Performance Analysis tools (Paraver)
	● Building prototypes to integrate with OSCAR

Release Version & Repository	The first release version of COMPSs for AI-SPRINT can be downloaded from: https://gitlab.polimi.it/ai-sprint/compss (daily mirror of the official repository https://github.com/bsc-wdc/compss).

4.7.1.1 Detailed Description of Activities for the First Release

The activities performed on COMPSs for this first release pursue the implementation of the improvements described in the previous section.

Creation of the Agent

Despite being a general-purpose programming model, the COMPSs runtime was mainly used as a workflow manager to execute the tasks composing a single execution of an application. As described above, COMPSs has evolved to become a service (agent) that allows the execution of functions on-demand. As with the main code of traditional applications, the login of these functions can be automatically converted into a task-based workflow, and the tasks composing it be executed in parallel to reduce the execution time. For this first release, the interfaces of this agent have been designed and implemented to invoke the COMPSs runtime to execute tasks.

Resource Management

The original COMPSs Runtime followed a master-worker approach; i.e., one node -the master- knows all the details of the infrastructure to perform an optimal workload distribution and the rest of the nodes run a simple client that hosts the executions as orchestrated by the master. In the Cloud-Edge-IoT Continuum, the amount of resources drastically increases, hindering the scheduling process. To reduce the complexity of this process, the COMPSs' organisation has evolved into a hierarchical peer-to-peer network where each node (Agent) reduces the scheduling problem to selecting to run the task on its local resources or offloading the task onto another agent. Part of the work done for this first release consisted on implementing efficient offloading mechanisms onto other agents.

On the other hand, Continuum resources might be mobile and the system needs to adapt quickly to respond to that change. The resource management of the runtime system has been enhanced to allow the addition and abrupt removal of resources on-the-fly.

Integration with Performance Tools

To validate the proper behaviour of the runtime and detect internal performance problems, the runtime engine uses Extrae [66] to emit events and create Paraver [67] traces that allow a post-mortem performance analysis. The master-worker COMPSs runtime produces a global trace that contains all the events of the execution in all the nodes. This mechanism has been adapted to the new approach of the runtime, where each agent produces a trace with the local events, and, later on, all the traces are gathered and merged into a single one.

Creation of utilities for agent deployment

To ease the deployment of a testing infrastructure we developed a script that deploys several agents and organises them in the shape of the desired hierarchy (chain, tree, flat). The script allows also to submit an invocation to one agent of the infrastructure, and, at its completion/failure, undeploy all the agents and produce the corresponding trace for an invocation performance analysis.

Building prototypes to integrate with OSCAR

To validate the proper integration with OSCAR, a container image was created with a sample application following the COMPSs programming model. Starting the container triggers the deployment of a COMPSs agent and an invocation to the Agent API requesting the execution of the corresponding function as a workflow.

4.7.1.2 Evaluation

To validate the results obtained by COMPSs, we performed experiments building on two different services: a Classification Service (training and inference) and a real-time video processing.

Classification Service

RandomForest is a classification algorithm that constructs a set of individual decision-trees, also known as estimators, each classifying a given input into classes based on decisions taken in a random order. The final classification of the model is the aggregate of the classification of all the estimators; thus, the accuracy of the model depends on the number of estimators composing it. The training of the estimators are independent

from each other, each one consisting of two tasks: a first one that selects a combination of 30,000 random samples from the training set, and a second one that builds the decision tree.

To conduct the experiments, we deployed the three-layer infrastructure depicted in Figure 4.7. Each node on the Cloud hosts the execution of an agent managing the 48 cores of the server. Below the Cloud, which is shared by all the users, we defined two disjoint domains, named Colonies, with a similar configuration. On the Fog level, each domain has a resource-rich node equipped with 24 CPU cores (green), and, at the Edge level, each domain has four resource-scarce devices with only 4 CPUs. All the nodes within the same domain share their workload, know the model and are able to run it to infer a class out of a reading.

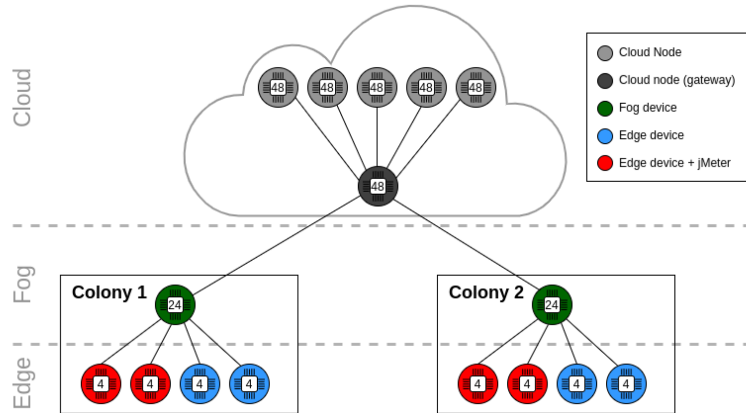


Figure 4.7 - Three-layer infrastructure managed by COMPSs

The training of the model allows us to validate the system running compute-heavy applications on the cloud. The following charts depict the results of the scalability tests for the training of the model varying the number of agents on the Cloud.

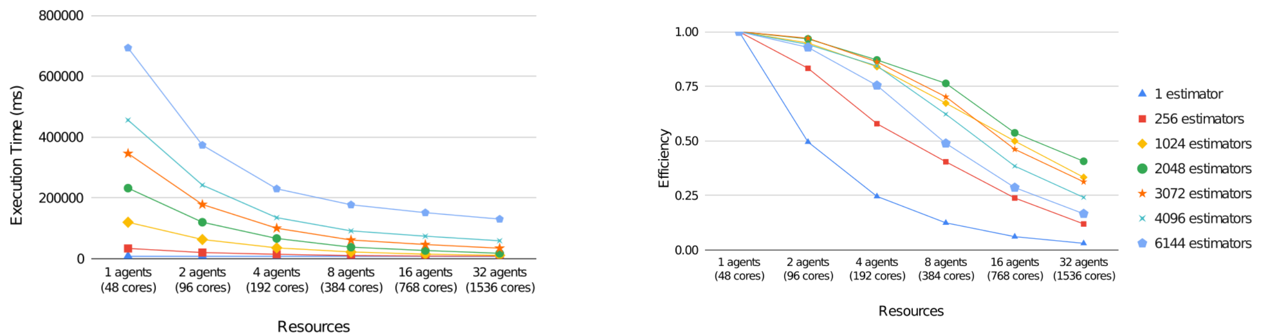


Figure 4.8 - Execution time (left) and efficiency (right) depending on the size of the infrastructure

The leftmost chart in Figure 4.8 illustrates the evolution of the training time when changing the number of resources to train a fixed-size model demonstrating the benefits of parallelisation: the larger the infrastructure grows, the shorter the execution time becomes. The rightmost chart in Figure 4.8 presents the efficiency (ratio between the speedup compared to the 1-agent execution of the same-size problem and the size of the infrastructure) of the execution and reveals some scalability problems. Up to 2048 estimators, the performance loss can be explained mainly by the load imbalance. The larger the infrastructure is, the more resources remain idle waiting for others to complete the training. A second cause is the overhead of COMPSs when detecting new tasks.

COMPSs sequentially detects tasks as the main code runs; the task creation delays build up. Training one estimator requires about 7 seconds (running both tasks). To keep the whole infrastructure busy on the 1,536 cores scenario, COMPSs must generate a task every 2.25 ms. The granularity of the tasks is too fine grained given the size of the infrastructure. Beyond 2048 estimators, the performance loss is explained by the implementation of the Task Scheduler; it has a single thread that handles in a FIFO basis both new task requests and end of task notifications. Hence, several end of task notifications may stack up before a new task request leaving several nodes idle waiting for a new task. Likewise, many new task requests might accumulate in front of the task end notification; thus, despite the resources being idle, the scheduler is not aware of that and does not offload more work to the node.

We used the inference to test the workload balancing mechanisms when serving several users. In this experiment, we attached a Jmeter instance simulating the workload generation on two of the Edge devices on each domain (red nodes in Figure 4.7). The blue devices are passive; however, they are able to run inferences requested by any other node in the domain. The number of agents deployed in the Cloud is limited to 6. The following charts depict the evolution of the number of requests handled by each device when 10, 100 and 300 users submit requests at an average pace of 1 request/second for 15 minutes.

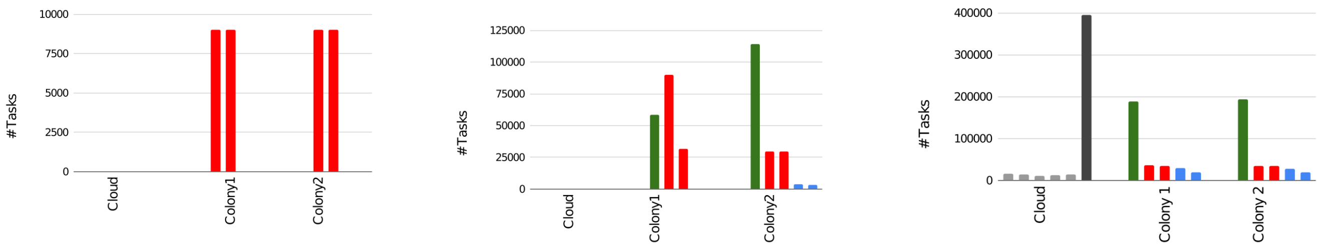


Figure 4.9 - Evolution of the number of requests handled by each device.

On the 10-user case (leftmost chart in Figure 4.9), the Edge nodes receiving the request are able to host all the workload; only 3 requests out of 36,000 were offloaded to the Fog nodes on peak-load moments. The average response time was 104 ms. In the 100-user case (chart in the middle), the nodes cannot assume the whole workload and offload onto their respective Fog colony which handles most of the requests. On Colony1, one of the edge nodes receives the requests at a periodic pace; thus, it is able to assume the whole workload (90,000 requests) by itself. On the other edge node, the requests arrive in bursts, and the scheduler decides to offload tasks to the Fog node (58,432). On Colony2, both edge nodes also receive the request in bursts, and both offload tasks to the Fog node which actually processes up to 116,041 requests. This Fog node cannot manage the whole workload and decides to offload part of it to the idle edge nodes (3,738 and 3,142 requests, respectively). The average response time is also 104 ms. Finally, in the last case (rightmost chart), with 300 users, both colonies cannot serve the workload locally and a large portion of the requests (464,999/1,080,000) are offloaded onto the Cloud. The average response time slightly increases to 109 ms.

Results of a real-time video processing

In the real-time video processing use case, the application obtains a video-stream directly from a camera, processes each of the frames to identify the people in it, and maintains an updated report with stats of their possible identifications. The application, originally being executed in a Raspberry Pi, needed 41,293 ms to process each of the frames when one person appeared in it achieving a frame rate of 0.024 fps.

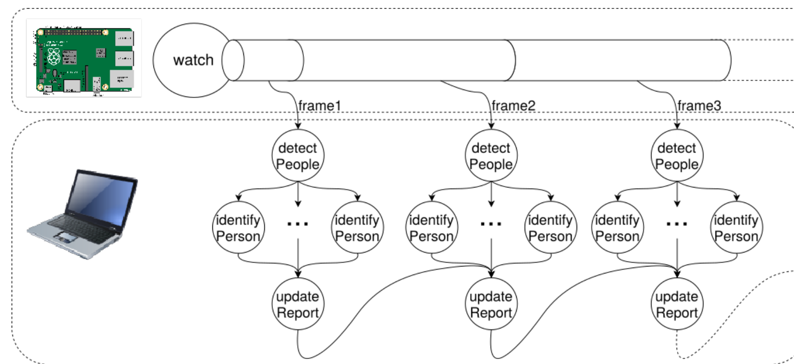


Figure 4.10 - Using COMPs in the application's algorithm

We taskified with COMPSs the application's algorithm as depicted in the Figure 4.10. One task was collecting images with the camera on the Raspberry Pi, and, upon the collection of each frame, a new workflow was created to process it. In this experiment, the testbed consisted only of the Raspberry Pi and a laptop with 4 cores. With COMPSs, the frame rate raises to 2.79 fps achieving a 116.25x speed up. The bottleneck of the application is the Raspberry Pi that needs 353 ms to obtain a frame and serialize it.

4.7.2 OSCAR

Identification	Open Source Serverless Computing for Data-Processing Applications (OSCAR)
Type	A service and a set of clients
Purpose	OSCAR is an open-source platform to support the Functions as a Service (FaaS) computing model for file-processing applications. It can be automatically deployed on multi-Clouds in order to create highly-parallel event-driven file-processing serverless applications that execute on customised runtime environments provided by Docker containers than run on an elastic Kubernetes cluster. With the ability to deploy OSCAR clusters in minified Kubernetes distributions such as K3s, event-driven workflows along the computing continuum can be executed, since it can also run on constrained devices such as Raspberry Pis.
Function	With OSCAR, users upload files to a data storage back-end and this automatically triggers the execution of parallel invocations to a service responsible for processing each file. Output files are delivered into a data storage back-end for the convenience of the user. The user only specifies the Docker image and the script to be executed, inside a container created out of that image, in order to process a file that will be automatically made available to the container. The deployment of the computing infrastructure and its scalability is abstracted away from the user.
High level Architecture	The following image describes the high-level architecture of OSCAR, including its dependencies with the storage back-ends supported.

Dependencies	OSCAR depends on the following open-source components, classified in the following categories: ● Storage back-ends: Amazon S3 [15], MinIO [16] and Onedata [17] (supported by the EGI DataHub [18]). ● Deployment of OSCAR clusters: Infrastructure Manager (IM) [1] and, alternatively, Elastic Cloud Computing Cluster (EC3) [19]. ● Runtime: Kubernetes [21] and FaaS Supervisor [29]. ● FaaS framework for synchronous invocations: OpenFaaS [20] (KNative [22] support is partially included, awaiting its support for volumes) OSCAR can optionally integrate with SCAR (Serverless Container-aware Architectures) [24], an open-source framework to transparently execute containers out of Docker images in AWS Lambda [25], integrated with AWS Batch [26] in order to create cross-services event-driven serverless workflows along the Cloud continuum, involving public FaaS services.
Interfaces	An OSCAR cluster supports the following interfaces: - A web-based graphical user interface for end users - An authenticated REST API - A CLI which interacts with the REST API
Data	An OSCAR cluster uses several types of information: ● Definition of workflows of functions described in the Functions Definition Language (FDL) [23] ● A customizable TOSCA template [27] of the OSCAR cluster for its automated deployment on multi-Clouds via the Infrastructure Manager ● Token-based authentication for the REST API and to optionally access the underlying the Kubernetes cluster ● Username and password to connect to the web-based GUI

Needed improvement	In AI-SPRINT, OSCAR is employed to support the scalable inference of pre-trained AI models out of data files being generated by the use cases, along the computing continuum. To this aim, first, OSCAR needs to be deployed in low-powered devices such as Raspberry Pis, in order to support inference in the edge, close to where the data is being generated. Second, it needs to support synchronous invocations in order to reduce the latency involved in provisioning the underlying computing resources by reusing the execution runtime. Third, OSCAR needs to support the security requirements identified by the use cases in terms of controlled access, privacy-aware execution. Fourth, OSCAR needs to support accelerated computing devices such as GPU devices in order to speed up the execution of compute-intensive inference processes.
Implemented Improvements for the First Release	The first release of OSCAR regarding AI-SPRINT includes the following improvements: - Support for deployment of OSCAR in clusters of Raspberry Pis, based on the K3s [28] minified distribution of Kubernetes. - Support for synchronous invocations for low-latency inferences, based on OpenFaaS. - Partial support of the security requirements, by implementing token-based authentication for service invocation endpoints. Support for secure enclaves via SCONE will be relegated to next releases. - Initial support for GPU-based computing, by allowing access to native GPUs via nvidia-docker and initial support for rCUDA, to exploit remote GPUs. We plan to further extend this support in subsequent releases.
Release Version & Repository	The first release version of OSCAR for AI-SPRINT can be downloaded from: https://github.com/grycap/oscar/releases/tag/v2.2.0 and from the AI-SPRINT GiLab (daily mirror) https://gitlab.polimi.it/ai-sprint/oscar

4.7.2.1 Detailed Description of Activities for the First Release

This section provides additional details on the activities carried out within AI-SPRINT in OSCAR for the first release.

Deployment of OSCAR in Raspberry Pis

A 4-node cluster of Raspberry Pis was procured (shown in Figure 4.11) in order to support the deployment of an OSCAR cluster on these low-power devices thanks to the use of a minified Kubernetes distribution such as K3s. For this, the FaaS supervisor, the component in OSCAR in charge to support the execution of the functions together with the data stage in and out with different storage back-ends had to be adapted to an arm64 architecture. Now, releases of this component are provided both for amd64 and arm64 architectures, as shown in this example release for the FaaS Supervisor [30].

Figure 4.11 - Cluster of Raspberry PIs with four nodes to support OSCAR running on K3s for the edge.

In addition, arm64-based packages for all the related dependencies identified in the table above are deployed in this cluster.

Synchronous Invocations Low-Latency Inferences

OSCAR mainly focuses on asynchronous event-driven executions of compute-intensive tasks, so that a file uploaded to MinIO triggers the execution of a certain application packaged as a Docker container. This typically introduces a certain overhead (in the order of several seconds) involved in resource provisioning, container image downloading and creation of the container itself. In order to speed up this procedure we enabled support for synchronous invocations in OSCAR thanks to the use of OpenFaaS, which keeps certain Docker containers active to be reused for the execution of several requests. By reusing the execution environment, the overhead can be mitigated. The downside of this approach, with respect to the asynchronous invocations, is that a sudden spike of synchronous invocations may overload the provisioned Docker containers and, thus, additional delay is introduced until new resources are provisioned. Therefore, in order to offer flexibility we will support both kinds of execution schemes for use cases to adopt whichever best suits their requirements.

Token-based Authentication

Initially, OSCAR supported basic access authentication (basic auth) [31] for system and service management REST API endpoints. However, paths for synchronous and asynchronous invocation of services were open, since MinIO does not provide support for basic auth on its webhook notification system. As it supports Bearer Authentication [34], we decided to introduce token-based authentication in the OSCAR server in order to protect invocation endpoints of services. For the time being, these are auto-generated long-lived tokens that are stored in the service's description and HTTP over TLS (Transport Level Security) is being employed in the communications between the client and the OSCAR server so that the token cannot be eavesdropped.

Support for GPU-based computing

OSCAR already included certain support for GPU-based computing based both on native GPUs (local to nodes where the OSCAR cluster is being executed) and through rCUDA, as described in the work by Naranjo et al. [32]. In the field of AI-SPRINT we maintained the existing support for newer rCUDA versions to include this support for the use cases, in case it is required.

4.7.2.2 Evaluation

In order to assess the proper operation and elasticity of the OSCAR platform, a series of experiments were carried out on an example workflow for object recognition on frames extracted from videos. The tests were performed with videos of different lengths to simulate diverse workloads and thus analyse the behaviour of

CLUES, the elasticity system deployed by the IM, on the OSCAR platform for starting new nodes. The configuration of the platform adopted in the experiments defines a maximum of 10 working nodes, each of which is designated as a virtual machine with 4 vCPUs and 8 GiB of RAM on top of an OpenNebula-based on-premises cloud. It is important to mention that the OSCAR platform in elastic mode always has a working node ready to process events.

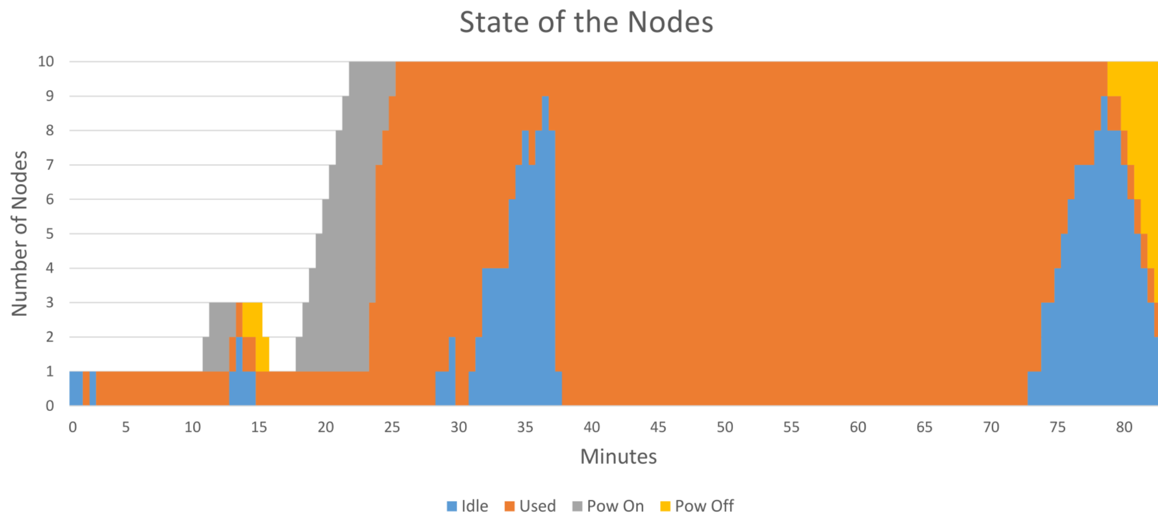


Figure 4.12 - State of the nodes in the OSCAR cluster along the execution time

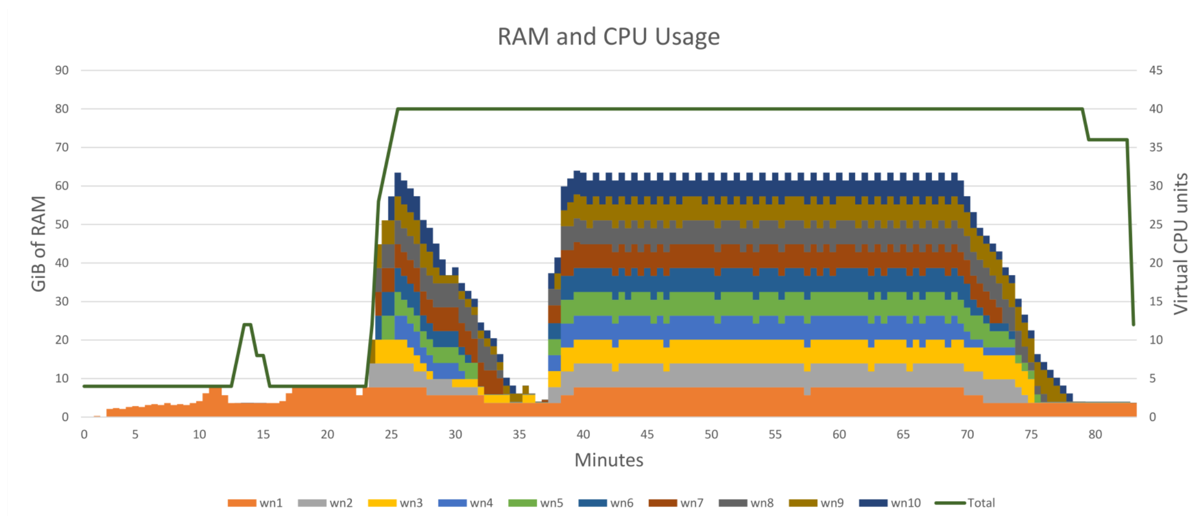


Figure 4.13 - Evolution of the RAM and CPU usage during the execution across the different nodes.

Figures 4.12 and 4.13 show the state of the infrastructure after the execution of three different workloads: i) in minute 10 a video is sent whose duration generates 10 frames to be processed; ii) in minute 17 another video that generates 100 frames and iii) finally in minute 37 a longer-duration video is submitted, generating 1,000 frames. On the one hand, Figure 4.12 shows the number of nodes powered on the platform, which after the spikes following the uploading of the videos, grow and shrink according to the workload. On the other hand, Figure 4.13 shows the number of resources utilised on the platform to process the files.

As can be appreciated, the combination of CLUES and the IM on Kubernetes clusters allows the OSCAR platform to be deployed on multiple cloud providers with the minimum number of virtual machines powered on, which can, however, be automatically increased to satisfy more demanding workloads.

4.8 Application Reconfiguration

These components are responsible to periodically reevaluate the initial resource allocation to consider load variations and degradations of the components' response times. Application migration, to the extent that is possible, will be triggered to counteract changes in the underlying performance of computing and network resources.

4.8.1 SPACE4AI-R

Identification	SPACE4AI-R
Type	Service
Purpose	Generating a new optimal components placement solution for the system at runtime, adapting the current assignment in terms of application components configurations and resources to the current load.
Function	SPACE4AI-R, initially triggered by the monitoring infrastructure or run periodically, checks whether QoS local and global constraints (namely, requirements on the maximum admissible response times of single components or sequences of components, see *AI-SPRINT Deliverable D2.1 - First release and evaluation of the AI-SPRINT design tools*) are satisfied. If any constraint is violated, the application manager (see below) invokes the application optimizer to determine a new optimal solution, adapting the current one to the actual load, and providing, possibly, an updated deployment description to the Infrastructure Manager (IM) server. The new optimal solution may include changing the components configuration (see section 4.6 in *D2.1 First release and evaluation of the AI-SPRINT design tools*), and/or scaling the number of cloud VMs used to run specific components and migrating some components from edge to cloud or vice versa.
High level Architecture	SPACE4AI-R is a separate component which includes the application manager providing a REST API and the application optimizer.
Dependencies	SPACE4AI-R depends on the performance model, resource descriptions and SPACE4AI-D, which includes the optimal placement of application components at design time. Moreover, SPACE4AI-R relies on Infrastructure Manager (IM) to deploy additional physical resources on cloud or edge and OSCAR for component execution migrations. SPACE4AI-R queries the monitoring tools which provide execution time of the running components and resource consumption (e.g., CPU utilisations).
Interfaces	SPACE4AI-R requires input files including the performance model of the application components and constraints, resource descriptions (see section 4.6 in *D2.1 First release and evaluation of the AI-SPRINT design tools*) and monitoring data. SPACE4AI-R will be integrated with IM and OSCAR for triggering the reconfiguration mechanisms of the AI-SPRINT runtime.

Data	The core input file of SPACE4AI-R will have the same syntax and semantic as SPACE4AI-D (see D2.1, section 4.7)
Needed improvement	This is a completely new development.
Implemented Improvements for the First Release	The first release of the tool will be provided at M24.
Release Version & Repository	Not applicable.

4.8.2 Krake

Identification	Krake
Type	Orchestrator Engine
Purpose	Krake [ˈkʀaːkə] is an orchestrator engine for containerised and virtualised workloads across distributed and heterogeneous cloud platforms. It creates a thin layer of aggregation on top of the different platforms (such as OpenStack, Kubernetes or OpenShift) and presents them through a single interface to the cloud user. The user's workloads are scheduled depending on both user requirements (hardware, latencies, cost) and the platform's characteristics (energy efficiency, load). The Krake scheduling algorithm can be optimised for example on latencies, cost, or energy. Krake can be leveraged for a wide range of application scenarios such as central management of distributed compute capacities as well as application management in Edge Cloud infrastructures. In AI-SPRINT, Krake provides the ability to perform application migration (and in particular rCuda clients) across Kubernetes clusters that may reside in different levels of the computing continuum. Furthermore, it ascertains the "best" level of resource usage based on user-defined parameters and re-evaluates the deployment periodically.
Function	Automated scheduling and migration service for containerised applications across heterogeneous cloud platforms based on predefined label constraints and metrics.
High level Architecture	The components in the architecture of Krake are loosely coupled and therefore can be exchanged, altered or extended with new components. The following figure shows the overall architecture of Krake.

	The control Plane is in continuous exchange with Krake's API and real world objects and therefore continuously manages all real world objects registered as resources in Krake. Real world objects are defined as different instances like Kubernetes clusters or Metrics Providers like Prometheus. The Kubernetes Application Controller of Krake manages containerised applications which are migrated to Kubernetes clusters; these clusters need to be registered in Krake. To take care of label constraints given by the Krake API or custom metrics distributed through the Metrics Providers, the scheduler watches and evaluates these indicators and decides which Kubernetes cluster is the best fit to support the application execution.
Dependencies	Besides a proper Krake installation, functionality depends on two other things: - at least one Kubernetes cluster must be set up - to run an application, Krake needs a containerised application (the image should be Kubernetes compatible) in a registry (like Docker Hub, GitLab Container Registry).
Interfaces	Internally the communication between Krake components is REST-based. The interaction between a user and Krake however is done via Rok, Krake's own command line interface.
Data	Krake uses two concepts for the usage of data: 1. All internal configurations, resource information like metric information, cluster information or application information is stored in Krake's *etcd* database. 2. The configuration is done via YAML configuration files which are bootstrapped to the *etcd* database, according to a user's preferred behaviour of Krake these files need to be adjusted.

Needed improvement	● Stateful Applications: ○ with external databases (phase 1) - in progress ■ prerequisites: shutdown hooks - in progress ○ with direct data transfer (phase 2) ● Docker controller to schedule/reschedule docker containers via Krake directly on VMs ● QoS, performance, energy related scheduling/rescheduling - in progress ● Support for new metrics and metrics provider if needed ● Usage of TOSCA templates for app deployment - in progress ● Handling of edge nodes that may be temporarily unavailable
Implemented Improvements for the First Release	Parts of the needed features of Krake are currently under development and will be available in the next months, namely: ● Stateful Applications: ○ with external databases (phase 1) - in progress prerequisites: shutdown hooks - in progress ● QoS, performance, energy related scheduling/rescheduling - in progress ● Usage of TOSCA templates for app deployment - in progress
Release Version & Repository	Repository: https://gitlab.com/rak-n-rok/krake and also at https://gitlab.polimi.it/ai-sprint/krake (daily mirrored) Documentation: https://rak-n-rok.readthedocs.io/projects/krake/en/latest/index.html

4.8.2.1 Detailed Description of Activities for the First Release

So far some of the long term needed features of Krake in a AI-SPRINT context are under development and set to be finished with the end of the next Krake milestone at the end of April 2022.

OpenStack

In regard to providing computational power as a cloud provider, Cloud & Heat (C&H) was able to install two ordered GPU enabled servers. The available GPUs consist of one NVidia A100 in the first server and two AMD MI100 GPUs in the second server. Besides that, OpenStack was installed on the servers as a cloud computing platform.

Name	VCPUS	RAM	Total Disk	Root Disk	Ephemeral Disk	Public
XXS	1	512 MB	10 GB	10 GB	0 GB	Yes
XS	1	1 GB	10 GB	10 GB	0 GB	Yes
S	1	2 GB	15 GB	15 GB	0 GB	Yes
S.mem+	1	4 GB	15 GB	15 GB	0 GB	Yes
M	2	4 GB	25 GB	25 GB	0 GB	Yes
M.mem+	2	8 GB	25 GB	25 GB	0 GB	Yes
L	4	8 GB	50 GB	50 GB	0 GB	Yes
L.mem+	4	15 GB	50 GB	50 GB	0 GB	Yes
XL	8	15 GB	100 GB	100 GB	0 GB	Yes
XL.mem+	8	30 GB	100 GB	100 GB	0 GB	Yes
AIS-8C:256:200s-GN:GA100	8	256 GB	200 GB	200 GB	0 GB	No
AIS-8C:452:200s-GN:GA100	8	452 GB	200 GB	200 GB	0 GB	No
AIS-8C:256:200s-GA:MI100	8	256 GB	200 GB	200 GB	0 GB	No
AIS-8C:452:200s-GA:MI100	8	452 GB	200 GB	200 GB	0 GB	No
AIS-8C:256:200s-GA:2xMI100	8	256 GB	200 GB	200 GB	0 GB	No
AIS-8C:452:200s-GA:2xMI100	8	452 GB	200 GB	200 GB	0 GB	No

‹ Back | Next › | Launch Instance

Figure 4.14 - Openstack flavor configuration for VM instances

In that regard, ten standard and six custom GPU flavors were made available for instances on the two physical machines (see Figure 4.14). VMs can be set up with different Linux distributions, e.g., Ubuntu or Fedora. The standard flavors share a RAM and vCPU pool whereas the GPU flavors are restricted to its dependent physical machines. Changes in flavor configurations can be made during the whole project time. Also worth noting, C&H provides custom domains in the form of “example.oncloudandheat.com” to make AI-SPRINT technologies which run on the C&H cloud infrastructure publicly available.

Krake & rCuda client migration

Beside the task of in general providing computational power as a cloud provider, these steps were also made in order to provide a computational environment for Krake. So Krake will be put in a VM on an instance of said OpenStack platform. This is also necessary to migrate rCuda clients between different Kubernetes clusters with Krake.

4.8.2.2 Evaluation

The general integrity and usability of Krake is continuously tested and thus evaluated. In Krake Gitlab, there are pipelines defined which are automatically triggered every day and with each commit and merge of new

code. Therefore, a test driven development has been followed for more than two years now. The Krake project proves its quality by providing 96% of test coverage at the current time, which means that this percentage of Krake's code is always tested. Figure 4.15 refers to unit testing the code itself. In addition end-to-end tests are also triggered each day, which means that Krake itself is installed and Kubernetes clusters set up on OpenStack, so that Krake's functionalities like scheduling and migration are tested automatically.

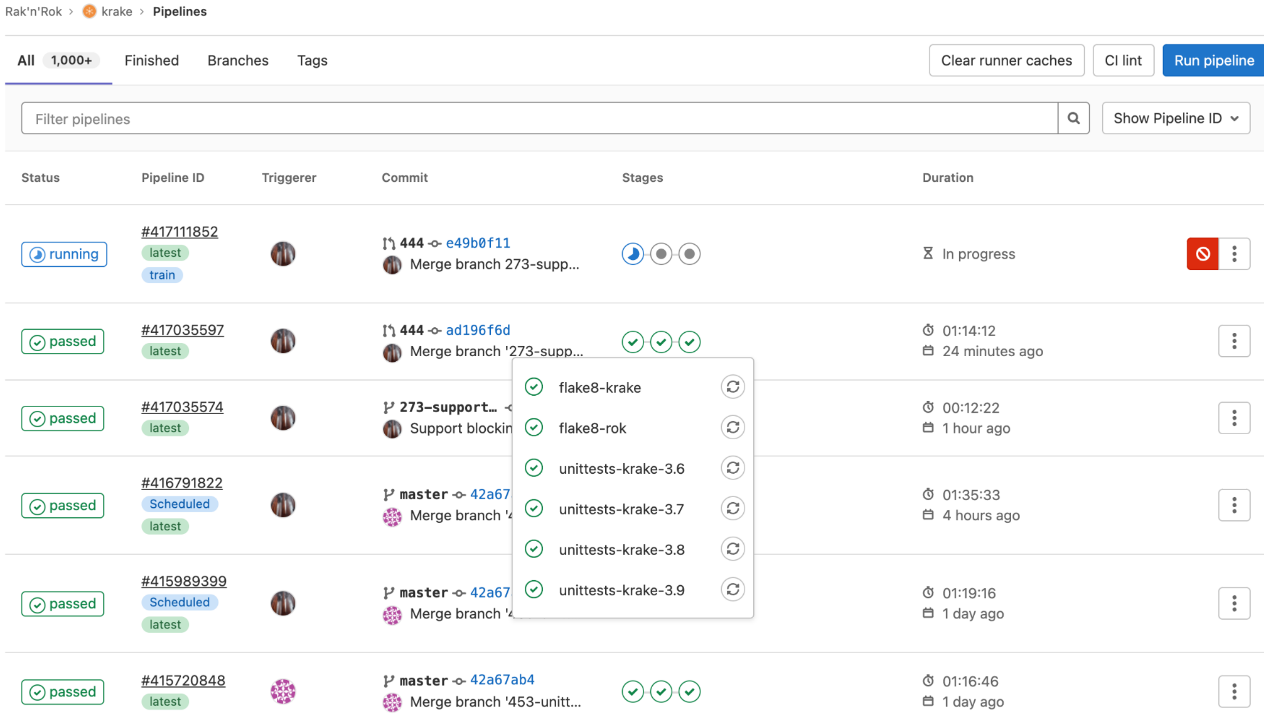


Figure 4.15 - Continuous testing of all implemented Krake features

Furthermore, the process of implementing new features, bug fixes or new code in general is guided by a review loop. For each merge request, one or more reviewers, other than the contributor, need to be set to check the new code to ensure that only high-quality code will be merged into the Krake master branch.

All features under development and to be developed that are required for AI-SPRINT will also undergo this process. This ensures a continuous evaluation of quality, code and features.

5. Towards the Integrated Framework for the Runtime Environment

The general architecture for the AI-SPRINT project was introduced in deliverable *D1.3 Initial design of the architecture*, identifying the main technological building blocks. This section describes the work carried out towards achieving this integrated framework by first addressing how the example application described in Section 2 has been implemented using the technological components identified in section 4. Notice that not all the components have been used at this stage, since there are disparate maturity levels for each component. Also, the focus has been set on the inference of pre-trained models along the Cloud continuum. Integration to support the training step will be the focus on the next year of the project. Still, this lead-by-example exercise paves the way to onboard the project's use cases by following similar technological adoption paths.

5.1 Integration for the Example Application

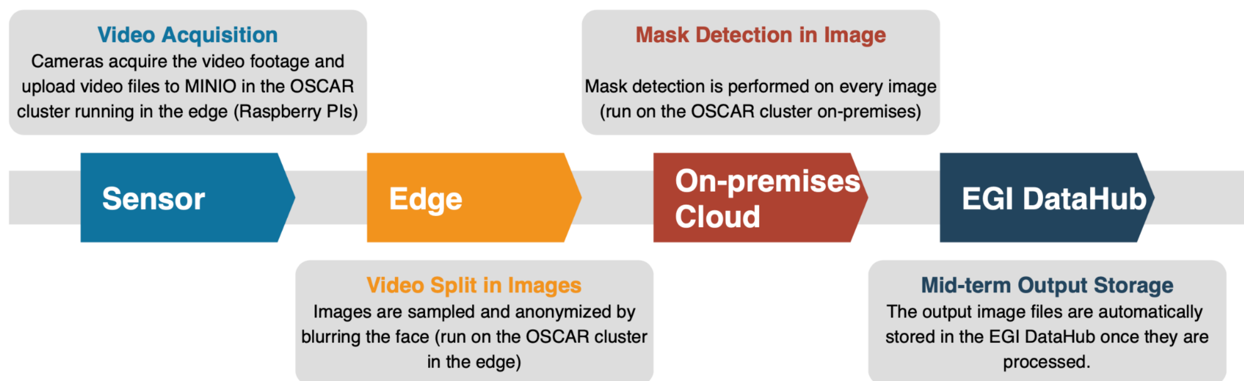


Figure 5.1 - Steps involved in the example application (anonymised mask detection)

In order to support the execution of the anonymised mask detection application, we used the flow depicted in Figure 5.1 and the technological components shown in Figure 5.2.

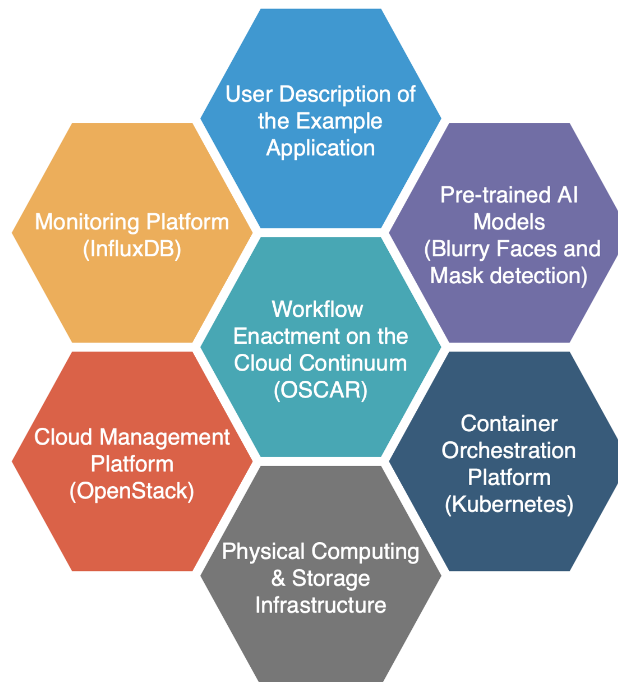


Figure 5.2 - Components involved in the implementation of the example application (anonymised mask detection)

Physical Computing & Storage Infrastructure

To perform data processing at the edge, we used a cluster of 4 Raspberry Pis 4 Model B with the following hardware configuration per node:

- Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC running at 1.5GHz.
- 4GB LPDDR4-3200 SDRAM.
- 2.4 GHz and 5.0 GHz IEEE 802.11ac wireless.
- Bluetooth 5.0.
- 1 Gigabit Ethernet port.

We also used the EGI DataHub, one of the data storage solutions offered by EGI. In particular, a Oneprovider hosted at Cyfronet [50] supporting our Onedata space was used, in order to support long-term data persistence of the files that result from the inference process. This allows us to showcase an integration with the EOSC (European Open Science Cloud) initiative, by providing interoperability between the computing platform and the data storage solution.

Cloud Management Platform (OpenStack)

The provisioned virtual computing infrastructure to support the mask detection part of the inference process (see Figure 2.1 for additional details) is carried out from an OpenStack-based on-premises Cloud offered by the UPV and integrated with the EGI Federated Cloud. Again, this demonstrates our commitment to align our technological developments with the EOSC initiative. The details of the OpenStack cloud are as follows:

- 14 Intel Skylake Gold 6130 processors, with 14 cores each.
- 5.25 TB of RAM.
- 2 x 10GbE ports per node.
- 1 Infiniband port per node.

Container Orchestration Platform (Kubernetes)

Kubernetes is used as the underlying container orchestration platform employed for both OSCAR clusters, the one running in the cluster of Raspberry Pis (edge) and the one running in the OpenStack cloud (on-premises Cloud).

Workflow Enactment on the Cloud Continuum (OSCAR)

The inference process consists of two steps, anonymisation in the edge and mask detection in the on-premises Cloud. Therefore, two services in OSCAR where created, as shown in the following snippet of code that uses the Function Definition Language (FDL) to define the data-driven workflow in the Cloud continuum:

```
functions:
  oscar:
  - oscar-edge:
      name: anon-and-split
      memory: 2Gi
      cpu: '2.0'
      image: ghcr.io/srisco/blurry-faces-arm64
      script: blurry-faces.sh
      input:
      - storage_provider: minio
        path: demo/in
      output:
      - storage_provider: minio.cloud
        path: demo/med
      - storage_provider: onedata.my_onedata
        path: demo/med
  - oscar-cloud:
      name: mask-detector
      memory: 1Gi
      cpu: '1.0'
      script: mask-detector.sh
      image: grycap/mask-detector-yolo:full
      input:
      - storage_provider: minio
        path: demo/med
      output:
      - storage_provider: minio
        path: demo/res
      - storage_provider: onedata.my_onedata
        path: demo/res

storage_providers:
  minio:
    cloud:
      endpoint: https://minio.determined-jang7.im.grycap.net
      verify: True
      region: us-east-1
      access_key: minio
      secret_key: xxxxx
  onedata:
    my_onedata:
      oneprovider_host: plg-cyfronet-01.datahub.egi.eu
```

```
        token: xxxxx
        space: srisco-space
```

The first service, “anon-and-split” is running out of an arm64 based Docker image that was specially crafted to run on the Raspberry Pis. It is allocated 2 GBs of RAM and 2 vCPUs. Notice that the output storage provider used for the result of these executions is the input storage provider that triggers the execution of the second service “mask-detector”, which runs in the OSCAR cluster in the OpenStack cloud site. This second service uses two storage providers, the local MINIO installation inside that OSCAR cluster and a Onedata space provided by the EGI DataHub.

This way, we are able to compare side-by-side the output images of the mask detection process, so that a human can easily and rapidly check if the inference process is proceeding appropriately. To this aim we are using an open-source tool developed called BLISS (Bucket-based Listed Images Side-to-Side) [33].

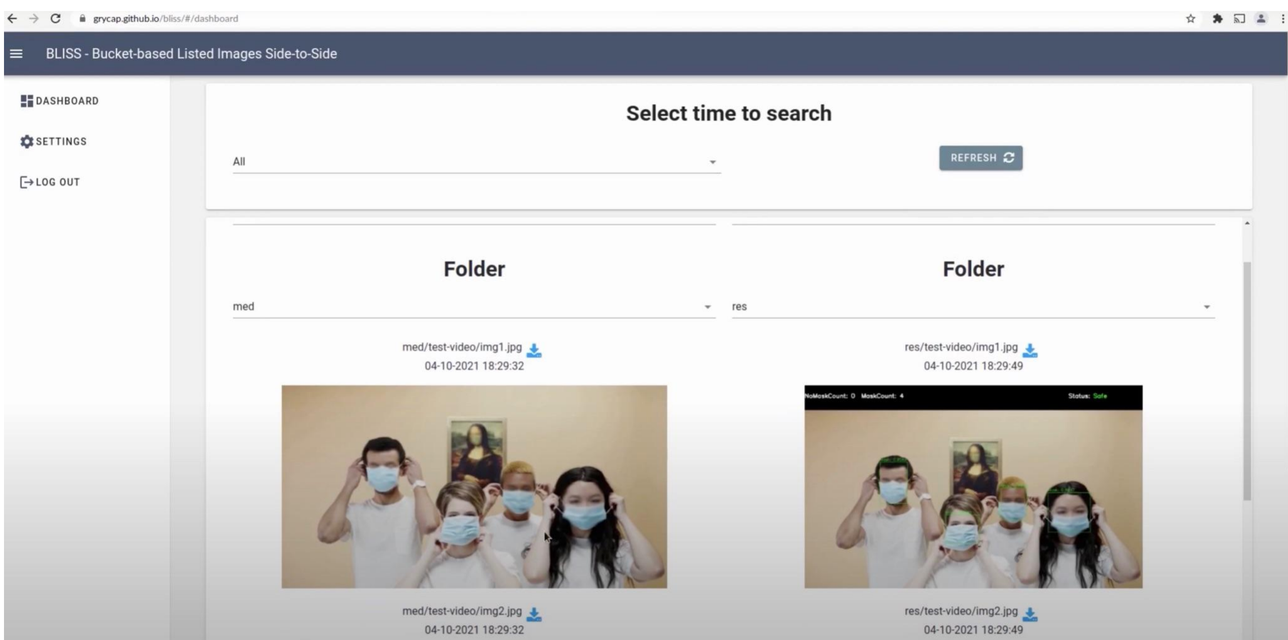


Figure 5.3 - Screenshot of BLISS to compare the images before and after the mask detection.

In order to properly monitor the execution time, including the overheads introduced by the execution platform, we designed an interception mechanism shown in Figure 5.4 (left) that detects whenever a new file has been uploaded to MinIO and contacts InfluxDB in order to store the start time for the inference of a particular file. An UUID (Universally Unique Identifier) is inserted in the file name to uniquely track an inference request on a particular file along its execution throughout the platform. This way, several components can register the start and end time in InfluxDB for that inference request in order to properly isolate the execution time required in each step.

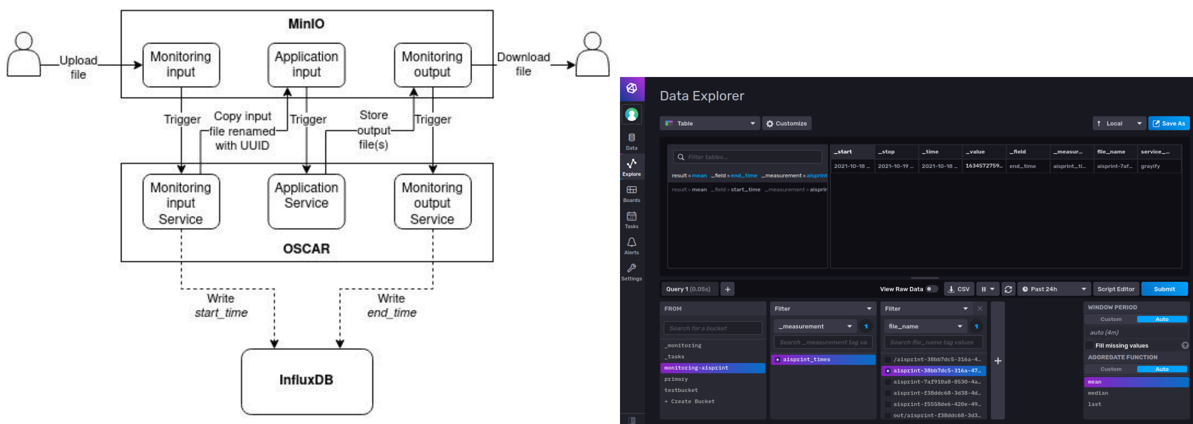


Figure 5.4 - Architecture for the monitoring integration between OSCAR and InfluxDB (left). Snapshot of InfluxDB with timestamps inserted to trace the execution time of inference-based executions (right).

Even if not all the components available in the runtime environment have been integrated for this example use case, we have been able to demonstrate the ability to trigger an inference workflow that involves resources at the edge of the network (cluster of Raspberry PIs) and Cloud resources (dynamically provisioned from a federated Cloud). This paves the way for an enhanced integration among the rest of the components provided by WP3 in order to cope with the requirements identified by the AI-SPRINT use cases. A "lead by example" approach will facilitate additional technology demonstrators more aligned with the use case needs to converge to the final use case implementations that involve a deeper integration level among the components of the runtime environment.

5.2 Integration plan for the Runtime Environment

The integration plan aims at harmonising the definition of the requirements with the design and development phase and ensuring that scientific and technical activities comply with the use cases definition. The integration plan is validated through verification activities and milestones in the work plan of the project, as depicted in Figure 5.5, extracted from Deliverable *D1.3 Initial design of the architecture.*

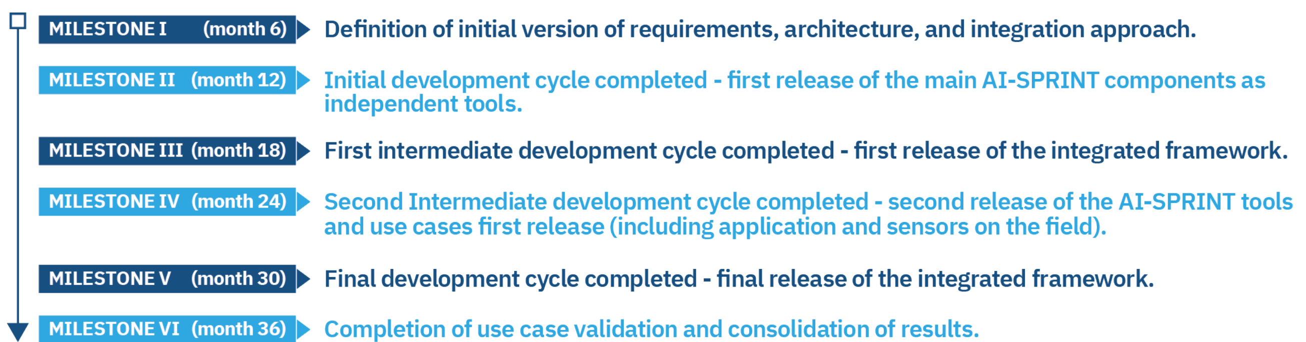


Figure 5.5 - Milestones for components development

A detailed roadmap for each set of components is available in Sections 6.2, 6.3 and 6.4 of the *AI-SPRINT Deliverable D1.2 - Requirements Analysis document.*

The second intermediate development cycle will involve the integration of the components from WP3 to support not only inference along the computing continuum but also training on Cloud platforms including the use of accelerator devices such as GPUs and reconfiguration of resources to include QoS requirements. Increased support in the runtime for the Cloud computing continuum will be extended by integrating OSCAR with the COMPSs runtime to exploit parallelism of the computational resources for accelerated inference, and also for reducing the time of training. For this, GPU scheduling will be integrated in the platform to decide

the optimal allocation of GPU resources together with certain application reconfiguration upon changes in the underlying performance of the computing resources, including but not limited to migration of application execution among different computing resources to account for increased latencies. An enhanced monitoring platform will be in place that supports hierarchical data synchronisation from the edge to the Cloud, including but not limited to infrastructure-level metrics and application-level metrics. Federated learning and privacy-aware executions will be performed using distributed approaches for learning and using SCONE for enhanced security and data-privacy.

The final development cycle will include enhanced support and adaptations to the specificity of the different use cases and will build on the experience gathered on the previous development cycles to streamline the integration among the several WP3 components, resolve potential performance bottlenecks and reduce the overheads of the different software layers.

In M36, the use cases will have adopted the WP3 components to support the runtime execution along the computing continuum. A validation phase will be performed beforehand to double check that the requirements defined by the use cases are met with the prototypes defined.

6. Conclusions

This deliverable has provided the first release and evaluation of the runtime environment for the AI-SPRINT platform. This document is one of the results of the first twelve months of activities.

This document has presented the different layers of the AI-SPRINT architecture, focusing on the runtime environment. A detailed description of the components has been provided together with the needed improvement for AI-SPRINT and the additional developments carried out in the first year of the project aligned with the use case requirements.

The second version of this document will be provided in M24 in *D3.3 Second release and evaluation of the runtime environment*, while the final version will be provided in M30 in *D3.5 Final release and evaluation of the runtime environment*, after completing the different phases of developments that will include the feedback of the use cases.

7. References

[1] IM (Infrastructure Manager). https://www.grycap.upv.es/im

[2] OSCAR (Open-source Serverless Computing for Data-processing Applications). https://oscar.grycap.net

[3] SCAR (Serverless Container-aware ARchitectures). https://github.com/grycap/scar

[4] CLUES (Cluster Elasticity System). https://www.grycap.upv.es/clues

[5] Krake. https://gitlab.com/rak-n-rok/krake

[6] OSCAR's FDL (Function Definition Language). https://grycap.github.io/oscar/fdl/

[7] AWS Lambda. https://aws.amazon.com/lambda/

[8] VMRC (Virtual Machine Image Repository and Catalog). https://www.grycap.upv.es/vmrc

[9] EGI AppDb (Applications Database). https://appdb.egi.eu/

[10] Ansible. https://www.ansible.com/

[11] Ansible Galaxy (GRyCAP roles). https://galaxy.ansible.com/grycap

[12] Blurry Faces. https://github.com/asmaamirkhan/BlurryFaces

[13] Face mask detector. https://github.com/adityap27/face-mask-detector/

[14] OSCAR Mask detector workflow. https://github.com/grycap/oscar/tree/master/examples/mask-detector-workflow

[15] Amazon S3. https://aws.amazon.com/s3/

[16] MinIO. https://min.io

[17] Onedata. https://onedata.org

[18] EGI DataHub. https://www.egi.eu/services/datahub/

[19] Elastic Cloud Computing Cluster (EC3). https://www.grycap.upv.es/ec3

[20] OpenFaaS. https://www.openfaas.com

[21] Kubernetes. http://kubernetes.io

[22] KNative. https://knative.dev/docs/

[23] Functions Definition Language. https://grycap.github.io/oscar/fdl/

[24] SCAR (Serverless Container-aware ARchitectures). https://github.com/grycap/scar

[25] AWS Lambda. https://aws.amazon.com/lambda/

[26] AWS Batch. https://aws.amazon.com/batch/

[27] OSCAR's sample TOSCA template. https://github.com/grycap/im-dashboard/blob/master/tosca-templates/oscar_elastic.yaml

[28] K3s. https://k3s.io

[29] FaaS Supervisor. https://github.com/grycap/faas-supervisor

[30] FaaS Supervisor 1.4.1 release. https://github.com/grycap/faas-supervisor/releases/tag/1.4.1

[31] Basic access authentication. https://en.wikipedia.org/wiki/Basic_access_authentication

[32] D.M. Naranjo, S. Risco, C. Alfonso, A. Pérez, I. Blanquer, and G. Moltó. Accelerated serverless computing based on GPU virtualization. Journal of Parallel and Distributed Computing, 139:32–42, may 2020.

[33] Bucket-based Listed Images Side-to-Side. https://github.com/grycap/bliss

[34] Bearer Authentication. https://swagger.io/docs/specification/authentication/bearer-authentication/

[35] F. Silla, S. Iserte, C. Reaño and J. Prades. On the benefits of the remote GPU virtualization mechanism: The rCUDA case, in Concurrency and Computation: Practice and Experience, vol. 29, no. 13, 2017.

[36] F. Lordan, et al. Servicess: An interoperable programming framework for the cloud. Journal of grid computing, vol. 12 (2014): 67-91.

[37] M. Gendreau and J. Y. Potvin, Eds., Handbook of Metaheuristics. International Series in Operations Research & Management Science. Springer International Publishing, 2019.

[38] Amazon web service pricing list, 2021. https://aws.amazon.com/ec2/pricing/on-demand/?nc1=h_ls

[39] Azure cloud services pricing list, 2021. https://azure.microsoft.com/en-us/pricing/details/cloud-services/

[40] E. Gianniti., L. Zhang., and D. Ardagna. Performance prediction of gpu-based deep learning applications, in CLOSER, vol. 1, 2019, pp. 279–286.

[41] V. Saxena, K. R. Jayaram, et al. Effective elastic scaling of deep learning workloads, in MASCOTS proceedings, 2020, pp. 1–8.

[42] Y. Peng, Y. Bao, et al., "Dl2: A deep learning-driven scheduler for deep learning clusters," IEEE TPDS, vol. 32, no. 08, pp. 1947–1960, 2021.

[43] F. Filippini, M. Lattuada, et al. Hierarchical scheduling in on-demand gpu-as-a-service systems, in SYNASC2020, 2020, pp. 125–132.

[44] F. Filippini, D. Ardagna et al. ANDREAS: Artificial intelligence traiNing scheDuler foR accElerAted resource clusterS, 2021 8th International Conference on Future Internet of Things and Cloud (FiCloud), 2021, pp. 388-393, doi: 10.1109/FiCloud49777.2021.00063.

[45] Ansible Role for Kubernetes created by GRyCAP. https://github.com/grycap/ansible-role-kubernetes

[46] TOSCA Template to deploy InfluxDB using the Infrastructure Manager. Phttps://github.com/grycap/im-dashboard/blob/master/tosca-templates/k8s_influxdb.yaml

[47] LAMMPS. https://www.lammps.org

[48] MontecarloMultiGPu NVIDIA Sample. https://docs.nvidia.com/cuda/cuda-samples/index.html#monte-carlo-option-pricing-with-multi-gpu-support

[49] M. Caballer, I. Blanquer, G. Moltó, and C. Alfonso. "Dynamic management of virtual infrastructures". Journal of Grid Computing, vol. 13, no. 1, Pages 53-70, 2015, ISSN 1570-7873, DOI: 10.1007/s10723-014-9296-5.

[50] Cyfronet. https://www.cyfronet.pl/

[51] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, L. Zhang. Deep learning with differential privacy. In Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, (2016, October), pp. 308-318.

[52] C. Dwork, A. Roth. The algorithmic foundations of differential privacy. Found. Trends Theor. Comput. Sci., 9(3-4), pp. 211-407.

[53] C. Dwork. A firm foundation for private data analysis. Communications of the ACM, (2011), 54(1), 86-95.

[54] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, M. Naor. Our data, ourselves: Privacy via distributed noise generation. In Annual International Conference on the Theory and Applications of Cryptographic Techniques, (2006, May), (pp. 486-503). Springer, Berlin, Heidelberg.

[55] E. Lomurno, L. Di Perna, L. Cazzella, S. Samele, M. Matteucci. (2021). A Generative Federated Learning Framework for Differential Privacy. arXiv preprint arXiv:2109.12062.

[56] C. Xu, J. Ren, D. Zhang, Y. Zhang, Z. Qin, K. Ren. GANobfuscator: Mitigating information leakage under GAN via differential privacy. IEEE Transactions on Information Forensics and Security, (2019), 14(9), 2358-2371.

[57] J. Jordon, J. Yoon, M. Van Der Schaar. PATE-GAN: Generating synthetic data with differential privacy guarantees. In International conference on learning representations, (2018, September).

[58] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, D. Bacon. (2016). Federated learning: Strategies for improving communication efficiency. arXiv preprint arXiv:1610.05492.

[59] E. Bagdasaryan, A. Veit, Y. Hua, D. Estrin, V. Shmatikov. How to backdoor federated learning. In International Conference on Artificial Intelligence and Statistics, (2020, June), (pp. 2938-2948). PMLR.

[60] L. Lyu, H. Yu, J. Zhao, Q. Yang. Threats to Federated Learning. In Federated Learning, 2020, (pp. 3-16). Springer, Cham.

[61] J. Zhang, J. Zhang, J. Chen, S. Yu. Gan enhanced membership inference: A passive local attack in federated learning. In ICC 2020-2020 IEEE International Conference on Communications (ICC), (2020, June), (pp. 1-6). IEEE.

[62] T. Takahashi, S. Takagi, H. Ono, T. Komatsu. (2020). Differentially Private Variational Autoencoders with Term-wise Gradient Aggregation. arXiv preprint arXiv:2006.11204.

[63] AI-SPRINT Publications. https://www.ai-sprint-project.eu/media/publications

[64] R. Shokri, M. Stronati, C. Song, and V. Shmatikov.Membership Inference Attacks Against Machine Learning Models, in Proceedings - IEEE Symposium on Security and Privacy, 2017, pp. 3–18, doi: 10.1109/SP.2017.41.

[65] M. Fredrikson, S. Jha, and T. Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures, in Proceedings of the ACM Conference on Computer and Communications Security, 2015, vol. 2015-Octob, pp. 1322–1333, doi: 10.1145/2810103.2813677.

[66] Extrae. https://tools.bsc.es/extrae

[67] Paraver. https://tools.bsc.es/paraver