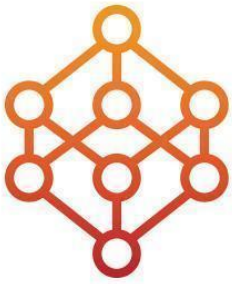

PHYSICS

oPTIMIZED HYBRID SPACE-TIME SERVICE CONTINUUM IN FAAS

D5.2 – EXTENDED INFRASTRUCTURE SERVICES WITH ADAPTABLE ALGORITHMS SCIENTIFIC REPORT AND PROTOTYPE DESCRIPTION V2

Lead Beneficiary	RHT
Work Package Ref.	WP5 – Extended Infrastructure Services with Adaptable Algorithms
Task Ref.	T5.1, T5.2, T5.3, T5.4
Deliverable Title	D5.2 – Extended Infrastructure Services with Adaptable Algorithms Scientific Report and Prototype Description V2
Due Date	2023-09-30
Delivered Date	2023-09-30
Revision Number	3.0

H2020 ICT 40 2020 Research and Innovation Action

This project has received funding from the European Union's horizon 2020 research and innovation programme under grant agreement no 101017047

Dissemination Level	Public (PU)
Type	Report (R)
Document Status	Final
Review Status	Internally Reviewed and Quality Assurance Reviewed
Document Acceptance	WP Leader Accepted and Coordinator Accepted
EC Project Officer	Mr. Stefano Foglietta

Contributing Partners

Partner Acronym	Role[1]	Name Surname[2]
RHT	WP and Tasks Leader	Luis Tomás Bolívar
UPM	Task Leader	Marta Patiño, Ainhoa Azqueta
RYAX	Task Leader	Yianins Georgiou
BYTE	Task Leader	Yiannis Poulakis
HUA	Task contributors	S. Tsarsitalidis, C. Giannakos, G. Kousiouris, V. Katevas, T. Kafatari, A. Lipitaki
INNOV	Task contributor	George Fatouros
ATOS	Task contributor	Carlos Sánchez
HPE, HUA	Reviewer	Domenico Dimitra
FUJITSU	Quality Assurance	Isabelle Ehresmann
	Revision	

[1] Lead Beneficiary, Contributor, Internal Reviewer, Quality Assurance
[2] Can be left void

Revision History

Version	Date	Partner(s)	Description
0.1	2023-07-01	RHT - Luis Tomas	ToC Version based on V1
0.2	2023-08-31	RHT - Luis Tomas	Modified version of executive summary, introduction, architecture prototype
0.3	2023-09-01	RHT - Luis Tomas	Initial modifications for Resource Management -- section 4.1)
0.4	2023-09-05	RHT - Luis Tomas	More modifications for Resource Management -- section 4.2), and Conclusions
0.5	2023-09-06	RHT - Luis Tomas	Modifications at sections 4.3, 4.4
0.6	2023-09-13	BYTE and HUA	Contributions to sections 2 and 3
0.7	2023-09-14	RYAX	Contributions to section 4 and 5
0.8	2023-09-18	UPM and HUA	Contributions to section 6
1.0	2023-09-20	RHT	Version for Peer Reviews
2.0	2023-09-27	RHT	Version for Quality Assurance
3.0	2023-09-28	RHT	Version for Submission

List of Abbreviations

API	Application Programming Interface
AWS	Amazon Web Services
CNCF	Cloud Native Computing Foundation
CNI	Container Network Interface
CR	Custom Resource
CRD	Custom Resource Definition
DNS	Domain Name System
eBPF	extended Berkeley Packet Filter
FaaS	Function as a Service
HPA	Horizontal Pod Autoscaler
IoT	Internet of Things
IRI	Internationalized Resource Identifier
K8s	Kubernetes
KEDA	Kubernetes Event-drivern Autoscaling
L3, L4, L7	Layer 3, Layer 4, Layer 7 (OSI)
ML	Machine Learning
OCM	Open Cluster Management
OKD	Community Distribution of Kubernetes that powers Red Hat's OpenShift
OS	Operating System
OW	OpenWhisk
PEF	Performance Evaluation Framework
QoS	Quality of Service
RHEL	Red Hat Enterprise Linux
VM	Virtual Machine
VPA	Vertical Pod Autoscaler

WP	WorkPackage

Executive Summary

The goal of this deliverable is to provide the description and implementation details of the PHYSICS components for the Extended Infrastructure Services with Adaptable Algorithms. This is the second/last version of this document and it covers the main components description and their interactions and API. In addition it includes some scientific experimentation outcomes. The different components are developed as part of the work package WP5.

The Extended Infrastructure Services with Adaptable Algorithms correspond with the infrastructure level and have Kubernetes (OpenShift [1]) as its centerpiece. It extends Kubernetes' provided functionality with extra APIs and with other components on top of that. The interactions between components, as well as with the infrastructure has been defined following the current upstream trends and best practices: using WebHooks, Custom Resource Definitions(CRD) [2] , Operators [3] and running containerized on top of the infrastructure itself. In addition, new components in this second release also made use of the new trends on event driven implementations, serverless computing, and energy metrics.

The infrastructure layer is divided into 4 main building blocks: a) the semantic model, b) the adaptable scheduling algorithms, c) the resource management controllers and interfaces, and d) the optimized co-allocation strategies. For each one of them, its design specification is described in this document, together with the main implementation and integration (API) details, and experimentation outcomes.

Summary of Changes

CHAPTER	UPDATE	SECTION(s)
Executive Summary	● Minor updates on the text	
1 - Introduction	● Minor updates on the text	1.1, 1.2, 1.3
2 - Architecture Prototype	● Minor modifications on the text. ● Added new components: Cluster Onboarding, Knative, VPA/HPA/KEDA, Kepler, … ● Better explaining/splitting of components ● Updated figures	2.1
2 - Architecture Prototype	● Minor modifications on the text ● Updated figures and text explaining them	2.2
4 - Resource Management Controllers and Interface	● Updated components subsection design ○ 4.1.a, 4.1.e, 4.1.f, 4.1.g ● Added new subsection for new components that are part of the architecture now (cluster onboarding, knative, kepler, and autoscalers) ○ 4.1.c, 4.1.d, 4.1.h, 4.1.i	4.1
4 - Resource Management Controllers and Interfaces	● Added figures for function registration/creation flow and update their explanation, and for cluster onboarding ● Updated components subsection implementation: ○ 4.2.a, 4.2.b, 4.2.c ● Added new subsections for new components: ○ 4.2.e, 4.2.f, 4.2.g	4.2
4 - Resource Management Controllers and Interfaces	● Updated section 4.3 and added subsections 4.3.a and 4.3.b ● Updated text for section 4.4	4.3 and 4.4
5 Adaptable Provider Level Scheduling Algorithms	● Updated introduction ● Updated section 5.2 and added subsections 5.2.a and 5.2.b ● Updated section 5.3 and added subsections 5.3a and 5.3b ● Updated section 5.4 and added subsections 5.4a and 5.4b ● Updated next steps	5.2, 5.3, 5.4 and 5.5
6 Optimized Service Co-location Strategies	Added new section 6.3 on the description of the profiling background process of the performance pipeline and according evaluation	6.3

3 Semantic Models for Service Characteristics	● Major text modification to match the current/final state of the component ● Various Figures removed added to reflect the new text	3.1- 3.4
6 - Optimized Service Co-location Strategies	● Updated component architecture and detailed sub-components description ● Added Interferences evaluation	6.1, 6.2
7 - Conclusions	Updated text	7

Contents

LIST OF TABLES

LIST OF FIGURES

LIST OF SOURCE CODE EXAMPLES

1. Introduction

The main aim of the PHYSICS Extended Infrastructure layer is to provide the functionality and interfaces (APIs) needed for enabling an optimized operation of the edge and cloud services, which in the PHYSICS case are used for the realization of the application service graph (i.e. the execution of its functions).

PHYSICS has selected Kubernetes as the cornerstone for the Infrastructure layer, extended with different projects to make it suitable for deploying applications in a multi-cluster setup (Open Cluster Management [4] and Submariner [5]), including special (low-footprint -> MicroShift, K3s, kind) clusters located at different edges.

In addition to ensure integration of those components by working together with their respective upstream communities, the work in WP5 focuses on the next action points to better support the execution of the application service graph:

- Semantic model to accurately depict and model each service and resource type capabilities and needs;
- Provider level Adaptive Scheduling Algorithm that maintains the QoS level by adapting to application needs;
- Optimized co-allocation strategies that minimize performance degradation enabling further exploitation of the available resources;
- Integration of the above items into the resource management layer, in this case Kubernetes. This includes the extra APIs and resources needed by the above components for interactions, in this case CRDs and Webhooks. It also provides the extra APIs needed for WP3 and WP4 mechanisms, such as cross-cluster orchestration or autoscaling mechanisms.

This deliverable presents the first version of the PHYSICS Extended Infrastructure architecture.

1.1 Objectives of the Deliverable

The goal of this deliverable is to define the final version of the infrastructure layer components and its interactions/APIs. It also presents the main scientific results and/or outcomes from them.

This deliverable describes the overall architecture of the infrastructure layer, with its main building blocks (semantics, scheduling, resource management, and co-allocation), and the interactions/APIs between them. It focuses on the design of those components and their implementation highlights, with focus on cross component integration (WP5 tasks).

This document is relevant for the design of the PHYSICS architecture (WP2), as well as for the interaction with other technical components developed in work packages WP3 (Functional and Semantic Continuum Service Design Framework) and specially in WP4 (Cloud Platform Services for a Global Space-Time Continuum Interplay). Note that the APIs exposed in this deliverable will be consumed by WP4 components. This deliverable is also useful for future adopters of the PHYSICS infrastructure layer platform, either as a whole or just for single components.

This deliverable presents the final version of the PHYSICS Extended Infrastructure Services with Adaptable Algorithms. The final design and implementation details of the different components, as well as their APIs, have been updated as the project progresses. It includes modifications from the initial design based on both new requirements as well as due to engagement with upstream communities.

1.2 Insights from other Tasks and Deliverables

The Extended Infrastructure Services of PHYSICS have been designed using as input the PHYSICS Architecture defined in deliverable D2.4. The infrastructure layer architecture also takes into consideration the inputs from upstream best practices regarding Kubernetes API extensions and integration, such as CRDs and Operators, as well as current trends on event-driven applications (based on Knative).

Although the deliverables D3.2 Functional and Semantic Continuum Services Design Framework, Scientific Report and Prototype Description V2 and D4.2 Cloud Platform Services for a Global Continuum Space-Time Continuum Interplay, Scientific Report and Prototype Description v2 are concurrent in time, they progressed in a coordinated manner. Several meetings were organized to define the interactions and APIs, especially in relation to WP4 components. These meetings provided very valuable information for the definition of the PHYSICS Infrastructure layer, its requirements and API extensions.

This deliverable provides input for the several WP6, Use Cases Adaptation, Experimentation, Evaluation, deliverables, regarding the integration of the prototypes (D6.2, D6.6) and the evaluation of it (D6.8).

1.3 Structure

The rest of the deliverable is organized as follows. First, an overview of the Infrastructure layer architecture, components and interactions is presented in Section 2. Then, the main building blocks (semantics, resource management, adaptable scheduling, and optimized co-allocation) are described in Sections 3 to 6, respectively. This includes information related to their design, implementation highlights, and scientific outcomes. Finally, conclusions are presented in the last section of the document.

2. Architecture Prototype

The main objective of WP5 is to provide a view and interfaces to enable optimized operations at different edges and the cloud services utilized for running the needed functions/workloads. To achieve this, the architecture prototype need to be able to:

- Accurately depict and model the abilities of each service, as well as resource type (e.g., nodes).
- Provide different adaptive and real-time provider-level schedulers so that current application needs are considered and overall QoS levels can be maintained.
- Incorporate those algorithms into the resource management and controlling layer, in this case Kubernetes/OpenShift clusters. This includes the relevant APIs so that it can be consumed by WP3 and WP4 components. Note that this also relates to the multicluster needs, such as extra configuration upon cluster onboarding.
- Improve the resource usage by providing co-location and optimization strategies, minimizing the performance degradation effects.

The architecture prototype focuses on both the ***single cluster components/extensions*** as well as the ***multicluster aspects***. All the above functionality needs to be incorporated in the infrastructure, providing the needed APIs for the upper components (e.g. those from WP4) so that applications (i.e. pods, functions, workloads) can be easily deployed across different Kubernetes/OpenShift clusters (including parts of the workflow on different clusters/edges). This also including the option to have edge clusters with limited resources and even different architectures (e.g. ARM), such as Raspberry PIs.

2.1 Main components

The main components of the architecture prototype can be differentiated depending on whether they are targeting the multicluster or the single cluster scenarios. Most of the tasks in this WP (all of them actually) focus on the single cluster scenario. However, some of them also cover the multicluster scenario, specially in its relation with the APIs provided to WP4. This is the case of tasks 5.1 and 5.3. Note that this does not mean the scheduling and co-allocation tasks are not run in multiple clusters, but they operate on them in an isolated mode from the other clusters. By contrast, T5.1 (semantics) needs to account for multicluster information, and T5.3 needs to provide the relevant APIs to components in other WPs (WP3/4), enabling spreading the load among clusters.

For the PHYSICS architecture prototype, we have based our design and implementation in both in-house solutions and already existing open source solutions. The next Figure shows the main components of the final architecture prototype which includes:

- Existing upstream projects (Kubernetes [6], Prometheus [7], OpenWhisk [8], Open Cluster Management [4], Submariner [5], Knative[9], Kepler[10]). We not only used them but also have contributed to some of them, in terms of finding bugs, fixing them, requesting future enhancements and related discussions, etc..
- New software tools developed within the project. They are either leveraging Kubernetes functionality as the base, such as webhooks, extending the Kubernetes API such as CRDs (Custom Resource Definition), enhancing K8s (such as the new scheduler) or independent of it, such as co-location engine or semantic engines.

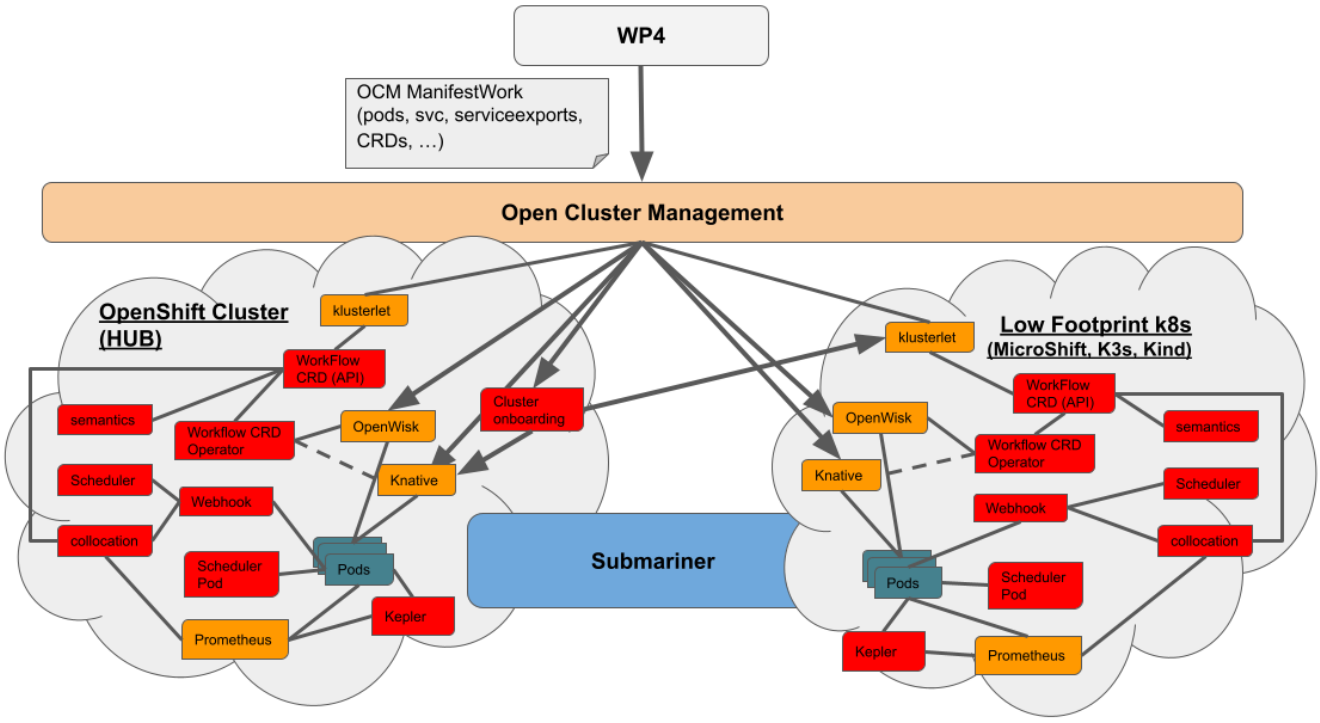

Figure 1. - Main components Overview

Let's cover the main functionality of the components by splitting them into multicluster and single cluster functional groups:

2.1.a Multicluster components

Open Cluster Management

It is a community-driven project [4] focused on the multicluster orchestration for Kubernetes applications. It is in charge of the clusters registration, work distribution across them, enabling dynamic placement policies and workloads. Thus, it allows the PHYSICS project to have a central Kubernetes cluster hub, from which it can manage all the managed Kubernetes clusters, as well as the applications running on them.

In PHYSICS it is used as the main API offered for WP4 to deploy workloads on the different available clouds, as well as to perform different configuration on the managed clusters (such as node scaling when possible). We work on its integration (agent side: Klusterlet) into the low footprint OpenShift (such as MicroShift or Kind, see section 2.1.2), to be able to leverage its functionality on resource scarce edges.

Submariner

Submariner [5] is an upstream (Sandbox) CNCF [11] project that targets to enable direct networking between pods and services deployed in different Kubernetes clusters, either on-premise or in the cloud. It is fully open source and designed to be network plugin (CNI) agnostic -- note that this does not mean it can work with all the CNIs, and some of them require specific drivers to be fully working.

There are other similar projects, like Skupper [12] that are also suitable for multicluster connectivity. However, Submariner joints the complete cluster (all namespaces), working on L3/4, while Skupper focus on L7 and simply joints applications. Both could be used but Submariner is more transparent to the PHYSICS components and helps to avoid the burden of having to create different proxies for each namespace that needs to be connected between clusters.

Therefore, in PHYSICS, Submariner is used to enable the distribution of pods and services across different clusters, yet enabling the connectivity between them. We focused on possible enhancements, such as support for ovn-Kubernetes CNI, integration into the open cluster management as well as with low footprint k8s distributions, such as MicroShift [13] or KinD [14] (see section 2.1.2). We are fully engaged in the upstream community (where Red Hat has the majority of contributions) and have reported issues and fixes as well as to engage on discussion about needs and use case, to make it a better product as well as to ensure it fulfills the PHYSICS needs [15][16][17].

Cluster Onboarding

This is a new component fully developed within PHYSICS, whose main objective is to automate the steps needed once a new cluster gets added to the multicluster setup. More specifically, it is in charge of installing some PHYSICS components on the new clusters and connecting them to the required components in the main hub. It also generates some artificial load for the semantic component, so that it is able to evaluate the new cluster. More details in Section 4.

This component makes use of serverless computing by leveraging Knative for event detection and processing. And it uses OCM to create the resources in the remote cluster and get the needed information about it. It can be easily extended to make extra configurations or deploy extra components. Its design can also be leveraged by other projects to make completely different configurations.

2.1.b Single cluster components

OpenWhisk and OpenWhisk Lite

Apache OpenWhisk [8] is an open source distributed serverless platform that executes functions in response to events, on containers. It supports different languages (such as Go, Java, NodeJS, Python, Rust, etc.) and allows developers to focus on the functional logic (called Actions) that run in response to events (Triggers) from external sources (Feeds).

In PHYSICS, it is used as the function as a service platform running on top of OpenShift, and it has been adapted to provide the needed functionality. There is also a lightweight version of it for running at small edges. In addition, as detailed in D4.2 and also in section 4 of this deliverable, we use a proxy behind it to ease the communication between the Workflow CRD operator and OpenWhisk.

Knative

Knative [9] is a Kubernetes-based serverless platform for building, deploying and managing serverless workloads. It is currently the leading upstream project related to serverless platforms and it has greatly evolved in the last couple of years, adding new functionality to its core, such as functions as a service capabilities.

Knative has two main components, Serving and Eventing, that helps to automate and manage tasks and applications. The serving part is in charge of running serverless containers in Kubernetes with simplified templates (one Knative service creates several k8s objects, such as pods, replica set, deployment, service and ingress/route) and allows scaling to 0. The Eventing part is in charge of managing the flow of events and redirecting them to the serving. It includes different types of events, such as sources, brokers and triggers.

We leverage Knative in the Hub/Main cluster for the cluster onboarding automation (both eventing and serving). In addition, a new controller for it has been implemented, for its integration with the workflow CRD Operator. This allows the workflow operator to be platform agnostic, being able to easily allow other FaaS/Serverless platforms, and helping leveraging other PHYSICS platforms components.

The present architecture with workflow CRDs allow to specify the platform for deployment. Due to the platform agnostic description of the workflow, we can easily allow other FaaS platforms to leverage the Physics platform components.

While an Openwhisk proxy facilitates the deployment and update of workflow descriptions into Openwhisk, a similar Knative proxy can be used to adapt the workflow description to Knative.

Autoscalers: HPA, VPA and KEDA

HPA, VPA and KEDA are different Kubernetes options for automatically scaling your workloads:

- HPA [18] stands for Horizontal Pod Autoscaling and automatically adjusts the number of pods (through a deployment or statefulset) in response to memory and/or cpu consumption.
- VPA [19] stands for Vertical Pod Autoscaling and automatically adjusts the resource limits [20] and requests for the containers in their pods. It therefore also only applies to cpu and memory. It can work in different modes, just giving you recommendations about the expected right values for the limits/requests or to also enforce those. It has a big limitation, which is that every time VPA updates the pod resources, the pod needs to be recreated.
- KEDA [21] stands for Kubernetes Event Driver Autoscaling. Unlike the previous ones that were based on cpu and memory consumption, this is based on the number of events being processed, therefore it provides a better way of scaling for event driven workloads (such as the Function as a Service target by PHYSICS). In addition, it already provides a large catalog of build-in scalers (such as for kafka queues) and new ones can be created for specific applications or use cases.

In PHYSICS, we have investigated the three of them and its extensions, such as using HPA with custom metrics (instead of just memory and cpu) as well as to write our own autoscaler plugin for KEDA.

Prometheus

Prometheus [7] is an open source CNCF (graduated) project focused on system monitoring and alerting. It collects and stores its metrics as time series data and it is adopted as the main monitoring tool on many projects/orchestrators, such as OpenShift.

In PHYSICS we leverage information provided by Prometheus from the Kubernetes clusters to improve the scheduling and co-location decisions over time, depending on the status of the system as well as previous executions of the applications/pods. In fact, Prometheus is broadly used across CNCF projects and thus, tools such as Openwhisk, facilitates an Openwhisk exporter that populates metrics into Prometheus. This feature is utilized by several components in WP3 and WP4, and the same happens for Kepler, making use of Prometheus to export the metrics related to Energy Consumption and make them available for other PHYSICS components (such as the semantic component).

Low Footprint K8s: MicroShift, K3s, Kind

When running workload in the Edge it must be taken into account that the available resources may be limited. In this respect, we evaluated and tested different Kubernetes distributions that are suitable for it: MicroShift, K3s and Kind.

MicroShift [13] is a new, experimental flavor of OpenShift/Kubernetes optimized for edge devices use cases. It targets low footprint resources, such as single node deployments or Raspberry PIs. It is designed to be executed on top of Fedora IoT and RHEL for Edge systems, leveraging the OS's capabilities for secure device on-boarding and system configurations/upgrades. It aims to be secure and resilient to adverse networking conditions as well as a very low resource footprint. In PHYSICS, we are working on integration of MicroShift with Open Cluster Management as well as Submariner. We are also working on building a community around it, so that the project gains some momentum and starts to be used as a solution for low footprint devices on the edge.

In addition, and for testing purposes we are also using Kind, a tool for running a small Kubernetes cluster using docker container nodes. Although it is primarily designed for testing, it is also useful to emulate small edge devices as well as multicluster integration. We have worked in PHYSICS in its integration with OCM and Submariner, engaging with their communities to solve the small issues found.

Semantics

The Semantics component is fully developed as part of PHYSICS. It is in charge of gathering information about the systems and building the ontologies and knowledge base that other components leverage, mainly the Reasoning Framework from WP4.

The component is deployed in each cluster as a dockerized Python application along the cluster onboarding process. Through a series of methods and provided with the necessary permissions, it retrieves functional and non functional cluster information by communicating with the Kubernetes API and prometheus. The latter also contains the relevant energy and performance metrics that Kepler provides. This information is afterwards translated according to the designed ontology, that captures the domain specific knowledge on cluster components and their descriptions. Finally it is propagated as N-triples to the reasoning framework for the data to be ingested and utilized for application to resource matching.

Additionally, it includes several service endpoints so the cloud dev administrators can interact with the semantics in a reliable manner for debugging and retrieving or annotating information directly. These endpoints include checking that the semantics component can successfully connect to the Kubernetes API, manual annotation of triples etc.

Workflow CRD (API) and its Operator

Custom Resource Definitions (CRDs) [22] is the standard way defined by Kubernetes to extend the Kubernetes API. We have defined a new one, named Workflow CRD, for the interactions with WP4, as the main API offered to its components. It stores all the needed information for registering functions, as well as extra parameters that are needed by other components such as the co-location engine.

The CRD operator is another component fully developed for the PHYSICS project. It is a new Kubernetes operator that will be in charge of reacting to Workflow CRD objects creation and fill in their information with the status information. Depending on the spec information of the Workflow object, some actions will be triggered, in this case the registration of OpenWhisk functions (by calling the OpenWhisk proxy developed in WP4) or Knative (serverless) services/functions -- by stating the target platform.

In addition, this Workflow CRD in coordination with statusFeedback from OCM serves to provide details on the workflow from the edge to the hub cluster without the need of additional tooling.

Webhook

The pods created by OpenWhisk need to be updated so that the proper scheduler and affinities as specified before Kubernetes process the pod. To this end we develop a Kubernetes Webhook, which:

- processes the labels added by OpenWhisk on the pod definition.
- selects the right scheduler to be used.
- selects the affinities/antiaffinities (see next two subsections).
- modifies the pod object accordingly before it is stored on the Kubernetes ETCD database [23], and therefore processed by Kubernetes operators.

Scheduler and Scheduler pod(s)

In Kubernetes the scheduler has 2 phases, filtering plus weighting, and they can be done based on different parameters/metrics available, such as the cpu and memory, or nodes labels, or even container images presence.

In PHYSICS, we have developed a new image layer aware scheduler that makes use of new information about the container images layers in the nodes, to speed up the boot time of containers by selecting the node with more layers already downloaded -- thus minimizing the cold start problem for functions.

This means not only the presence or not of the whole container image is considered, but also the presence of their different layers. This is important for FaaS use cases as there may be a lot of similar images that

share most of the common layers. The proposed scheduler will noticeably reduce the cold start problem in these cases, unlike existing scheduling mechanisms which are not aware of the container image layers. Note there are still ongoing efforts to fully upstream this scheduler and make it part of the default Kubernetes installations.

The scheduler component is another component developed for the PHYSICS project. It is integrated into the Webhook (see previous subsection) and it simply selects the scheduler type to be used for that pod. There are different types of schedulers running on the system as pods. As highlighted in the previous section, the webhook executes the scheduler logic to select the type of scheduler (i.e., scheduler pod) to be used.

Co-location

Similarly to the scheduler, the co-location component is another component developed for the PHYSICS project that will be integrated into the Webhook. This enables the addition of affinities/antiaffinities on the pods. This component also makes use of the information present on the WorkFlow CRD to better decide on the co-location strategy for a given set of functions (run as pods).

This component receives information regarding the usage of resources of the function (CPU, RAM, network and storage) and hardware needs. This information is provided at design time (WP3) and also during the performance evaluation component (WP4). The component keeps information regarding resource consumption of the pods running in the cluster and excludes some nodes for deploying the pod (the hardware needed is not present, there are no resources available) and recommends a subset of suitable pods where the pod can be deployed.

Kepler

Kepler [10] stands for Kubernetes-based Efficiente Power Level Exporter. It is a Kubernetes exporter that uses eBPF to probe CPU performance counters and Linux kernel tracepoint. Then it uses that data, together with cgroups data to deed Machine Learning models that estimate the energy consumption by Pods.

In PHYSICS we have been greatly involved in its upstream development, engaging with its upstream community and reporting important issues that blocked their usage in nested environments (i.e., running on top of VMs in, e.g., AWS or Azure). We have worked together with the community in demonstrating and finding the issue as well as testing solutions for it -- more information in section 4.2. In addition we have evaluated the accuracy of the ML model by comparing the obtained metrics by it with the real metrics obtained when running on Grid5000 [24].

In PHYSICS we have integrated Kepler by making use of its metrics (through Prometheus) as part of the cluster onboarding and the semantic component to estimate the energy scores for the new clusters.

2.2 API and Interactions

After defining the main components, let's describe how these components are connected/associated together. The focus is for the interactions between the different components in WP5 and the extended infrastructure/APIs (part of T5.3). For the interactions across WPs you can check the architecture deliverable (D2.4).

The next diagrams describe the interactions in the context of a single cluster. Multicluster interactions should be agnostic on whether if there is one or more clusters in the platform, this behavior is a consequence to the fact that the underlying layer that provides the multi-clusters connectivity (Submariner) and API (Open Cluster Management), which make it transparent as long as we use services IPs or the services domain name associated with the Submariner exposed service-domain -- when services are exposed through Submariner, so that DNS resolution works across cluster too.

2.2.a Semantic API and Interactions with Resource Management layer

The semantic engine captures clouds and service's functional and non-functional requirements, capabilities and characteristics, such as service types, hardware needs, scaling parameters, energy consumption, etc. The semantic component additionally provides an API for the WP4 components (available through Submariner in the multicluster scenario), more specifically for the Reasoning Framework. These WP4 components will use this information to improve the cluster selection for future functions/workloads. In addition, to provide an API that other (Kubernetes) components can easily leverage, PHYSICS makes use of the Kubernetes Custom Resource Definition (CRDs). This extends the Kubernetes API with specific objects, in our case the Workflow CRDs created from the WP4 ecosystem (OCM), which contains extra information needed by other PHYSICS components.

The main interactions are captured in the next Figure.

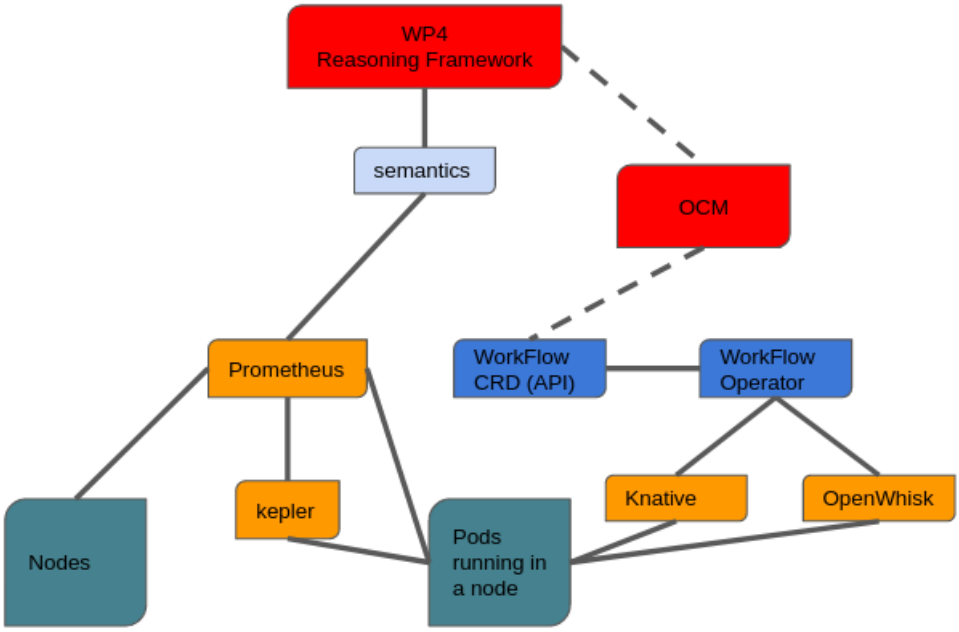

Figure 2. - Semantic component Interactions (dotted line means indirect interaction)

As it can be seen, the semantic engine obtains information from Prometheus about both the cluster resources and about the energy consumption of (some) pods -- made available through Kepler. This information may then also be part of the workflow CRD objects which in turns can be used by other PHYSICS components, in this case the Co-location engine to make better decisions about where to locate the new pods to be created (see section 2.2.c). Note that the workflowCRD object is generated in WP4, by using the OCM and leveraging the information provided by the semantic component to the Reasoning Framework. More details on WP4 related workflow can be found in D4.2.

Initially, a node CRD was envisioned (in the first prototype version), but after the initial phase it was decided that it won't be needed and the Reasoning Framework component could consume directly the information from the semantic component instead -- thanks to Submariner.

2.2.b Scheduling APIs and Interactions with Resource Management layer

The support for multiple intelligent scheduling algorithms for a more efficient resource sharing is integrated into the system through Kubernetes webhooks and different scheduler pods:

- Each scheduler logic should be running in a pod in the Kubernetes cluster. There may be schedulers focused on energy efficiency, while others more targeted to the FaaS platform, considering things like image layers presence in the different nodes, or performance interference

(including information coming from co-location and semantic engines). In PHYSICS, we have developed a prototype that implements a CacheLocality scheduler which takes into account not only the images available in the nodes, but also their layers. More details on this scheduler are in Section 5. Example information about how to run different schedulers on Kubernetes can be found in [25].

- A Webhook is created so that different schedulers can be selected for different pods. The way webhooks work in Kubernetes is explained with more details in [26], as well as in section 4. For the PHYSICS case, the process is as follows (also depicted in Figure 3):
    1. Pod object creation request arrives to the K8s API.
    2. The Webhooks kicks in, fetches the pod object and performs certain actions on it. In our case, the scheduler engine gets executed and analyzes the pod annotations (made by the OpenWhisk and WP3/4 components, regarding the workflow it belongs to, and other extra information to be used, such as particular constraints or objective scores coming from the global continuum placement component, as mentioned in deliverable D4.1). It also includes the information about the scheduler to use, and if none stated it used the new CacheLocality scheduler.
    3. The webhook changes the pod object definition to include the scheduler to be used:

```
apiVersion: v1
kind: Pod
spec:
  schedulerName: physics-scheduler
  containers:
  ...
```

Code 1. Pod template with specific scheduler

    4. The modified pod object gets stored, after the webhook modifications, into the ETCD Kubernetes DB.
    5. The scheduler pod sees there is a new pod without an associated node, and executes its logic to decide the node it should be scheduled to.

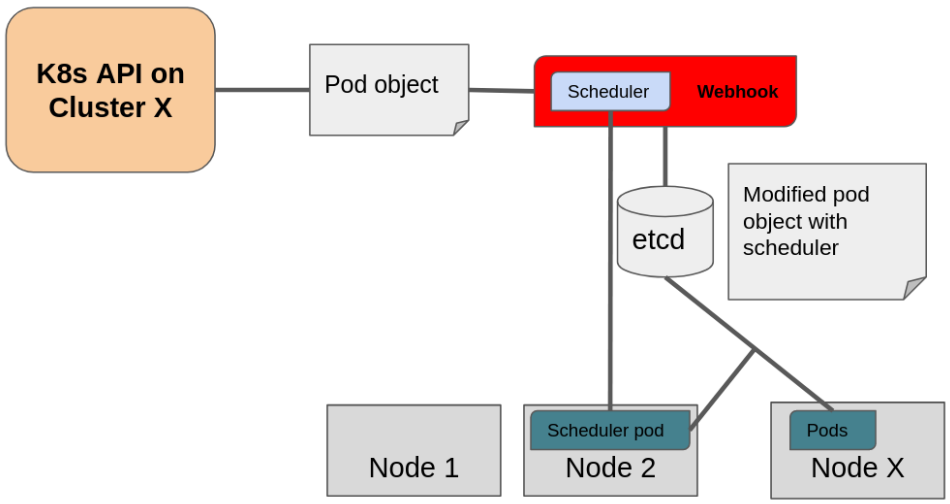

Figure 3. - Scheduler component(s) Interactions

Each scheduler (pod) may use different information to perform its decisions. For the PHYSICS project, the CacheLocality scheduler uses node information about the existing container image layers.

2.2.c Coallocation APIs and Interactions with Resource Management layer

The coallocation engine integration is similar to the scheduler case. It focuses on the definition of affinities and antiaffinities rules. It is integrated into the webhook as shown in Figure 4, by having a second round of pod object modification to also add the affinity information to the pod object. This information needs to be considered by the scheduler pod in their filtering (usually Kubernetes schedulers divide its execution in 2 phases, filtering out the nodes that do not satisfy the requirement, and then weighting in the remaining nodes by some factor).

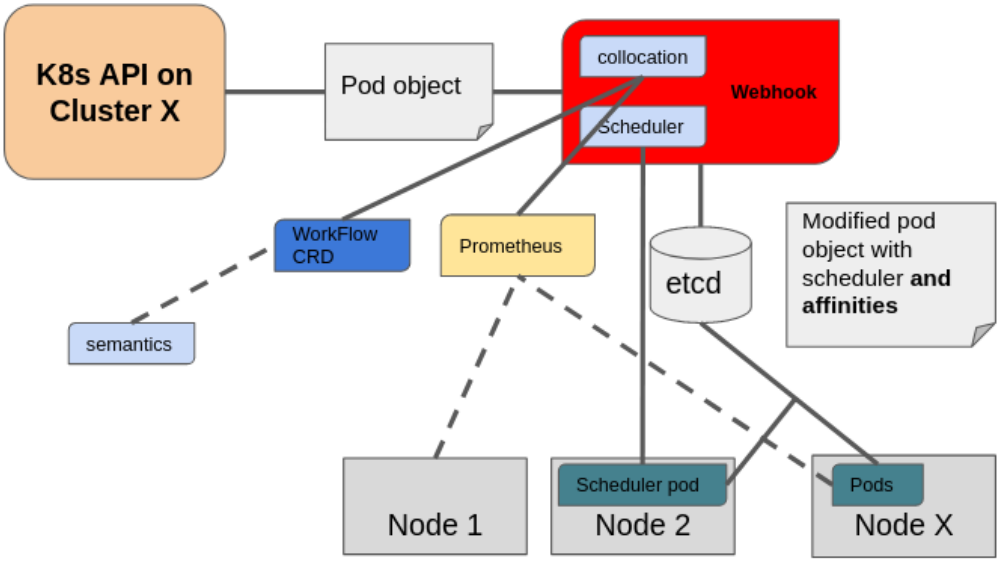

Figure 4. - Co-location components Interactions

Unlike in the scheduler case, here the coallocation part of the webhook needs access to the Workflow CRD information in order to select the right affinities -- while in the scheduler case that may be done only at the scheduler pod, not at the webhook phase if the information in the annotations is enough. Of course, it can also be used if the scheduler type should be selected depending on the workflow semantics.

The final prototype was enhanced by making use of the information stored in the WorkflowCRD, that comes from the performance profiling done in WP3/4. The co-location engine has been enhanced so that affinities are calculated based on the performance expectations of the functions as well as their possible interferences with other already running functions. To achieve that, it accesses local information stored in the local cluster (Prometheus) as well as information related to the functions profiling through the workflowCRD object.

3. Semantic Models for Service Characteristics

The goal of this task is twofold; (a) to design an ontology that acts as the basis for semantic representation of resources and their capabilities, and (b) to develop the necessary endpoints and functionalities to extract and transform such information to ensure that it is in accordance with the aforementioned ontology model.

Designing the aforementioned ontology sets the standard for PHYSICS to capture information on resources and along with the application semantics (T3.1), enables semantic reasoning in the inference stage (T4.1), matching resources to applications according to their specifications. For the remainder of this deliverable, these three components, when they need to be referenced together, will be referred to as the "Semantics Block".

Examples of specifications for the service semantics to capture are the locality of deployment, GPU enablings, SLA and more functional and non- functional capabilities. It should be noted that resources in this sense refer to internal project based services, edge services and services provided by external public providers with either computing or storage capabilities. In addition, information from the resource ontology will be available to be leveraged by other components in the work package, described later on in the present deliverable.

Many of the concepts described in this section that refer to the Service Semantics component have also been documented in a relevant publication focused around the ontology that has been designed as part of the PHYSICS project [27].

3.1 Design Specification

3.1.a Domain Specific Language

The first step towards component realization is choosing a domain specific language that will semantically describe the ontology. Several standards and such languages have been introduced to enable the creation of ontologies. TOSCA [28], CAMEL [29] and OWL [30] are all viable options for semantic representations, all of them utilized in a variety of different contexts such as the semantic web. The domain specific language of choice for T5.1 Semantic Models for Service Characteristics is OWL, which has been chosen as the main domain specific language due to its flexibility, popularity and the general consensus among the components included in the "*Semantics Block".*

OWL ontologies are essentially RDF graphs that consist of triples. These triples are composed through the usage of the following components:

- **Classes:** Represent objects that are to be described.
- **Data Properties:** Characteristics of the classes.
- **Object properties:** Indicate relationships between classes.
- **Datatypes:** Indicate the data type for a data property.
- **Individuals:** Instantiated classes with actual information.
- **Annotation properties:** Meta Characteristics of all the previous types.

Furthermore, a unique IRI (Internationalized Resource Identifier) has been assigned to the ontology and is used as a prefix in ontology objects. This unique identifier plays a key role when importing ontologies, allowing the source origin of classes to be easily distinguished..

The normative exchange syntax for OWL is composed of XML and RDF. Although it is not in the scope of this document to describe all the XML/RDF and OWL vocabularies and syntax, certain examples are provided to visually aid the reader in understanding the proposed standard. Examples of the XML/RDF syntax are presented below, while visual examples of the classes are available in the section "Implementation and Integration highlights".

The first XML block below showcases the definition of the ontology while the second gives an overview of how a Class is defined. The latter two indicate the syntax for creating the object and data properties respectively.

```
<!-- Example of ontology iri specification, addressed at the top of the ontology -->
<?xml version="1.0"?>
<rdf:RDF xmlns="http://www.physics-h2020.eu/physics/"
     xml:base="http://www.physics-h2020.eu/physics/"
     ....
    <owl:Ontology rdf:about="http://www.physics-h2020.eu/physics/"/>
```

Code 2 - Example of ontology IRI specification

```
<!-- Example of class definition in the ontology -->
<owl:Class rdf:about="http://www.physics-h2020.eu/physics/EdgeDevice">
        <rdfs:subClassOf rdf:resource="http://www.physics-h2020.eu/physics/Node"/>
    </owl:Class>
```

Code 3 - Example of class definition

```
<!-- Example of object property definition in the ontology -->
<owl:ObjectProperty rdf:about="http://www.physics-h2020.eu/physics/isHostedOn">
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
        <rdfs:domain rdf:resource="http://www.physics-h2020.eu/physics/Cluster"/>
        <rdfs:range rdf:resource="http://www.physics-h2020.eu/physics/Resource"/>
    </owl:ObjectProperty>
```

Code 4- Example of object property definition

```
<!-- Example of data property definition in the ontology -->
<owl:DatatypeProperty rdf:about="http://www.physics-h2020.eu/physics/architecture">
        <rdfs:subPropertyOf rdf:resource="http://www.physics-h2020.eu/physics/so"/>
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
        <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    </owl:DatatypeProperty>
```

Code 5 - Example of data property definition

3.1.b Service Semantic Models in the PHYSICS ecosystem

The service semantics component is deployed in each cluster managed by the PHYSICS. More specifically when the cluster onboarding (section 4.1 , Task 5.3) process takes place for a new cluster, a benchmark application is deployed and when it is finished the service semantics component is notified of its execution and the pod's name. It then starts a pipeline of methods that retrieve, semantically transform and finally send to the Reasoning Framework (T4.1) the cluster's information. The whole process is visually represented in Figure 5.

Firstly, the component retrieves information from the cluster by performing various queries to the Kubernetes API and the Prometheus endpoint, which also consumes metrics from Kepler. Specifically, Kepler is responsible for providing the energy consumption and performance metrics of the benchmark pod that is used to generate cluster scorings. Afterwards, the component transforms this raw information into semantic representations which ensure that semantic rules are followed. Certain aspects of this information are inferred logically from a set of rules. Examples of this are the total energy used by a benchmark pod and the locality of the cluster based on the architecture of the nodes cpu units. Finally, the ontology that consists of the individuals is propagated to the Reasoning framework through Submariner and the triplets are stored in the AllegroGraph database. This database is a graph database that enables reasoning over the ontology and execution of SparQL queries.

The Service Semantics component is deployed as a service in each cluster and as such it incorporates a number of endpoints that allow for users, usually cluster administrators, to interact with the component in various ways. One such important feature is the manual annotation endpoint to which the user can add triplets directly in the ontology that describes the cluster.

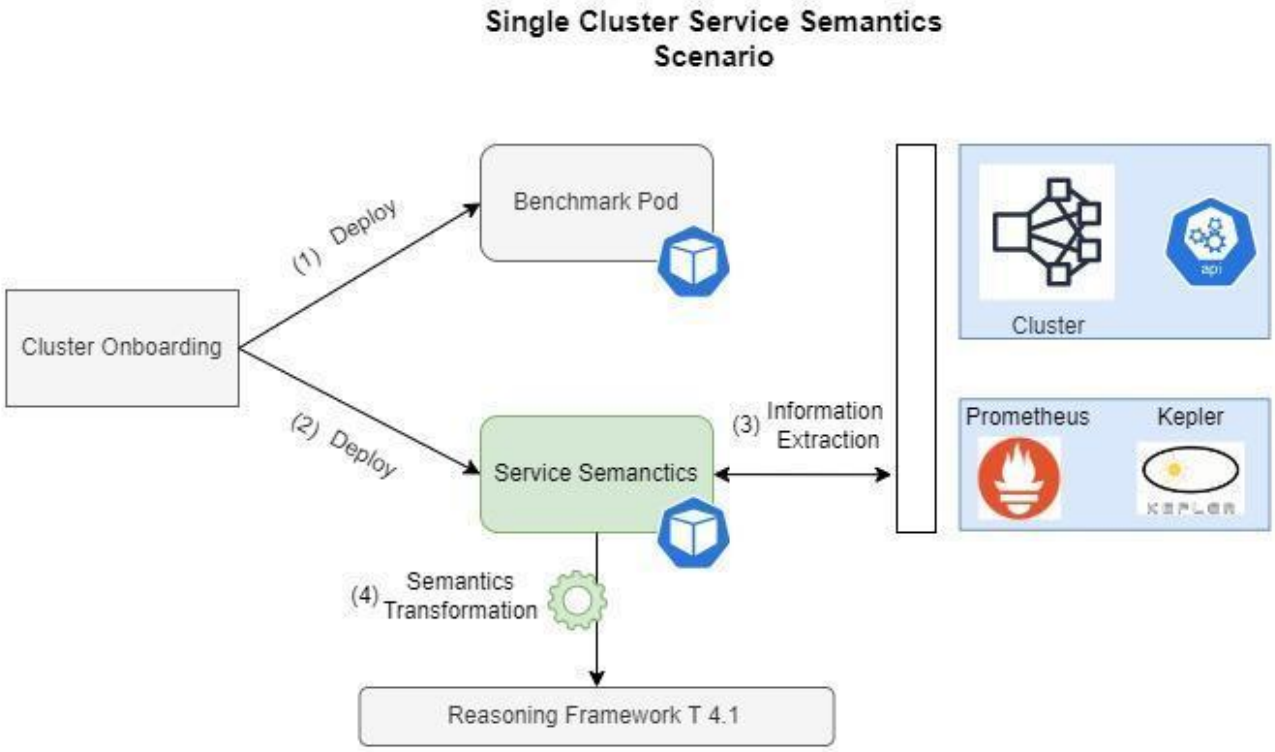

Figure 5. - Semantic Models for Service Characteristics in the PHYSICS architecture

In the scope of PHYSICS, semantics information is utilized in an automated manner in the reasoning framework. In different situations, where an actual user would be necessary to have immediate access to the information, two cases can be foreseen; For a single cluster the user can retrieve the ontology of the cluster by the service semantics endpoint. If the user needs to retrieve information for all the managed clusters and/or the application deployed, they can use SparQL queries through the Reasoning Framework.

3.2 Implementation and Integration Highlights

The ontology is built using Protege [31]. It is a staple Java-based framework in building ontologies using OWL syntax and provides a graphical user interface for doing so. It also allows for the export of ontologies in various formats. Although it is visually impossible to include the view of the whole ontology. In Figure 6, we include a preview of some of the included classes, using the Protege builtin add-on OntoGraph. For a more elaborate view, we refer to WebVowl, a web-based ontology visualization interactive tool.

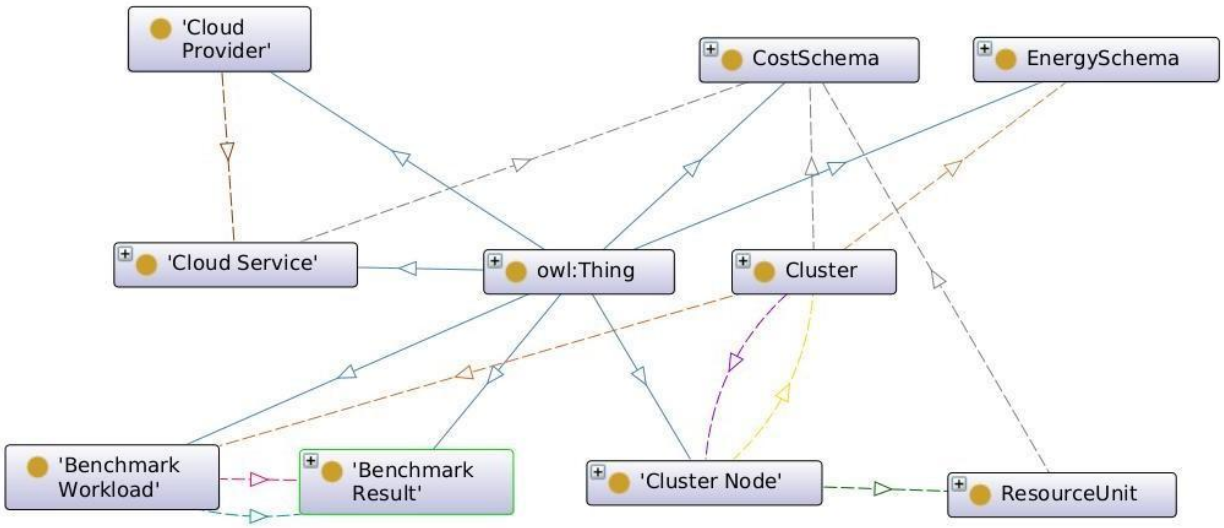

Figure 6. - A sample of ontology classes visualized through protege

The component is built using Python as the programming language of choice and some front-end languages and elements with HTML, CSS and JS along the Flask web framework [32] to realize its functionalities. Owlready2 [33] and OntosPy [34] are two key Python libraries that allow the handling of ontology data. They provide the necessary methods to create individuals and manage aspects of the ontology, such as deleting and creating classes and properties.

Additionally, Kubernetes-api-client and prometheus-client are two libraries that allow the service to extract information from the respective endpoints. The first one allows us to retrieve information about the cluster and the nodes. Additionally, it enables the component to discover the Prometheus in-cluster IP and therefore perform queries to it. The latter allows us to access Prometheus and retrieve information about the benchmark pod.

A couple of rules have also been developed that allow the service to infer semantic information directly from the available information. Examples of such cases include the energy related metrics, in which case it is ensured that they are all measured using the same unit of measurement and are summed to provide a new attribute totalEnergyConsumption. Another includes the identification of edge nodes according to the CPU architecture, in case this is not explicitly stated in the ontology. Almost certainly all edge nodes, such as raspberry pi, employ ARM based CPU which allows us to identify them in an inferred way.

The component is packaged as a dockerized service which is deployed along with the necessary permissions to list and get Kubernetes resources such as nodes and services. Endpoints and parameters are discussed further in deliverable 6.2.

3.3 Experimentation Outcomes

Due to the nature of the services provided by the task, the outcomes are difficult to be quantified in order to be assessed. First, we provide some quantifiable measurements of the ontology that allow for a more thorough examination. We report the ontology components to be 69 classes, 18 object properties and 39 data properties. These numbers indicate an ontology that consists of many subclasses , is not densely connected and contains classes with many properties.

A representative example of this is the benchmarkResult class contains many subclasses of results like cpu_cycles, execution_time etc. but each is only connected with the benchmarkWorkload class.

We have opted to include a couple of ontology related metrics that provide some insights on some of its aspects, but should be interpreted with caution when used to compare different ontologies. Namely, the two metrics are (a) **Relationship Richness (RR)** and (b) **Attributes Richness (AR).** Both are defined as follows :

$$RR = \frac{|P|}{|SC|+|P|} \quad (1)$$

and,

$$AR = \frac{|ATT|}{|C|} \quad (2)$$

The Relationships Richness (Equation 1) metric is defined as the ratio of existing relationships in the ontology divided by the number of subclasses and relationships. It indicates the diversity of relationships of the ontology and we report this value to be 0.68 for our ontology. The Attributes Richness (Equation 2) is defined as the ratio of all data properties (attributes) by the number of classes. It is an indicator of the amount of data properties that are tied to classes on average and provides some insight on how densely information is linked to classes. For our ontology, we report this number to be 0.57. We also report these values to be 0.55 and 0.65 for another IT related ontology [35] as point of reference.

In terms of execution time, the component is relatively lightweight. As tested on two of the available to the project clusters, one hosted on AWS and one on Azure, the response time to receive the semantics service's output is under a minute. In actuality the component is only affected in terms of its execution time by the number of nodes of the cluster it is deployed, in which case even we deem these changes insignificant as some operations are performed not matter the number of nodes and the transformation of an extra node's information retrieved through Kubernetes API is still very fast.

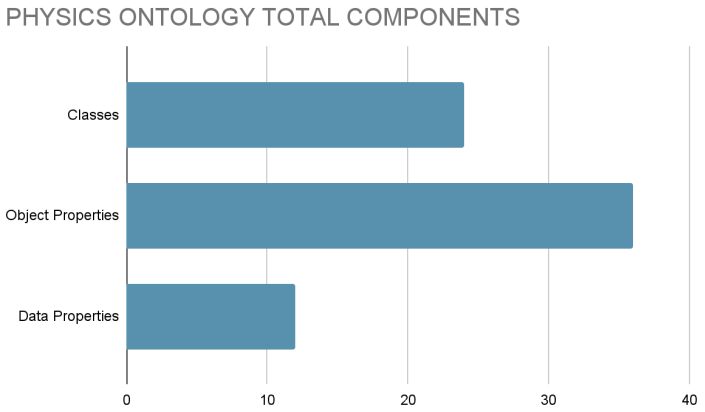

Figure 7. - Total number of the ontology different components.

3.4 Next Steps

In the previous sections we presented the design and various aspects of the implementation and integration of the component. We also aspire for the future continuation and support for the component after the project's lifespan has come to an end. Both the ontology and the actual service that extracts cluster level information are open sourced through GIT projects and the PHYSICS ramp. As such, we expect future usage of the component and its included functionalities to point towards potential improvements to the next guide future development according to needs. Especially among ontologies, it is a common practice to reuse them either as standalone or by importing them into other ontologies which can be connected semantically.

As it currently stands, we expect that improvements could be made by developing additional methods that support various information extraction methods. One such domain could be potentially using natural language processing (NLP) to automatically extract SLA information from the official documents provided by each cloud vendor. Of course, this is not a straightforward task and it remains to be seen whether future services will be developed that need to reason upon such information depicted in a standard format.

4. Resource Management Controllers and Interfaces

This section (related to Task 5.3) focuses on the infrastructure layer, both at single cluster (i.e., Kubernetes/OpenShift layer) and multicluster (set of Kubernetes/OpenShift clusters). It is in charge of providing the needed APIs for the upper layers (WP4 components) and the components developed in this Work Package (Semantics, Scheduler and Co-allocation engines), so that they can better control both the infrastructure itself and the applications running on top. This means offering the functionality and APIs to be able to perform wiser scheduling and co-location decisions, as well as enabling applications deployment across different clusters, including cluster onboarding processes.

As a result, this task also focuses on adding the missing functionality at the infrastructure layer to be able to support the PHYSICS architecture. When possible, this functionality is added to already existing upstream projects, increasing the impact, reusability, and innovation achieved (for example to core Kubernetes or Kepler). And for the components that do not have a clear fit upstream, we develop them so that they are well integrated into the Kubernetes ecosystem, by leveraging the tools that Kubernetes provides for such scenarios. For example by using Custom Resource Definition and the Operators Model to extend Kubernetes API -- OpenShift is based on this to extend the Kubernetes API and have controllers able to manage the cluster itself, the CNI, the monitorization, etc; or by using the new Serverless framework, in this case Knative, to manage event driven actions, such as the cluster onboarding.

4.1 Design Specification

As previously highlighted, the cornerstone of the Infrastructure layer is Kubernetes. However, this is not enough for PHYSICS needs as it does not cover the multicluster bits, nor the low-footprint with central orchestration/management. Also, it does not come with the needed extra hooks for scheduling/co-allocation options to make a more efficient usage of the resources.

We have followed different approaches to take each one of the above issues, as previously highlighted in Figure (Main Components Overview):

- For the **multi cluster** problem, in PHYSICS we researched the upstream options and decided to base our architecture in two upstream projects. First one is **Submariner**, which provides connectivity and discovery of pods and services across clusters. The second one is **Open Cluster Manager,** which allows onboarding of new clusters and central management of them, both in relation to the cluster configuration as well as related to running applications on them. In addition we have developed a **cluster onboarding mechanism** [36], based on Knative for both event detection and service (serverless), which performs remote configurations on the joined cluster upon onboarding event.
- For the **edge** problem, PHYSICS (in this case Red Hat) has started a new project, named MicroShift, to provide a minimal, low-footprint OpenShift binary that can be used to deploy small clusters at the edges, including IoT devices such as Raspberry PIs. In addition we have studied other options, such as K3s or KinD -- and used KinD for testing purposes and edge emulation.
- For the **integration of semantics, adaptive schedulers and optimized co-allocation strategies**, PHYSICS has extended the Kubernetes API with **CRDs** and a **Webhook**. The first one (CRD) allows special types of objects behind the Kubernetes API (in this case with information about the Workflows), and therefore provides a perfect mechanism for the different PHYSICS components to interact and communicate -- for example for the co-location to get the needed profiled information about the functions to be created. The latter (Webhook) allows the scheduler and co-allocation strategies developed in this WorkPackage to be executed at the right moment, before the pods are scheduled with the default Kubernetes scheduler, without any extra co-allocation hint.

- Finally, for energy consumption information we have evaluated different upstream options to identify the ones that are more suitable for PHYSICS use cases, as well as evaluated how responsive their communities are. We evaluated Scaphandre and Kepler. Even though Scaphandre looked a bit more mature, it had some gaps that were blocking its usage. We decided to go for Kepler, seeing the community was much more reactive.

Next we cover more details about each one of those components.

4.1.a Submariner for Multicluster networking

To allow the connectivity between applications (pods and services) deployed at different Kubernetes clusters, PHYSICS is using the Submariner upstream project.

Submariner architecture have 4 main components:

- Gateway Engine: manages the secure tunnels to other clusters, by default IPSEC tunnels. It is deployed in the selected Gateway Node in each cluster.
- Route Agent: routes cross-cluster traffic from nodes to the node with the active Gateway Engine.
- Service Discovery: provides DNS discovery of Services across clusters.
- Broker: facilitates the exchange of metadata between Gateway Engines enabling them to discover one another. Note, unlike the other components, this only needs to be installed in one cluster (the central/hub one).

This provides out of the box IP connectivity, and it provides the APIs to provide service discovery (DNS) across clusters -- deciding what services to expose. This is the API needed by upper layers in case DNS is required. There are pros and cons of using the IP connectivity or the DNS/submariner service. Using the service IP is simpler and it does not depend on the extra service discovery mechanism of submariner. However it may be impacted if the service gets recreated with a different IP (it can easily be managed in the application layer though). This is what the DNS option covers, at the expense of having to take care of that service exposition on the application (instead of IP discovery).

4.1.b Open Cluster Management for Multicluster management

To allow the centralized management and configuration of clusters, as well as the deployment of applications on them, PHYSICS selected the Open Cluster Management upstream project. Open Cluster Management (OCM) is a powerful, modular, extensible platform for Kubernetes multi-cluster orchestration. Unlike previous efforts trying to bring the Kubernetes federation, OCM tries a different approach. It embraces the "hub-agent" model. In OCM, the multi-cluster control plane is modeled as a "hub" and on the other hand each of the cluster being managed by the "Hub" will be a "klusterlet":

- **Hub Cluster**: cluster that runs the multi-cluster control plane of OCM. It is supposed to be either light-weight cluster hosting merely a few fundamental controllers and services, or have 2 roles -- hub cluster and klusterlet-- so that workloads can also be executed on it.
- **Klusterlet**: clusters being managed by the hub cluster, also called “managed cluster” or “spoke cluster”. The klusterlet actively pulls the latest prescriptions from the hub cluster and consistently reconciles the physical Kubernetes cluster to the expected state.

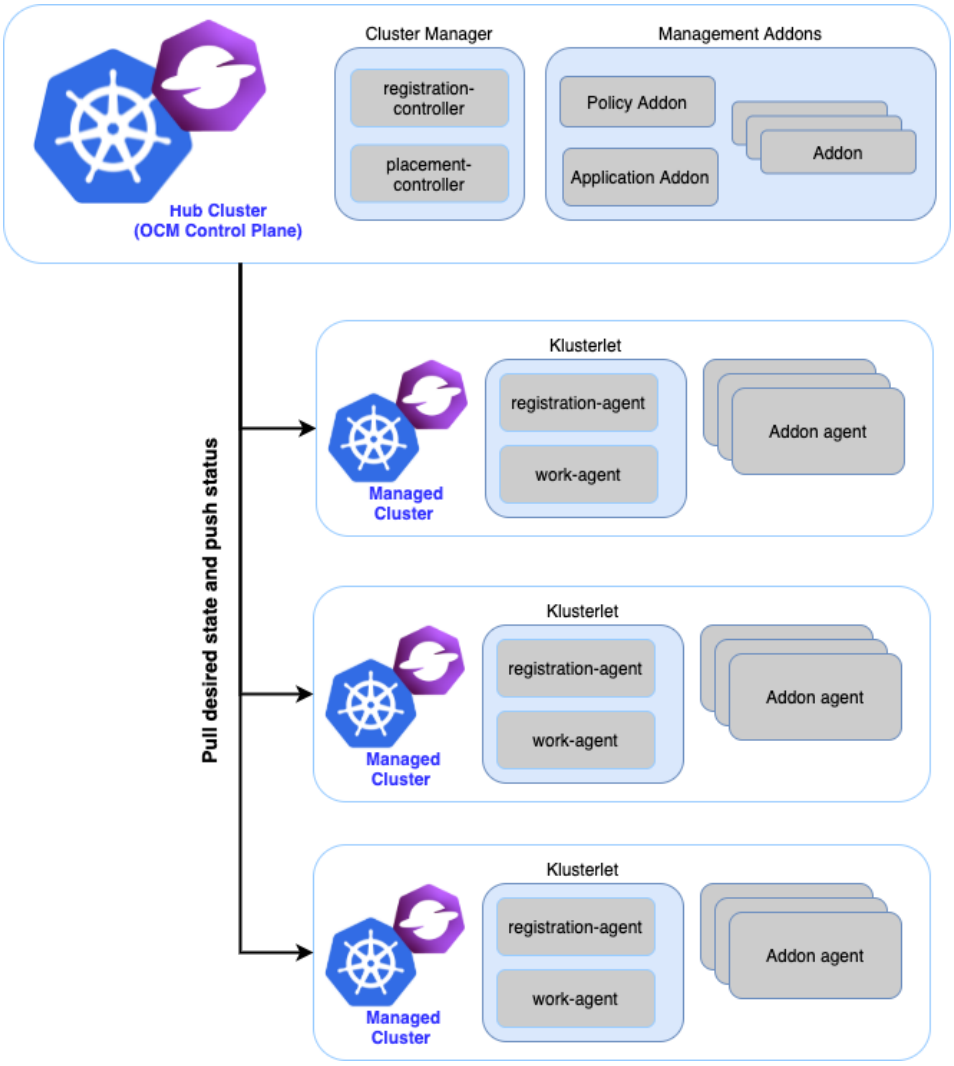

Figure 8. - Open Cluster Management Architecture (taken from [37])

It is important to note that each klusterlet works independently and autonomously, so they have a weak dependency to the availability of the hub cluster. If the hub goes down (e.g. during maintenance or network partition) the klusterlet or other OCM agents working in the managed cluster are supposed to keep actively managing the hosting cluster until it re-connects. This is ideal for edge cluster use cases. In addition, it provides a mechanism to provide back information from the objects deployed in the edges to the HUB cluster. This is done via the OCM feedbackStatus and it is leveraged by several PHYSICS components, such as the cluster onboarding.

4.1.c Cluster Onboarding

When a new cluster gets added to the set of clusters managed by the OCM hub, there are certain extra actions that are required. In the PHYSICS use case we need to ensure that certain components are installed and configured as needed, as well as connected to the needed components in the hub -- note this can be further extended to include any other resources/components, as long as they are Kubernetes objects.

The cluster onboarding mechanism (see Figure 9) is designed in a serverless fashion, using Knative eventing and serving (see next subsection):

- **Eventing**: It makes use of the **ApiServerSource Knative Sink** component to detects events related to cluster creation (OCM object)
- **Serving**: and then makes use of Knative services to deploy the actual logic in a serverless fashion to save resources -- scaling to 0 as clusters are not being added at the time.

The logic inside the Knative service is the one that can be easily extended/modified/changed to account for new/extra/different components/resources. And it is in charge of creating the needed resources in the remote cluster as well as to gather the required information, such as the service IPs to be used (through submariner) in the central hub components. It also performs calls to both local and remote PHYSICS components to provide the required information/configuration - again leveraging submariner networking.

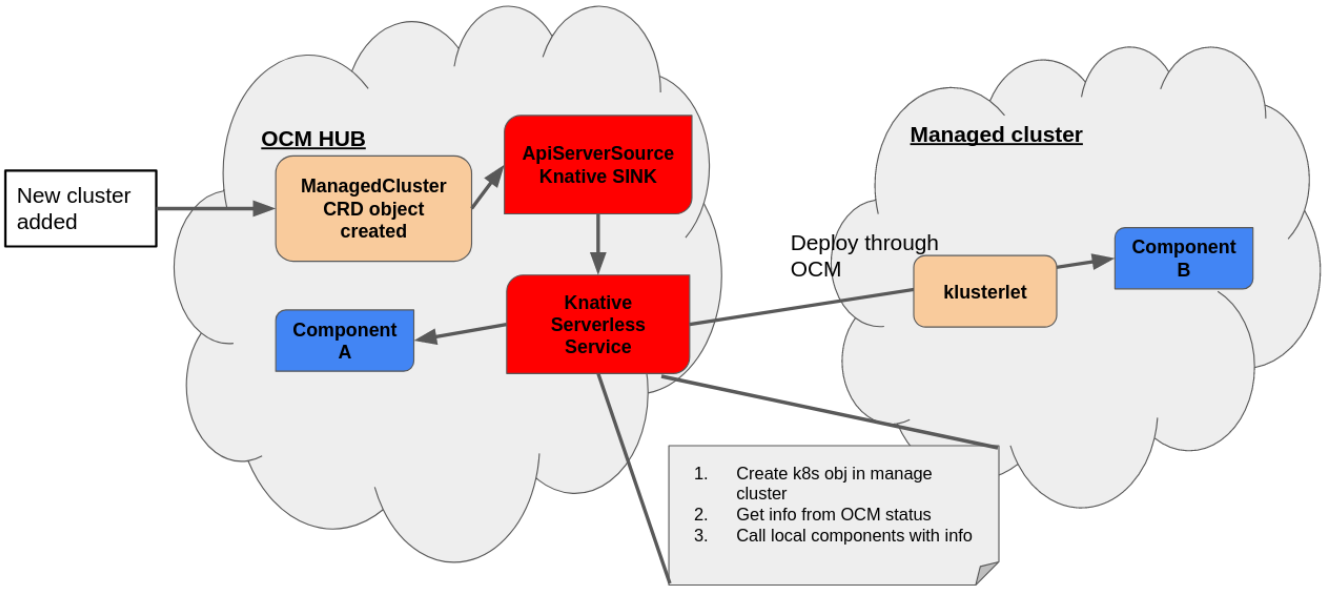

Figure 9. - Cluster Onboarding overview

4.1.d Knative

Knative is used only on the Hub, an overview of its components is shown in Figure 10, as part of the cluster onboarding process, we use both the serving and the eventing. The serving part makes it easier to run containerized applications in Kubernetes as it abstracts away some of the complexity. By creating a Knative service, the Knative operator takes care of creating the needed resources in Kubernetes, including, deployments, replicaset, services, routes, etc. The eventing part provides an easy way to react to events, by providing components to handle different types of data streams in a declarative manner, and connecting them to their related services -- thus it is perfect to enable event-driven architecture, in this case related to new clusters being onboarded. Note it uses standard HTTP POST requests to send/receive events between producers and sinks, following the CloudEvents specifications [38].

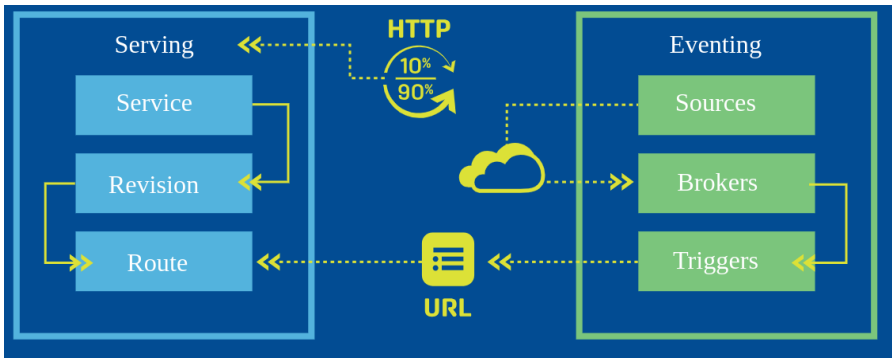

Figure 10. - Knative Components (taken from [9])

4.1.e Low footprint Edge deployments: MicroShift, K3s, Kind

In the edge (as well as for testing) we may have limited resources and therefore there is a need for Kubernetes distributions with a low footprint. As an example, it is not needed to have API HA tooling if there is only one single node. In PHYSICS we have studied several suitable projects, such as K3s, MicroShift and Kind. We have focused on the last two.

Kind [14] is a tool for running local Kubernetes clusters running containerized nodes. It was primarily designed to make testing easier, but it is also used for local development as well as for CI systems. We have made use of it to have an easier way of testing remote edges, as well as for development purposes, mainly on the multicluster related components. We have also worked on making sure its integration with both OCM and Submariner works and covers the need for PHYSICS.

MicroShift [13] started as a research project to create an experimental OpenShift/Kubernetes (OKD, the Kubernetes distribution by the OpenShift community) flavor which is optimized for edge use cases. Edge devices/clusters deployed out in the fields (as in PHYSICS e-agriculture pilot) pose very different operational, environmental, and business challenges from cloud computing. Due to that, MicroShift flavor makes some trade-off and cleanly layers on top of edge-optimized Linux OS like Fedora IoT or RHEL for Edge. At the same time, it aims to be secure and resilient to adverse networking conditions and have a very low resource footprint. In PHYSICS we have worked in promoting it, ensuring it can be used together with submariner and OCM for our use cases. During the time of the projects, this has evolved to a Red Hat product, part of the "Red Hat Device Edge" [39] .

4.1.f CRDs for PHYSICS components interactions

Kubernetes Custom Resources (CRs) [40] are extensions to Kubernetes API that are not necessarily available in default Kubernetes installation. It represents a customization of a particular Kubernetes installation and is how many core Kubernetes functions are built nowadays, making Kubernetes more modular. This model is the one followed by PHYSICS.

As previously shown in section 2, the different components developed in this WP as well as in WP4 need to communicate and generate/consume shared information. To this end, at the infrastructure level, we have created a new CRD, named WorkflowCRD, so that the needed information is accessible behind the Kubernetes API.

The CRD API resource allows defining CRs. By defining a new CRD object, a new CR is created with the name and the schema specified. And then Kubernetes API serves and handles the storage of that CR (in our case Workflow and Node). This frees us from writing our own API server to handle those resources. In addition, CRs can appear and disappear in a running cluster through dynamic registration, so a cluster admin can update the CRs independently of the cluster itself, and therefore not break the applications/controllers using them.

Once a custom resource is installed, users (in this case the PHYSICS components) can create and access its objects using standard Kubernetes APIs (for instance, kubectl), just as they do for built-in resources like Pods.

There are well known examples of CRD usage in the Kubernetes ecosystem. For example, to manage the own Kubernetes infrastructure as another Kubernetes resource: this is done with the Cluster API, where servers are defined by machine and machinesets CRD objects (in the same way as pods and replicaset are defined for applications). A definition of the machine CRD can be found here [41]. And once defined in Kubernetes, they can be retrieved as normal objects:

```
$ oc get machines -n openshift-machine-api
NAME                                          PHASE     TYPE        REGION       ZONE           AGE
ocphub-t4rh8-master-0                         Running   m5.xlarge   eu-north-1   eu-north-1a    44d
ocphub-t4rh8-master-1                         Running   m5.xlarge   eu-north-1   eu-north-1b    44d
ocphub-t4rh8-master-2                         Running   m5.xlarge   eu-north-1   eu-north-1c    44d
ocphub-t4rh8-submariner-gw-eu-north-1a-krrtj  Running   m5.xlarge   eu-north-1   eu-north-1a    32d
ocphub-t4rh8-worker-eu-north-1a-txkh8         Running   m5.xlarge   eu-north-1   eu-north-1a    14d
ocphub-t4rh8-worker-eu-north-1b-fxhzw         Running   m5.xlarge   eu-north-1   eu-north-1b    14d
ocphub-t4rh8-worker-eu-north-1c-ktkjn         Running   m5.xlarge   eu-north-1   eu-north-1c    14d
```

Code 6 - Kubernetes machine objects

And the object looks like:

```
apiVersion: machine.openshift.io/v1beta1
kind: Machine
metadata:
  annotations:
    machine.openshift.io/instance-state: running
  finalizers:
  - machine.machine.openshift.io
  labels:
    ...
  name: ocphub-t4rh8-worker-eu-north-1a-txkh8
  namespace: openshift-machine-api
  ownerReferences:
  - apiVersion: machine.openshift.io/v1beta1
    blockOwnerDeletion: true
    controller: true
    kind: MachineSet
    name: ocphub-t4rh8-worker-eu-north-1a
    uid: 590ce0b9-7a7f-4f81-bc97-b1147b827ef0
  uid: b4601937-6be3-4572-bac2-407f48c27423
spec:
  metadata: {}
```

```
providerID: aws:///eu-north-1a/....
providerSpec:
  value:
    ami:
      id: ami-...
    apiVersion: awsproviderconfig.openshift.io/v1beta1
    blockDevices:
    - ebs:
        encrypted: true
        iops: 0
        kmsKey:
          arn: ""
        volumeSize: 120
        volumeType: gp2
    credentialsSecret:
      name: aws-cloud-credentials
    deviceIndex: 0
    iamInstanceProfile:
      id: ocphub-t4rh8-worker-profile
    instanceType: m5.xlarge
    kind: AWSMachineProviderConfig
    placement:
      availabilityZone: eu-north-1a
      region: eu-north-1
    securityGroups:
    - filters:
      - name: tag:Name
        values:
        - ocphub-t4rh8-worker-sg
    subnet:
      filters:
      - name: tag:Name
        values:
        - ocphub-t4rh8-private-eu-north-1a
    tags:
    - name: Kubernetes.io/cluster/ocphub-t4rh8
      value: owned
    - name: ocphub
      value: "true"
    userDataSecret:
      name: worker-user-data
```

Code 7 - Kubernetes machine CRD example

In the PHYSICS architecture, the design around the CRDs is the next:

1. The Workflow CRDs is created at the infrastructure layer (more information about the specifics of this CRD in the Implementation and Integration in section 4.2)
2. WP4 components are in charge of creating objects of those new types (through OCM for the edges), with the relevant information. For example, for a given Workflow CRD, information about the functions that belong to a given workflow, the needs for each specific function, the relation to the functions in the workflow, etc. It also includes information from performance profiling for instance, that can be later consumed by other components in the edges, such as the co-location engine.
3. Pods will be annotated with information about the workflow CRD they belong to, as well as what function they represent in that workflow.

4. Both the scheduler and the co-allocation engines can access this information to make their decisions. As an example, the co-allocation module, when defining the affinities for a pod it will:
    - check the pod annotations to get the workflow it belongs to, and the function it represents.
    - get the information about the workflow, by retrieving the specific workflow CRD object.
    - execute its logic, depending on the information in the workflow, such as the functions it receives inputs to, or where the outputs should go, its profile information, etc.
    - update the pod object with the relevant affinities and antiaffinities.

More information about the co-allocation logic can be found at section 6. And more information about how it gets triggered in the next subsection.

4.1.g WebHook for triggering scheduling and co-allocation actions

Another part of the design is how to make sure that our scheduling and co-allocation techniques get applied in a Kubernetes cluster when a pod gets created by another entity, in this case from WP4 by using an OCM ManifestWork template, or by OpenWhisk itself.

In Kubernetes, the way to specify the scheduler to use by a pod (if the default one is not to be used) is to add the name of the scheduler in the pod spec [25]. Similarly for affinities [42]. To be able to inject that into the pod object before it is processed by Kubernetes (and therefore scheduler with the default scheduler, and with no affinities) we leverage the functionality offered by Kubernetes named Dynamic Admission Controllers and Webhooks [26].

Admission webhooks are HTTP callbacks that receive admission requests and do something with them. There are 2 types of admission webhooks: validating admission webhooks and mutating admission webhooks. Mutating admission webhooks are invoked first, and can modify objects sent to the API server, usually used to enforce custom defaults. After all object modifications are complete, and after the incoming object is validated by the API server, validating admission webhooks are invoked and are usually used to reject requests to enforce custom policies. In PHYSICS we focused on the Mutating admission webhook and the logic is the next:

- The pod object gets created, triggering the HTTP callback to execute the webhook
- The webhook has both the co-allocation and scheduler logics
    - Scheduler: based on pods annotations it selects the scheduler to be used by the pod and modifies (inserts) the information by changing the pod spec. If nothing is annotated the CacheLocality scheduler gets selected (see next section for details on the scheduler).
    - Co-allocation: based on pod annotations related to the Workflow CRD it belongs to and the function, it performs the optimizations and resolves a set of affinities and antiaffinites that are added to the pod spec too.
- The modified pod object gets stored into the Kubernetes DB (ETCD), which triggers the pod scheduling process with the new scheduler stated and the set of affinities/antiaffinities.

4.1.h Energy consumption in virtualized environment: Kepler

In PHYSICS we have chosen to use the Kepler project to obtain the energy consumption related information that other components (scheduler) leverage. Kepler uses eBPF and linux kernel tracepoints to obtain the needed data and then uses a ML model to estimate the power consumption per container. We used in nested environments, meaning on top of VMs in Azure/Amazon, therefore it was important that this functionality was provided and we worked together with the upstream community on it.

Kepler has two main components:

- **Exporter** (Figure 11): exposes a variety of metrics about energy consumption of Kubernetes components (pods, nodes) to Prometheus.

- **Model Server**: Its main feature is to return the power estimation model model. Two power modeling approaches are supported, Power Ratio Modeling and Power Estimation Modeling. More information on them at [10]. In PHYSICS we have focused on the Power Estimation Modeling, as that is the one that provides estimations when no direct access to the power metrics is available, such as when running on top of public Cloud Providers.

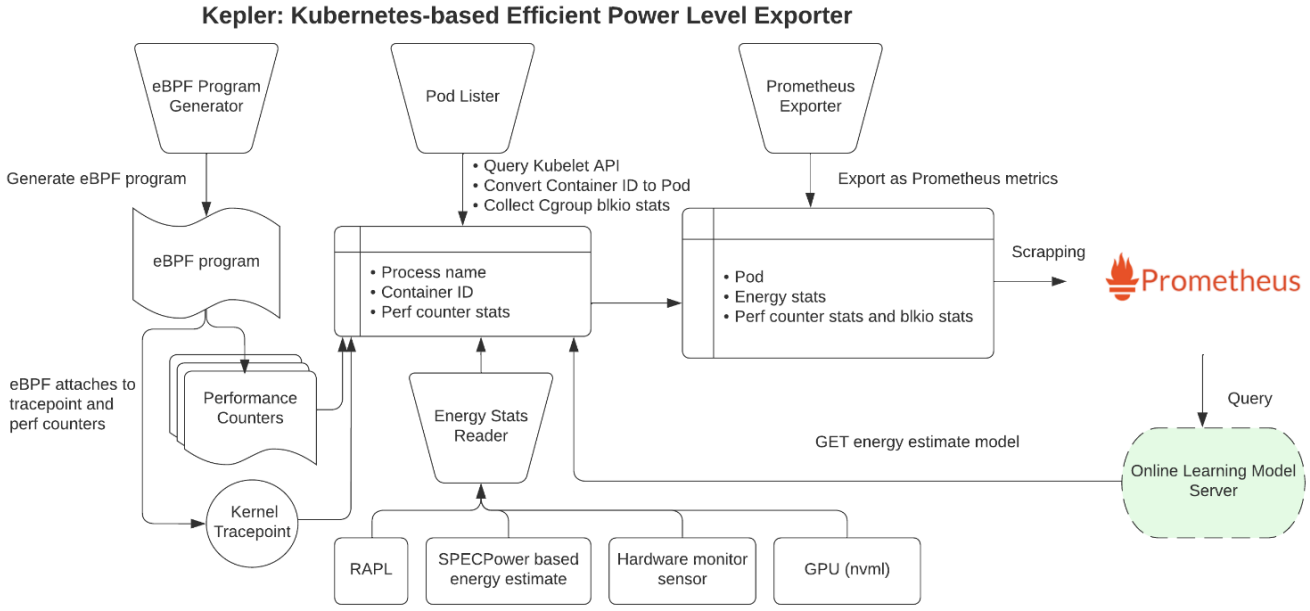

Figure 11. - Kepler Exporter (taken from [38])

4.1.i Autoscalers: VPA, HPA and KEDA

The autoscalers are not represented in the main components overview at Figure 1, as the work done is presented in deliverables related to WP3, i.e., in D3.2. However, at the infrastructure level, we have made available the different autoscalers APIs from the Kubernetes side. Initially we focused on the Horizontal and Vertical Pod Autoscalers (to scale the number of pods and their associated cpu/memory, respectively). However, those are related to CPU and Memory, which are not the best metrics for event driven applications, such as in Function as a Service, which is what we target in PHYSICS. For this reason we kept investigating the upstream space and found out the growing KEDA project [20], which focuses on the events needing to be processed instead to take the scaling decisions. More information in D3.2.

4.2 Implementation and Integration Highlights

This section covers the different implementation and integration details between components. Note some of them will keep evolving after the due time of this deliverable, and even after the PHYSICS project finishes.

4.2.a Submariner and Open Cluster Management integration

As part of PHYSICS, we are contributing to the upstream projects used for multicluster management. Besides being involved in the relevant communities along the whole life of the project (e.g., slack channel where upstream work happens), ensuring alignment with PHYSICS needs, we have worked on:

- Integration of Submariner and Open Cluster Management projects.

- Reporting bugs we discovered, and collaborating in their troubleshooting and fixes. We have reported several issues and worked upstream in their resolution:
  - https://github.com/submariner-io/submariner/issues/1608
  - https://github.com/submariner-io/submariner/issues/1625
  - https://github.com/submariner-io/submariner/issues/1631
  - https://github.com/submariner-io/submariner-operator/issues/2144
- Submariner support for the ovn-Kubernetes CNI used by OpenShift, needed to be able to connect clusters with ovn-Kubernetes to other clusters.
  - https://submariner.io/getting-started/architecture/networkplugin-syncer/ovn-Kubernetes/
  - https://github.com/submariner-io/enhancements/issues/96
- Integration of Submariner and Open Cluster Management on the edge clusters based on vanilla Kubernetes distributions (the one we have deployed in the Azure (edge) cluster or MicroShift/Kind (low footprint)
  - https://github.com/submariner-io/submariner-operator/pull/2146.

Regarding their integration into the PHYSICS architecture, PHYSICS leverages their provided APIs. For Submariner, as mentioned before, for IP connectivity there is nothing different to be done. If Submariner is properly installed and configured, it provides connectivity for pods and services when they are created in the standard Kubernetes way. The ServiceExport CRD offered by Submariner needs to be used in case service discovery is needed (DNS resolution of services cross clusters). This would be managed through the Open Cluster Management integration, providing the needed API for ServiceExport at WP4 orchestration engines.

As for the Open Cluster Management, we leverage the clusters related CRDs for multicluster management APIs. We add annotations into them so that extra information can be available, such as the cluster region. For application lifecycle management we also use the ManifestWork CRD offered. This is how the WP4 components can make use of different clusters through the central hub cluster, deploying the applications in the same way as it was in the local cluster, but with the ManifestWork CRD wrapper. The next is an example of how to deploy a container (CONTAINER_IMAGE) in a pod named "hello", in the "default" namespace, in the cluster "edge-cluster". Note the ManifestWork is created in a namespace in the central cluster, which is associated with the remote (edge) cluster. This namespace is the one being watched by the remote cluster agent to detect the need for deploying the manifest.

```yaml
apiVersion: work.open-cluster-management.io/v1
kind: ManifestWork
metadata:
  name: mw-test
  namespace: edge-cluster
spec:
  workload:
    manifests:
      - apiVersion: v1
        kind: Pod
        metadata:
          name: hello
          namespace: default
        spec:
          containers:
            - name: demo
              image: CONTAINER_IMAGE
```

Code 8 - OCM manifest work example

4.2.b Low Footprint Kubernetes distribution: MicroShift integration

In PHYSICS we have contributed to the development and testing of the MicroShift Kubernetes flavor. We have worked to make this project a more widely known upstream project, and build a community around it. We have prepared several talks in upstream events (e.g., Kubernetes meetups or devconf), as well as engaged in discussions with the folks developing them (most of them in Red Hat).

This project started as a research project in Red Hat and now it is part of the portfolio offering, as part of Red Hat Device Edge. In PHYSICS we focused on its integration with Open Cluster Management and Submariner. We discovered several issues related to pod connectivity across site due to its initial default CNI used (flannel). Later the project switched to the option to use ovn-Kubernetes instead, and we had already worked on that on the submariner side as reported in the previous subsection. As for the APIs, once integrated into Open Cluster Management, it behaves exactly the same as any other Kubernetes clusters, so the same ManifestWork (API) can be used to deploy applications on them, and thanks to Submariner integration, pods in that cluster should be able to reach to or be reachable from other pods/services in other clusters.

The main target was to be able to provision a low footprint edge by simply setting the image in a device, shipping it on-site (for example in our greenhouse use case), plug it into the network and power, and then simply join it to OCM. Then the cluster onboarding mechanism should do the rest and install/configure the needed components leveraging both OCM for management and Submariner for the connectivity across sites. Note this is partly done by the cluster onboarding mechanism where the steps to configure any extra components are explained -- see subsection 4.2.e.

4.2.c Workflow CRDs

As explained in the design specification section, as well as in Section 2, in PHYSICS we rely on new CRDs to store the information needed by different PHYSICS components and make them available through the Kubernetes API.

We have defined a new API (WorkflowCRD) in Kubernetes, and developed an operator for it, named **Workflow CRD Operator,** which is in charge of creating the specification of the CRD objects that we use (**Workflow** CRDs), following the Kubernetes Operator pattern [3]. Operators are software extensions to Kubernetes that make use of those CRDs to manage applications and their components, i.e., it allows you to extend the Kubernetes cluster's behavior without modifying the Kubernetes cluster itself.

Once the operator defines the Workflow CRD, both OCM and the PHYSICS components (mainly WP4) can start creating and/or updating objects of this type (workflow) which contain the relevant information for the other WP5 components, in this case scheduling and co-allocation engines. WP4 components are in charge of adding information related to the workflow CRD object created through OCM ManifestWorks:

- Functions that belong to the workflow
- inputs and outputs for those functions
- requested resources: cpu, mem, net
- specific hardware needs: GPUs, FPGAs, ARM/IoT, …
- performance profile of those functions, e.g., memory intensive, cpu spiky, …

Then the workflow CRD operator, local to each cluster, detects the creation of those objects and performs the needed steps:

- Detects the target platform, it can be OpenWhisk or Knative. Note the project is mainly based on the OpenWhisk support as that was the platform supporting Function as a Service when the project started. Recently Knative has added support for it and we have worked in PHYSICS in adding support for Knative functions too. The support is basic and just to highlight the workflow CRD versatility to integrate new platforms.

- Then it process the rest of the spec on the WorkFlow CRD object and perform the needed steps to register/create the function(s)
    - If the platform is OpenWhisk, it reads the information about the functions in the workflow CRD object, supporting different sources for the functions, such as code repo (e.g., GitHub), or function image (including NodeRED image). Then, it calls the OpenWhisk Proxy developed in WP4 (to abstract away some extra configuration) with the required information about extra resources and performance profile. And with that the OpenWhisk Proxy will call the OpenWhisk API to register the function, ensuring the pods generated by it will contain the required specification, regarding labels/annotations as well as resource requirements.
    - If the platform is Knative, it reads the image to deploy (i.e., it only supports passing the image at the moment, not generating it from the source code as the Knative client supports) and generates a Knative Serverless service CRD object with the parsed information from the Workflow CRD spec, in this case the maximum and minimum scaling targets and the concurrency. This triggers the Knative operator to process that CRD and create the Kubernetes objects (service, deployment, pods, routes, …). Finally, the status of the Knative CRD is reported back, including a link to the deployed function for invocation.

Note both platforms work slightly differently with regards to functions registration/deployment. While in OpenWhisk it is only registered and then the pods are created on the first invocation, and then maintained for some time waiting for successives invocations, in Knative the service is created and the pod for the function gets created and then its deployment is scaled down to 0 if no activity/request is received within some predefined period of time (30 seconds by default), then, upon the next invocation it will scale the deployment up again.

We defined an initial version of the CRD and then it was enhanced with the extra information that was needed to be passed from WP3/4 to components in WP5, more specifically related to the performance profiling. This highlights how easy it is to update the CDR spec (API) and discover new information that can be relevant for other components.

The final structure of the Workflow CRD spec is defined as:

- Type → Flow
- Platform → OpenWhisk|Knative
- Execution → NativeSequence|NodeREDFunction|Service
- ListOfActions → ordered list of actions -- for native sequences
- Actions → array of actions with its spec

Then, each Action is defined as:

- Name: name of the function
- Description: description
- Id
- Version
- Runtime: NodeJS|Python
- CodeRepo: repo to obtain the function code from
- Code: function code passed as string
- Image: function code in a docker image directly
- Annotations: extra annotations passed from WP3/WP4 components
- Resources: default K8s resources, limits and requests
- ExtraResources: other extra resources that may be needed, such as GPUs or DiskType
- PerformanceProfile: performance profile for the function, obtained in WP3. It contains low|medium|high|spiky information for the next
    - CPU, memory, fsReads, fsWrites, networkReceived, networkTransmitted

- DefaultParams: name and value parameters. It provides a way to pass extra parameters into the function invocation without further changing the API, in case they are needed for different platforms. This is leveraged for spreading actions across clusters, pointing to the endpoint in the remote cluster.

In the next code snippets, we see one example for a NodeREDFunction, a NativeSequence, and a NodeREDFunction but for multicluster -- actions across different clusters.

```
apiVersion: wp5.physics-faas.eu/v1alpha1
kind: Workflow
metadata:
  name: hello-NodeREDfunction
  namespace: physics-namespace
  annotations:
    id: "7c6a3de135b840c5"
    version: "1"
spec:
  execution: NodeREDFunction
  listOfActions: []
  native: true
  platform: openWhisk
  type: flow
  actions:
  - name: hello-world
    description: "hello world"
    id: 247a1728e0231123
    version: 1.0.0
    runtime: blackbox
    code: "function main(msg){\n\nconsole.log(msg);\nmsg.payload={'response':'hello
'+msg.payload.value.name};\nreturn msg;}"
    image: "gkousiou/NodeREDhelloaction"
    annotations:
      optimizationGoal: Performance
      importance: "High"
    resources:
      limits:
        memory: 128
      requests:
        cpu: 1
        memory: 128
    extraResources:
      gpu: true
      diskType: ssd
    performanceProfile:
      cpu: medium
      memory: low
      networkTransmitted: low
```

Code 9 - NodeRED Function (single cluster)

Note in the above code snippet that type, platform and execution. In addition, it just contains one single action, with some extra annotations, requesting a GPU, and with some performance profile set for helping the WP5 components -- in this case helping the co-location engine to decide where to locate/not-locate the pod created for it.

```
apiVersion: wp5.physics-faas.eu/v1alpha1
kind: Workflow
metadata:
  name: hello-sequence
  namespace: physics-namespace
  annotations:
    id: "19fe4293742e0b2c"
    version: "1"
    cluster: cluster1
spec:
  execution: NativeSequence
  listOfActions:
    - id: 339d2ef8b0b29795
    - id: 3a807141f16764a5
  native: true
  platform: openWhisk
  type: flow
  actions:
  - name: hello
    description: "hello"
    id: 339d2ef8b0b29795
    version: 1.0.0
    runtime: nodejs
    code: "function main(msg) {\nmsg.payload=msg.payload+' hello';\nreturn msg;}"
    performanceProfile:
      cpu: medium
      memory: low
      networkTransmitted: low
  - name: world
    description: "world"
    version: 1.0.0
    id: 3a807141f16764a5
    runtime: nodejs
    code: |
      function main(msg) {
      //implies affinity with the other function in the sequence
      msg.payload=msg.payload+' world';
      return msg;
      }
    resources:
      limits:
        memory: 256
```

Code 10 - Native Sequence Function (single cluster)

Similar to the previous one, this object defines a flow which consists of two native sequence actions. It includes a performance profile only for the first function, and a resource limit only for the second one, just to demonstrate that different functions can have different parameters/options.

```
---- # In cluster1
apiVersion: wp5.physics-faas.eu/v1alpha1
kind: Workflow
metadata:
  name: hello-sequence
  namespace: physics-namespace
```

```
  annotations:
    id: "19fe4293742e0b2c"
    version: "1"
    cluster: cluster1
spec:
  execution: NodeREDFunction
  listOfActions:
    - id: 339d2ef8b0b29795
  native: true
  platform: openWhisk
  type: flow
  actions:
  - name: hello
    defaultParams:
      actionname-host: ip ow cluster 2
      actionname-namespace: openwhisk namespace
      actionname-credentials: credentials ow cluster 2
    description: "hello"
    id: 339d2ef8b0b29795
    runtime: blackbox
    image: registry/image:label
    performanceProfile:
      cpu: medium
      memory: low
      networkTransmitted: low
---- # In cluster 2
apiVersion: wp5.physics-faas.eu/v1alpha1
kind: Workflow
metadata:
  name: hello-sequence
  namespace: physics-namespace
  annotations:
    id: "19fe4293742e0b2c"
    version: "1"
    cluster: cluster1
spec:
  execution: NodeREDFunctionNativeSequence
  listOfActions:
    - id: 3a807141f16764a5
  native: true
  platform: openWhisk
  type: flow
  actions:
  - name: world
    description: "world"
    id: 3a807141f16764a5
    runtime: blackbox
    image: registry/image:label
    performanceProfile:
      cpu: medium
      memory: low
      networkTransmitted: low
    resources:
      limits:
```

```
        memory: 256
```

Code 11 - NodeREDFunction for Multicluster invocation. Function Hello in cluster1 gets the default parameter to build the invocation for the world function in cluster 2

For the multicluster, each part of the application is an independent manifest and the workflows are deployed in each corresponding cluster using OCM. The important part is the usage of the defaultParams to pass the key:value pairs that allow invoking the functions. The values are processed by the internal node red application to call the right functions. For this reason, the solution uses the main deploying object, the NodeRED image. Then it uses WP3 flows to obtain parameters from invocation to build the remote invocation. This allows us to point to remote endpoints and keep the flow of functions execution in a different cluster, while changing them without interfering with the internals of the node red application. Each function could have more than one remote function as a dependency, for this reason the names of the related functions in the application should be included in the application graph (see D4.2 Chapter 6).

4.2.d Webhook for scheduler and co-allocation engines

As mentioned before, and as explained in the next Pod/Function workflow creation flow section, we made use of Kubernetes Mutating Webhooks to implement the functionality that allows us to modify the pod object definition with the decisions of the schedulers to be used and the affinities to enforce.

The implementation consists on 3 main building blocks:

- **MutatingWebhookConfiguration Kubernetes object**: This is the Kubernetes knob to allow us to define what type of objects we are going to receive a call back from. In our case the next, which ensure we receive events in case of pods being created, which is the step when we need to decide on the scheduler to use and the affinities:

```
apiVersion: admissionregistration.k8s.io/v1
kind: MutatingWebhookConfiguration
metadata:
  name: physics-webhook
webhooks:
  - name: physics-adminssion-controller.openshift.io
    ...
    rules:
      - operations: ["CREATE"]
        apiGroups: [""]
        apiVersions: ["*"]
        resources: ["pods"]
```

Code 12 - Mutating webhook example

- **Kubernetes DaemonSet:** Runs our webhook logic in a container, ensuring it is running in all the master nodes.
- The webhook logic itself, runs as a **webhook HTTP server** that will receive the pod object request and perform the needed actions to modify it with the proper scheduler and affinities to use.

Note the pod object needs to have some annotations that allow the co-allocation and scheduler to take the needed decision. In addition, note the scheduler being selected needs to be running on the cluster as a pod, as explained here [25]. For more information about the co-allocation engine actions, see chapter 6.

4.2.e Kepler integration

PHYSICS uses Kepler to obtain the energy related information for both the applications running on top (pods) as well as for the cluster/nodes itself. As mentioned in subsection 4.2.e, and in Section 3, its information is used to obtain an energy score for the managed (edge) clusters -- through Prometheus.

In our case, where our cluster is running on VMs in different cloud providers (AWS and Azure), it was important that Kepler was able to work in a virtualized environment, hence the use of power estimation modeling.

We found some problems in such a type of deployment where Kepler was only getting information about the pods that were running before it was deployed. To tackle the problem we engaged with the Kepler upstream community, providing valuable inputs about missing functionalities that PHYSICS use cases will require. We worked together with them to report the problem, replicate the issue and evaluate the solution. The different details on the issues we reported, for the particular problems and how we collaborated with the upstream Kepler community can be found in these already closed and corrected issues[3][4]. The code changes made in the context of these bug corrections and the involvement of the PHYSICS team to resolve them were crucial for the correct functioning of Kepler in the particular context of PHYSICS project and use cases.

Furthermore, another aspect that we needed to update was the sampling rate of Kepler's collection (or estimation) of energy data, which by default was hardcoded at 3 seconds. We needed this to be modifiable in order to have the flexibility to minimize it using a configuration parameter and hence be able to have more accurate monitoring. This was reported in this issue[5] and eventually our team proposed a pull request which has been accepted by the Kepler community and merged in the upstream version[6].

In addition, we have made use of the Grid5000 experimental testbed to perform an evaluation and validation of the Kepler's power estimation model, comparing the values that were being estimated with the real values obtained from Grid5000. Grid5000 provides per node wattmeters which capture the instant power consumption of the different components of the node (such as CPU, Memory, network card, etc). More on this evaluation is provided in Section 4.3. Our ongoing efforts are dedicated to addressing a recently identified bug that prevents us from experimenting and eventually proving the validity of the power estimation model with real wattmeters. This can be followed in the currently open (at the time of the writing of this report in September 2023) issue here[7]. In any case, the collaboration with the Kepler upstream community will be continued after the end of the project, since the integration of energy related aspects with serverless environments, FaaS applications and in general Clouds is very important as it can play a role in the decarbonization of future Cloud platforms.

4.2.f Cluster onboarding flow

This section describes the interactions between the above components during the cluster onboarding process, i.e. when a new cluster (managed cluster) joins the OCM Hub.

Once a (set of) cluster(s) is onboarded into Open Cluster Management, the different clusters can be managed from the hub cluster (i.e. the central cluster used to manage the different edges, and that contains the Submariner broker). The first step, before being able to deploy Functions in a given edge, is to configure (from the central cluster) the edge with the required PHYSICS components/applications, connecting them with the required components in the central (Hub) cluster. This is depicted in the next Figure:

[3]https://github.com/sustainable-computing-io/kepler/issues/594
[4]https://github.com/sustainable-computing-io/kepler/pull/635
[5] https://github.com/sustainable-computing-io/kepler/issues/539
[6] https://github.com/sustainable-computing-io/kepler/pull/942#event-10436063440
[7] https://github.com/sustainable-computing-io/kepler/issues/790

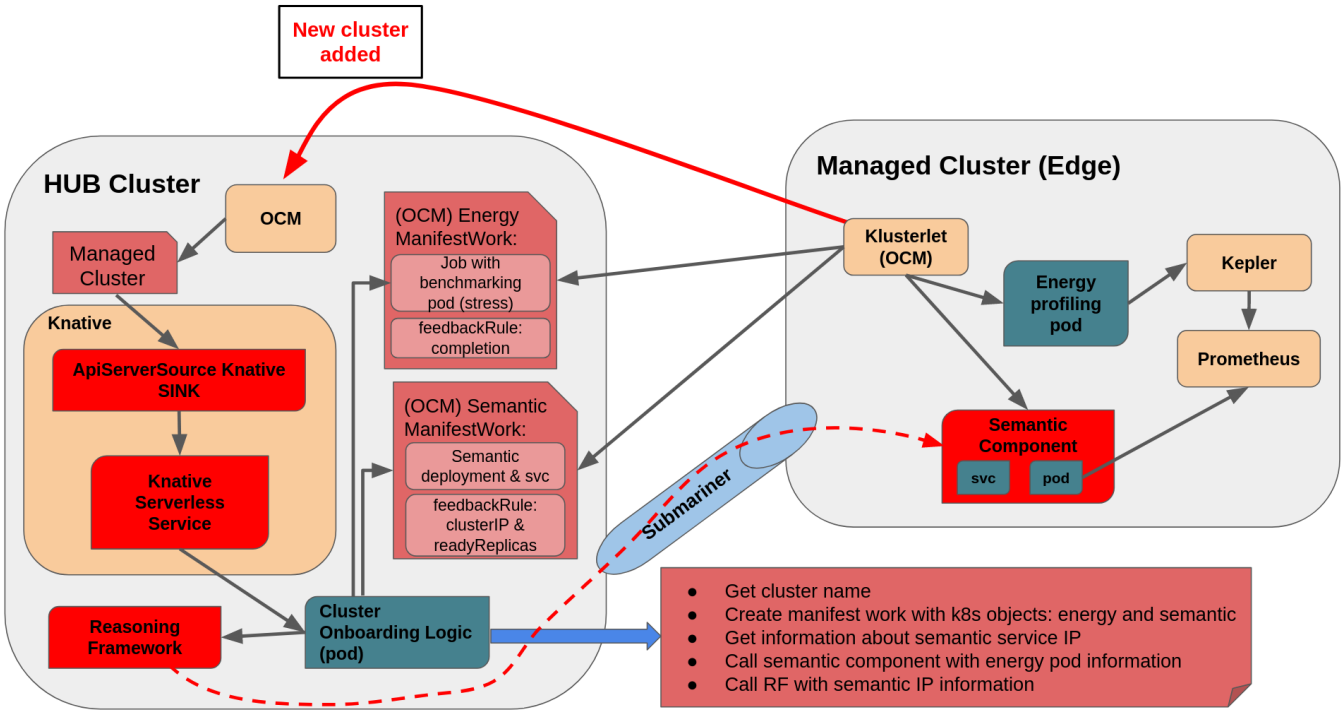

Figure 12. - Cluster Onboarding overview

As highlighted in Figure 12, the **API and components of Open Cluster Management** are leveraged as the basis for **configuring the (edge) clusters** as needed. The OCM **ManifestWork** contains a list of Kubernetes objects to be created, and that could include not only pods, but also services, ServiceExports (for the Submariner support, to make the pods/services available across different clusters), and other specific CRDs, for instance the ones related to the Workflow CRD, or the Kubernetes nodes itself.

In addition, PHYSICS relies on **Knative capabilities** to make it event driven and serverless (saving resources as clusters are not being added all the time).

The process is the next:

1. A remote cluster gets added to the HUB (as Managed Cluster) through OCM, which creates an object of type ManagedCluster
2. The Knative APIServerSource receives this event and invokes the Knative Serverless Service
3. The Knative Serverless Service receives the event and creates a new pod with the **cluster onboarding logic** (if there is not one already created), and once it is ready, it redirects the event to it.
4. The cluster onboarding pod process the request and:
    a. Obtains the **cluster name**
    b. Create an (OCM) ManifestWork which includes a Kubernetes Job that will generate some **benchmarking load** in the managed cluster. The klusterlet agent in the remote cluster is in charge of applying this ManifestWork associated to it, and create the pod with that benchmark
    c. **Waits** until the job is **completed** -- by using OCM feedbackRule
    d. Create an additional (OCM) ManifestWork which includes the definition of the **semantic deployment** and its associated **service**. Again the klusterlet agent is in charge of creating the local resources in the remote cluster.
    e. **Waits** until the deployment is **ready** and **obtains its service IP** - by using OCM feedbackRule

f. Makes use of submariner to **call the semantic service IP** and provide the information about the previous job, to be used to estimate the energy consumption and cluster score
g. **Calls the Reasoning framework** to provide the information about the semantic service IP, so that it can start requesting semantic information from the new cluster.

Note extra configurations can be easily added, as part of the cluster onboarding logic component, or even created extra Knative Serveless Services that react to the same events and perform other actions in parallel.

4.2.g Pod/Function registration and invocations flows

This section presents the interactions between the above components for both the functions registration and their execution. Once the infrastructure is configured as needed, the functions/pod registration and invocation can be started. The creation flow is depicted in the next Figure.

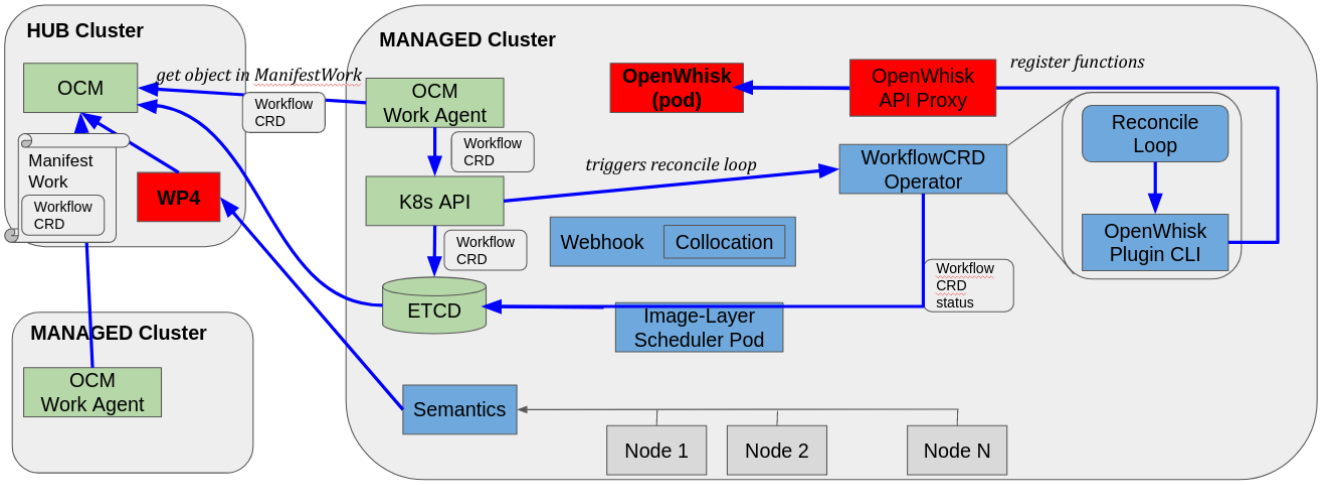

Figure 13. - Function Registration flow and interactions

And the steps are:

1. WP4 components, making use of the Semantic information, will select one or another managed cluster and will use the OCM (Kubernetes) API to create a ManifestWork object which includes the WorkflowCRD object that defines the flow with the set of action(s) and its related information. This object gets created in the HUB, in a specific namespace that is associated with the selected Managed cluster.
2. The selected managed cluster gets the ManifestWork object to apply and creates the objects inside it in the local cluster (this is the OCM Klusterlet component).
3. When the WorkflowCRD object gets created, the WorflowCRD operator gets notified about it and processes its specification (this is called a reconcile loop in the Kubernetes operator world).
4. Then, the Operator calls the OpenWhisk API proxy with the preprocessed information from the WorkflowCRD, which in turns calls the OpenWhisk API to do the actual registration of the function.
5. Finally, the workflowCRD Operator reports back the status of the operation in the WorkflowCRD status section.
6. In parallel, the OCM also fills in the status information in the ManifestWork that is used by the Hub to obtain the status of the operation.

Once the function is registered, it can be executed. There are other PHYSICS components that are exercised as part of this function invocation operation. Figure 14 shows the flow and interactions.

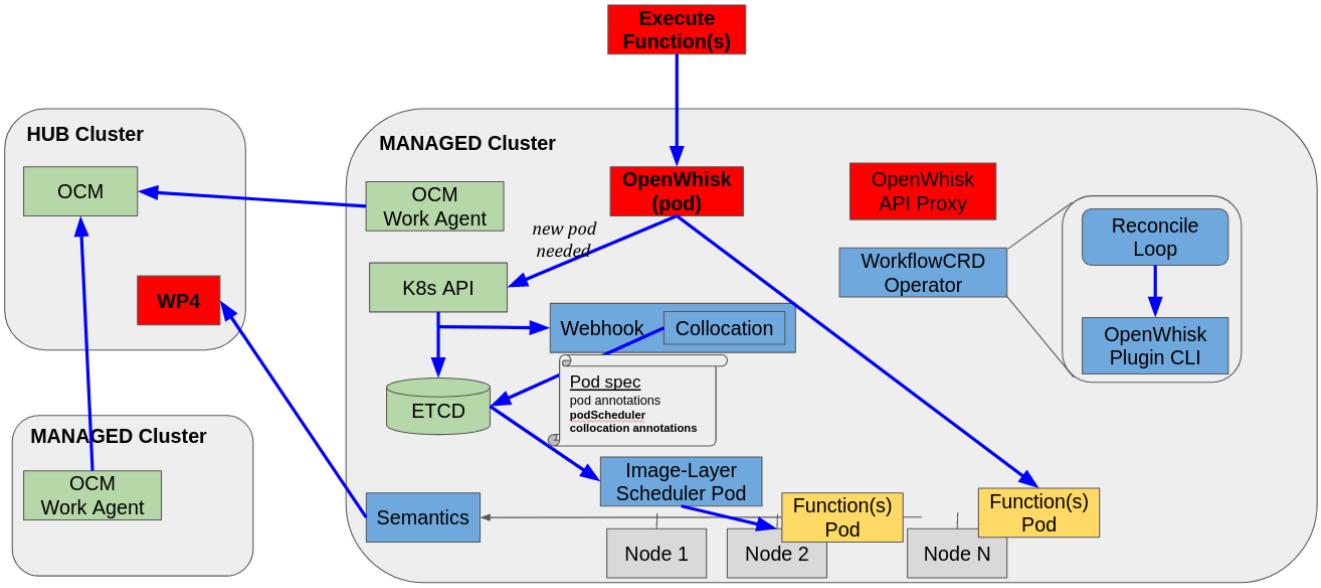

Figure 14. - Function Execution flow and interactions

When a function gets invoked by directly using the OpenWhisk API or from another Kubernetes application consuming it, the flow is the next:

1. If there is no available (hot/warm) pod already running for handling the execution of the function, OpenWhisk will contact the Kubernetes API with the pod specification that needs to be created, which in addition will contain certain annotations that may be required or passed through other WP components
2. This pod creation request is intercepted by the Webhook. The webhook selects the scheduler that must be used for the pod, in our case by default the CacheLocality scheduler presented in Section 5 and modifies the pod spec accordingly to point to it.
3. Then, the second part of the webhook logic is to execute the co-allocation logic, which in turns makes use of the workflowCRD associated object to obtain the affinities/antiaffinities depending on the function performance profile and the possible interferences with other already running functions. The result gets also annotated on the pod spec and finally stored in Kubernetes ETCD datastore for further processing.
4. The normal scheduling process starts in Kubernetes. The scheduler is notified about a pod that needs to be scheduled (it is not associated with any node) and executes the filtering (depending on the resources availability and the affinities) and the weighting. The weighting selects the most suitable node to associate with the pod, depending on the presence of container image layers. Note: the scheduler executed is not the default scheduler, but the one associated with that pod.
5. Finally, the pod gets associated with the node, that kubelet and the CNI running in that node are the ones in charge of creating the pod and connecting it to the network -- this is the normal Kubernetes process.

4.3 Experimentation Outcomes

During the first phase of the project, the focus was on the selection of the main (upstream/open-source) components as well as defining their interactions and APIs by using the CRDs and Operators model. We have implemented a first prototype based on those, as well as an initial (limited in functionality) Webhook that is able to trigger the scheduling and co-allocation engines. With more details we have:

- Deployed the main OKD (upstream version of OpenShift) on AWS and started its configuration and the deployment of other components/operators on top.
- Deployed an external (testing) cluster and connected it to the cluster running on AWS.
- Deployed and evaluated the feasibility of using Open Cluster Management for our multicluster orchestration needs.
- Deployed and evaluated the feasibility of using Submariner as the tool to interconnect workloads in a multicluster scenario. We have identified several bugs and worked with the upstream community on them, as reported in section 4.2.a.
- Work with the MicroShift project for low footprint openshift at the edges. Initial testing and evaluation of missing integration points with OCM and Submariner. We started working on fixing the issues (contributing the fixes to the respective projects).
- Create the initial webhook logic that can later be extended to plug in the scheduler and co-location engines.
- Create the initial CRDs definitions used for interactions between WP4, and WP5 semantic, scheduler and co-allocation engines.

During the second phase we have focused more on the multicluster setup and completing the integration points that were missing. We have also worked on new components, such as Kepler for the energy consumption, or the Knative integration due to its increasing adoption and evolution upstream (now they support function as a service too, not just serverless). With more details we have:

- Work on the cluster onboarding mechanism, and its deployment in the HUB cluster, including testing with remote edges
- Work on extra issues with Submariner integration with OCM, when a multicluster setup containers heterogeneous clusters (different Kubernetes/cni versions)
- Extended/Update APIs by updating the WorkflowCRD APIs to enhance the communication between different PHYSICS components
- Energy Monitoring tools evaluation and adoption of Kepler, including configuration and testing in our testbeds
- Engagement with Kepler community for fixing the gaps of the tool, specially targeting the PHYSICS use cases
- Engagement with Knative upstream community to enhance both projects (WorkflowCRD operator, and Knative) and find synergies between both projects. As part of that we also included basic support for Knative in our WorkflowCRD operator.
- The deployment, testing and usage of different upstream tools/projects led us to report several bugs or missing functionalities, as well as collaborating with the communities in their resolution. A complete list is gathered in deliverable D7.4.

4.3.a Knative Integration and Upstream engagement

When we started the project, there was no support for functions at Knative, which made us go for OpenWhisk instead. During the project duration, Knative has matured a lot and it included, among many other features, the support for Function as a Service.

In addition, we made a presentation about PHYSICS as part of the Red Hat Research Days where there was a good attendance from people working on the Knative upstream project. This started a nice collaboration with their upstream team. We engaged in several discussions about the gaps we found in Knative as well as the extra functionality being provided by the WorkflowCRD operator or the extra requirements from the PHYSICS point of view. This engagement was also continued as part of the DevConf Hackathon that we organized, where people from Knative team wanted to participate and whose main topic was about the PHYSICS infrastructure and how to build an operator for Knative abstraction, with the focus on multicluster [43]. As a result of that, we also integrated basic support for Knative in our WorkflowCRD Operator, so that it could create either OpenWhisk or Knative functions.

4.3.b Kepler Performance Model Estimations Evaluation Methodology

As part of the Kepler usage in the projects, and besides the contributions made to its functionality, we have started a study about the accuracy of Kepler performance estimations which will be used as a validation of their ML based energy estimations. This is an important point because even if Kepler is supported by a large community there is no previous work performed that evaluates the accuracy of their energy model estimations.

The interesting aspect of Kepler is that through its fine-integration with Kubernetes, it allows the collection of energy related metrics in the granularity of pods, and since we know exactly which pod participate in each FaaS workflow and application, it will allow us to calculate the energy consumed for each FaaS application. This may eventually enable the motivation of users to better optimize their applications, while opening to new functionalities such as charging the consumed energy per application.

In the context of PHYSICS WP4, we have developed and published a multi-objective algorithm and study for FaaS applications' execution optimizations upon the edge-cloud continuum [44]. The algorithm proposed in that article is currently being enhanced to take into account the energy consumption as one additional heuristic and objective in the algorithm. In this context, we are taking into account the Kepler monitoring since it can give us fine-grained results per pod level.

For this purpose, we have prepared the following methodology to evaluate the accuracy of the models Kepler estimation models. It consists of the following steps:

- We make use of Grid5000 experimental testbed which provides Omegawatt wattmeters on a number of nodes, on some of the Grid5000 sites[8]. These wattmeters give us accurate instant power consumption in watts per node (including the consumption of CPUs, RAM, network card, PDU, etc) with a 1 sec sampling rate.
- We allocate 2 compute nodes with the wattmeters on Grid5000 and we collect their power consumption profile when running just the OS on the node.
- We then deploy Kubernetes, Openwhisk and Kepler on the compute nodes having one as master and the other as worker and we collect their power consumption profile with the Wattmeters again. We are mainly interested in the worker node profile.
- We then deploy a workload composed of 10 repetitions of each FunctionBench benchmark and collect the power consumption profile of the worker node when executing each benchmark. We pay attention having only one benchmark (meaning one pod) running at each time to be sure that only this benchmark/pod (besides the default OS and basic tools) is influencing the power consumption of the node.

[8] https://www.grid5000.fr/w/Lyon:Wattmetre

- We collect the power consumption profile from the wattmeters at the node level and from Kepler at the pod level. This will allow us to extrapolate the Kepler values and try to determine how accurate its power estimation model is in comparison to the real values collected by wattmeters.

This methodology is currently being used and the experiments are currently being conducted but we are blocked because of another bug[9] that we have found in Kepler which does not allow us to get valid results when Kepler is deployed upon Grid5000 nodes. Once we have managed to resolve this bug we will be able to use the above methodology and validate the Kepler estimation models and then use these results in a new article which is currently being written for the multi-objective algorithm considering energy consumption as a new objective.

As an initial example of usage of energy consumption metrics in the context of this methodology; Figure 15 shows the power consumption profiles of different FunctionBench benchmarks per pod level when executing them upon the PHYSICS Azure testbed. Figure 16 depicts different power consumption profiles of different FunctionBench benchmarks per node level when executing the 10 repetitions of the same benchmark, one after the other upon different Grid5000 nodes.

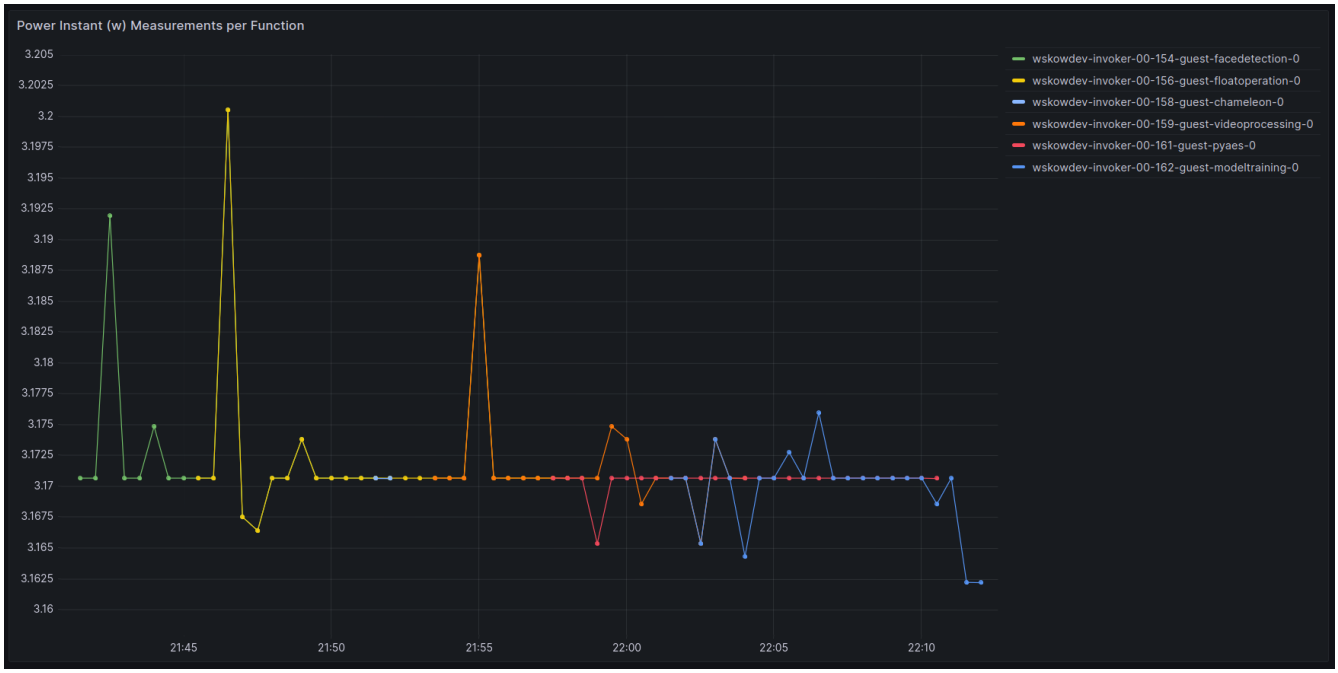

Figure 15. - Instant power consumption (in Watts) per pod when executing a workload composed of different Function Bench applications upon the PHYSICS Azure testbed

[9] https://github.com/sustainable-computing-io/kepler/issues/790

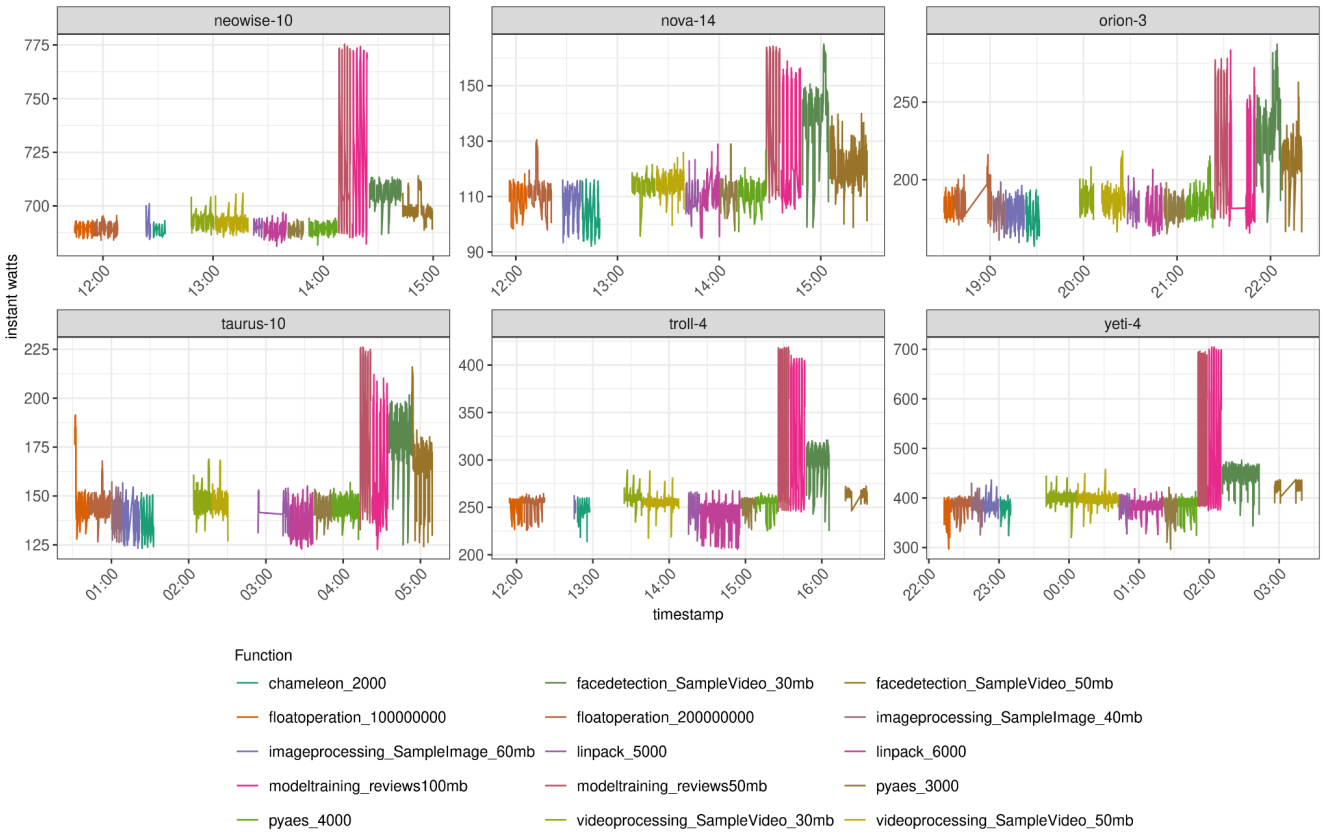

Figure 16. - Instant power consumption (in Watts) per node, featuring different Grid5000 nodes, when executing a workload composed of different Function Bench applications upon the Grid5000 testbed

4.4 Next Steps

After the project concludes, we intend to remain actively involved in the infrastructure resource management components. Our goal is to enhance versions of some of the components as well as increased usage of them. The primary objective is as follows:

- Keep engaging with the Knative upstream community. This will hopefully end up in more adoption inside the Knative project of developments (or ideas) investigated during the PHYSICS project in the WorkflowCRD operator. As an example, a similar idea about the workflowCRD is being used now in Knative, defining sequences. More details in [45].
- As part of partners participation in other EU projects, the target is to reuse some of the PHYSICS components, from the infrastructure layer, in some of them that have some similar needs. As an example, Red Hat is participating in the CODECO [46] project and there is interest in automation around multicluster configurations and the mechanism developed for the cluster onboarding may be resources/enhanced to cover their use cases.

5. Adaptable Provider Level Scheduling Algorithms

This section focuses on the description of the local level scheduling taking place individually on each cluster that participates in the global continuum. In particular, the scheduling algorithms and related mechanisms will be responsible for the intelligent placement of independent "Functions" of a broader FaaS application workflow upon the underlying compute infrastructure of a single cluster.

The research upon scheduling algorithms for FaaS applications executed on hybrid computing infrastructures has to pass through studies that analyze the behaviour of FaaS applications executions capturing the different phases of deployment environment preparation, resources utilization, load balancing, etc. This study makes use of a scheduling simulator Batsim[47], a real testbed infrastructure deployed upon Grid5000[24] and a suite of FaaS benchmarks FunctionBench[48] that have been particularly adapted to fit the needs of our experimentation. The methodology and mechanisms used during this study play an important role not only in the initial development of the scheduling algorithms but also in their optimizations and scalability improvements. Hence, it will be described as side mechanisms of the component and the relevant code will be made available in the PHYSICS repo.

Based on this methodology and the initial study we have determined a scheduling algorithm with very interesting benefits for FaaS applications and we have implemented it in the context of PHYSICS. This algorithm has been implemented as a plugin in the level of Kubernetes and its goal is to minimize the cold start delays of FaaS applications by taking into account the layers of the images to be downloaded and by favoring the placement of a function to nodes where more layers of the particular image are available.

The remainder of this section is as follows: Initially, we provide an experimentation methodology and an initial analysis which is performed to have an initial feedback on which are the most interesting scheduling algorithms to explore in a FaaS execution environment; then, we provide the design specification of our component followed by the implementation and integration highlights of the simulated version along with the related experimentation outcomes. The following subsection presents the details related to the actual implementation upon Kubernetes, along with the experimentation procedure to validate the effectiveness of the new scheduling algorithm. Finally, the last subsection describes the next steps.

5.1 Experimentation methodology and initial analysis

Based on the specific architectural choices of PHYSICS, such as the selection of OpenWhisk as the FaaS layer and Kubernetes as the resource manager and orchestration layer; the local level scheduling algorithms will be implemented as schedulers in Kubernetes to capture the allocation of computational resources, but also parts of the scheduling logic will lie within OpenWhisk to address the problematics related to each individual function to be deployed.

In order to better understand the internals and interactions taking place during the scheduling of functions and try to investigate which are the most interesting scheduling algorithms to focus on we have started our research by defining an experimentation methodology and by performing an analysis of FaaS applications execution. For this, we have used the following tools:

- A suite of FaaS benchmarks FunctionBench which we have adequately adapted to be executed under the OpenWhisk-Kubernetes context [49].
- The usage of Grid5000 experimental platform to allow the deployment of our analysis.
- A set of scripts to automate and reproduce the deployment of an OpenWhisk-Kubernetes-Prometheus environment upon the Grid5000 platform.
- A set of scripts to collect the outputs of the experiments and provide plots and graphs to visualize various metrics and get insights regarding the behavior of FaaS applications' executions [50].
- The Batsim-Simgrid scheduling simulator to allow the study of scheduling policies under particular contexts in a simulation mode.

Based on these tools we have defined an experimental methodology that makes use of the real FaaS benchmarks deployed upon a pre-provisioned Openwhisk-Kubernetes-Prometheus cluster upon Grid5000 under different conditions to study the internals of Openwhisk and Kubernetes while trying to extract interesting insights related to the execution of typical FaaS applications and their scheduling needs.

In this context, we were initially able to separate and study the different phases of functions' execution - allocating container, deploying container, executing containers, destroying containers - and regarding the platform level - downloading input data, executing functions, uploading input data - to then analyse potential space for improvements.

In order to do this, we have deployed Kubernetes-OpenWhisk on two isolated nodes of the Cluster Grid5000, with the following configuration per node: CPU: 2 x Intel Xeon E5-2660 v2, cores: 10 cores/CPU, memory: 128 GiB, storage: 1 x 600 GB HDD + 4 x 600 GB HDD, network: 1 Gbps (SR-IOV) + 2 x 10 Gbps (SR-IOV). Then we performed executions, with different inputs, several FaaS adapted functions[41] such as, float operation, matrix multiplication.

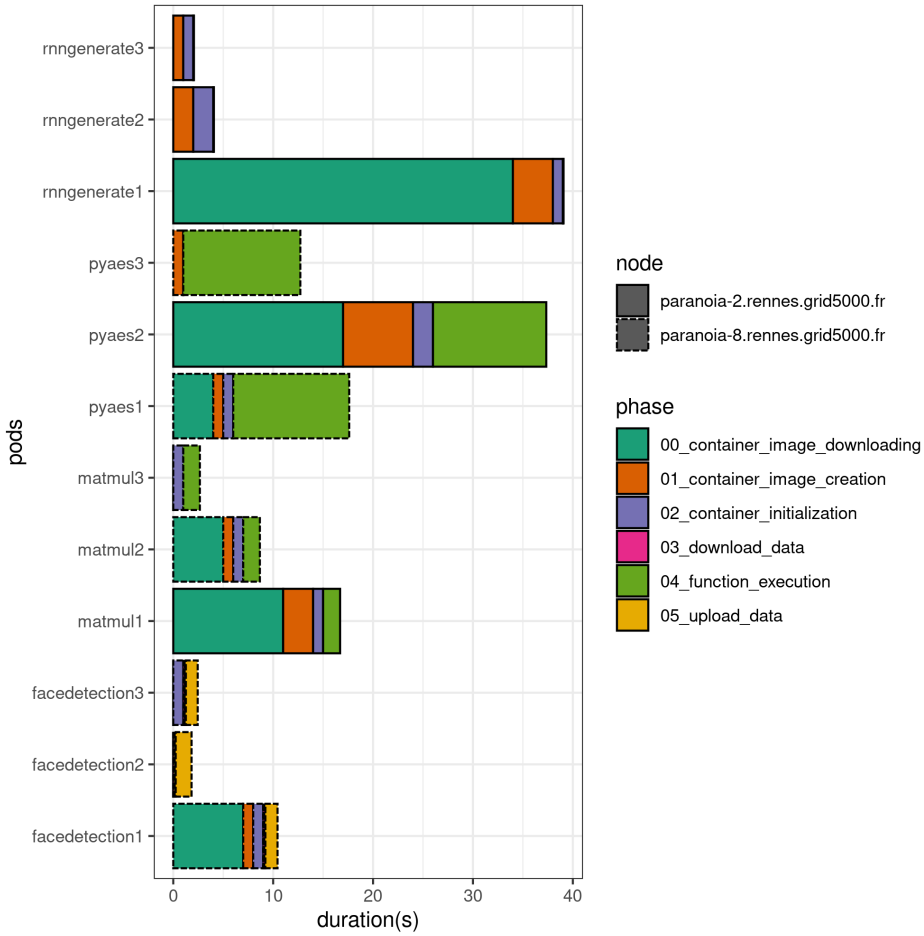

Figure 17. – Results from executing several functions

The Figure above summarizes our results and the execution of several functions. The y-axis shows function names while their duration(s) is shown in the x-axis. By the colors it is possible to see the different phases performed by each function, and by the lines (solid or dashed) it is possible to see the node where the functions were executed. The phase from 00 to 02 relies on the preparation of the containers required by the functions. They are performed by Kubernetes.

The following phases, from 03 to 05, are performed by OpenWhisk, once the containers are ready for usage. This simple experiment aims to show the different behavior of different instances of the same functions, when allocated to different nodes. It is possible to see that instances of functions allocated to the same nodes reduced the duration of the three first phases in subsequent instances. For example, from top to bottom, rnngenerate, pyaes and facedetection considerably reduced the duration of phases related to their containers when executed the second or third instance of the same function in the same node. This is due to the re-usage of the same container when such functions were allocated to the same machine. On the contrary, functions such as, from top to bottom, pyaes and matmul performed the same phases twice or more because their different instances were allocated to different nodes.

Hence, our observation is that in a typical FaaS context where task executions are usually less than 10 minutes the containers' download and initialization time have taken, proportionally, a considerable part of the whole deployment, while most of the times taking even longer than their functions' execution time.

Based on the above analysis as performed for the default Openwhisk-Kubernetes case we decided to initially focus on the image and image layers locality as a means to minimize the download phase and speed-up the deployment of functions. This leads us in designing scheduling policies in a way to favor nodes that already have the needed container image or at least some layers of it. Both cases will contribute in minimizing the pre-execution phase and eventually decrease the turnaround time of each function while improving the performance of the system.

5.2 Design Specification

The local cluster scheduling algorithms are provided basically by specific Kubernetes scheduling policies. The way that these algorithms are called and interact with the various components of PHYSICS stack is described in more detail in section 2.2.2, where a detailed figure shows where the scheduler is situated and how the information flows in relation to it.

In particular, the higher level placement decision, as performed by the Global Continuum Placement component, described in D4.1 is forwarded through the global Orchestrator by using Open Cluster Management to the local cluster orchestrator, managed by Kubernetes. Then through the described technique using the webhook a particular scheduling algorithm is selected to better fit the needs of the execution to be performed, while of course respecting the constraints. Figure 18 shows a high-level view of the structure of the 2 scheduling levels as taken into account in PHYSICS. The scheduling algorithms implemented in this deliverable are related to the 2nd Scheduling level, which resides on the local clusters managed by Kubernetes, as mentioned previously.

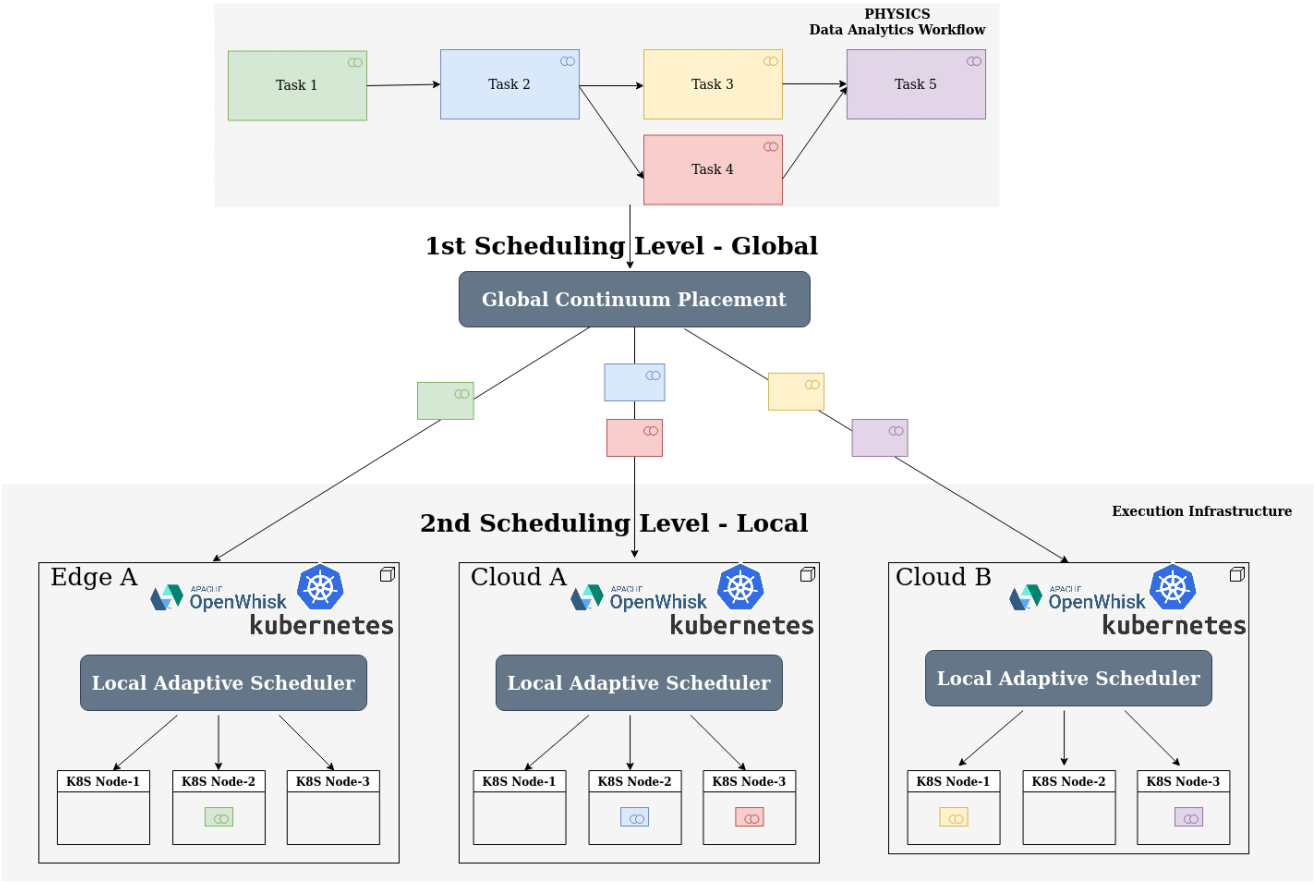

Figure 18. – High-level view of the 2 scheduling levels of the continuum as managed in PHYSICS

The design of our scheduling algorithms is based on the experimentation and simulation methodology that we described in the previous section. In particular the study of scheduling algorithms needs to be done using particular platforms which will allow us to simulate different aspects of scheduling under various contexts. For that we make use of platforms such as Simgrid [51] and Batsim [52] which simplify the simulation of distributed systems and in particular the scheduling on complex hybrid infrastructures which is our focus here. Hence the design of our algorithms in parallel with the preparation of our simulation platforms is an important aspect of our study.

After the analysis described in section 5.1 we started designing our first scheduling policies that seemed most interesting based on our initial results. These policies focused on the placement of tasks-functions based on the locality of the containers or its layers as required by the functions.

Hence, our initial designs of scheduling algorithms are based on the locality of the containers and their layers and are named CacheLocality. These scheduling policies search for available resources and among them, search for the container required by the particular function. If found, the function execution will benefit from a reduction of the time needed to download and deploy the required container. The goal of this policy is to explicitly search for containers, since we have seen in our studies the benefits of avoiding having to download a container prior to an execution. In this context, there are many possible scenarios of investigation.

For instance, it is possible to reverse the order of the priorities of such a policy, and first search for nodes that already have a required container and then to check if such nodes are available or not. This can result in two different policies: the first one can enforce the function to wait for the nodes to become available, once the ones with the required container are found but are currently utilized; the second possible policy

would not wait for the nodes to become available and would just download a new instance of the container in an available resource.

Going further, we also designed variations which take into account the existence of particular layers of a container again following the similar choices like in the whole containers case. When we say layers here, we mean Docker container layers. Since each container is based on layers, if the needed container uses layers from other containers that already exist then its download time is minimized to the time needed to download only the layers that are still missing. This can greatly speed up the download of a container (and also save overall space on disk needed as we avoid downloading the same layers in other nodes). Such variations of the CacheLocality policy were designed, implemented and studied and are described more thoroughly in the following subsections .

5.3 Implementation details and Integration highlights

5.3.a CacheLocality Scheduling algorithms variations

As described before we focused first on the CacheLocality policies and its variants in order to try to address the various delays that we may have when downloading containers and their layers. In order to adapt to the different contexts, we created several variations of the CacheLocality policy as described right beneath:

a) **Algorithm1 CacheLocality:** it looks for the available machines that already have the required container and selects the first one that fits on these requirements. If the required container is not found among these available machines, the container will be downloaded in the first available machine.

```
Algorithm 1: CacheLocality
Require: functions_queue, machines_available
    while  functions_queue is not empty do
        f←functions_queue[0]
        container_required←f.container
        machines_cadidates←sort(machines_available, container_required)
        m←machines_candidates[0]
        allocate(f, m)
    end while
```

Code 13 - Cache locality algorithm

b) **Algorithm2 CacheLocalityHard**: it looks for machines that already have the required container, and it selects the first machine that is available among them. If none of them is available, but the container exists on at least one of them the function is forced to wait until one of the machines that already has the container becomes available. This behavior avoids as much as possible repeated download of containers;

```
Algorithm 1: CacheLocalityHard
Require: functions_queue, machines_available
    while  functions_queue is not empty do
        f←functions_queue[0]
        container_required←f.container
        machines_cadidates←sort(machines_available, container_required)
        m←machines_candidates[0]
        while m is not available do
             sleep()
```

```
        end while
        allocate(f, m)
    end while
```

Code 14 - Cache locality hard algorithm

c) **CacheLocalityWithLayers:** it looks for the available machines that already have the required container and selects the first one that fits on these requirements. If no complete image is found, it looks for container image layers that can be used from other available containers and the task is scheduled on the machine that has the most shared layers while it downloads the remaining ones. If neither the required container nor usable layers are found among these available machines, the container image will be downloaded in the first available machine.

```
Algorithm 3: CacheLocalityWithLayers
Require: functions_queue, machines_available
    while  functions_queue is not empty do
        f←functions_queue[0]
        container_required←f.container
        for m in machines_available do
           m.score←score(m, container_layers_required)
        end for
        machines_cadidates←sort(machines_available, container_required)
        m←machines_candidates[0]
        allocate(f, m)
    end while
```

Code 15 - Cache locality with layers algorithm

d) **CacheLocalityWithLayersHard**: it searches at first for machines with the required container, if there is none, it searches for container image layers from other available containers that can be shared. After listing the machines with the required containers or usable layers, it verifies if there are any machines available among them, if so, it is selected. If none of them is available, the function is forced to wait until such availability. This behavior avoid as much as possible repeated download of containers or layers;

it looks for the machines that already have the required container and selects the first one that fits on these requirements. If no complete image is found, it looks for container image layers that can be used from other available containers and the task is scheduled on the available machine that has the most shared layers, while it has to download the remaining ones. If none of them is available, but the container or some of the needed layers exist on at least one of the machines, the function is forced to wait until one of the machines that already has the container or some layers, becomes available.

```
Algorithm 4: CacheLocalityWithLayersHard
Require: functions_queue, machines_available
    while  functions_queue is not empty do
        f←functions_queue[0]
        container_required←f.container
        for m in machines_available do
           m.score←score(m, container_layers_required)
        end for
        machines_cadidates←sort(machines_available, container_required)
        m←machines_candidates[0]
        while m is not available do
```

```
            sleep()
        end while
        allocate(f, m)
    end while
```

Code 16 - Cache locality with layers hard algorithm

In addition, a baseline policy has been created which was called **AlwaysDownload**, that will always download every container for all functions, even if containers already exist in the machines. This is a basic policy (that can happen in specific contexts, as for example, in applications that enforce security, not re-using containers).

The implementation of our first prototype of algorithms has been done initially within the simulator so that we can study their behaviour in different contexts. However, we have studied and analysed the internals of Kubernetes scheduler and we are currently implementing the above algorithms as new scheduling plugins within Kubernetes. In particular, based on the details given in [50] and [25], we are working on adapting the existing ImageLocality plugin in order to provide the above variations of CacheLocality.

These are only some first algorithms to be implemented and further policies will be evaluated such as the ones that prioritize warm or hot containers. For that there are some parts of scheduling that take place within Openwhisk and hence in that case both Openwhisk and Kubernetes schedulers will need to be modified.

Finally, most of the different scheduling constraints and parameters forwarded from the Global Continuum Placement are taken into account by default by using the typical parameters of defining a task/pod execution. Furthermore, in relation to the co-allocation strategies of task T5.4 the scheduler will set the needed affinities and antiaffinities as an additional filter based on the related inputs. If further adaptations are needed we will need to modify the definition of task/pod scheduling and take this into account within new adapted scheduling algorithms.

5.3.b Layers Locality Kubernetes Scheduling algorithm

Based on the CacheLocality algorithm we have implemented the Layers Locality scheduler on Kubernetes which aims to minimize the delays due to image downloading for function execution: minimizing the Cold starts of functions. In this context, Kubernetes already provides an ImageLocality plugin which takes into account the existence of images on particular nodes. Following the same path, we have implemented a variation of ImageLocality plugin taking into account the existence of Containers' Layers and trying to favor the execution of functions on nodes where layers of the containers to be deployed already exist. The new scheduler is named LayersLocality. The LayersLocality scheduler is going one step forward by scoring nodes using the percentage of the container images already available and thus taking advantage of the image layer caching.

Here are some advantages and limitations of this scheduler:

Advantages

- May reduce significantly the deployment time of large images for workloads with containers that share container layers.
- It is based on the same mechanism as Image locality scheduler plugin so since it does not make huge changes in the code it has more chances to be accepted by the Kubernetes community in the upstream version, hence increasing the impact of it.

Limitations

- Only useful when the containers in the workload are sharing layers.
- Like in the image locality plugin, it requires some spread policy to avoid allocation to always select the same node (the one with large layers already present)
- The first time a container image is encountered by the scheduler, it doesn't know its layers' characteristics until it is pulled for the first time by the CRI. Thus, this scheduling optimization only kicks in when a container image is deployed for the second time.

On the more technical side, the main issue for this plugin is to acquire the layers size and locality. For this implementation we need the following:

- To get the available layers on each node (name and size)
- For each new pod compute a score per node considering the cumulative size of already available layers

For this we had to go through 2 different software of the Kubernetes hierarchy. The first is Kubernetes itself and second is the Container Runtime Interface (CRI). In our case we consider mainly CRI-O which is one of the most known and used Kubernetes CRI. Hence in this regard, we had to make changes on various areas of the different involved software without breaking retro-compatibility:

- So we made changes in the Kubernetes internal interfaces to add the Layers info into Kubernetes: optional Layers field in the core.v1.Node.NodeStatus API and add a new field in the scheduler snapshot to have it exposed to the plugin.
- while modifying the Container Runtime CRI-O to get available layers name and size on node and send them through annotations (without API change)

Exposing layers info for each of the nodes might have an impact on the message size, and thus the overall Kubernetes performance, but most of the images have only 2 or 3 layers (and they might be shared) so the overhead should be very limited. Also, to cope with this issue, we can add some mechanism to make the CRI only expose layers if their size is above a certain threshold or simply expose the N biggest layers. These are all possible enhancements under evaluation.

The implementation of the new scheduler composed by adaptations on the two different tools has be done initially upon specific forks of the upstream software code repositories and currently reside in the following forks: one for Kubernetes[10][11] and one for CRI-O,[12]-[13]. Furthermore, we have pushed the LayersLocality Scheduler to the Kubernetes community[14] by proposing to integrate our changes in the upstream version. In particular we are in contact with the SIG-Scheduling[15] group which is responsible for the different upgrades related to the Kubernetes scheduler and we hope to have this accepted and integrated to simplify the usage of it directly through the open Kubernetes version.

The Kubernetes LayersLocality scheduler has been integrated to the rest of the components through the webhook described in 4.1.g and it has been installed and configured on the PHYSICS testbed and in particular on the Azure-based cluster of the continuum where it can be leveraged by the different FaaS applications and use cases running on the clusters.

[10] https://github.com/RyaxTech/Kubernetes
[11] https://github.com/RyaxTech/Kubernetes/compare/v1.22.6...image-layer-locallity-scheduler
[12] https://github.com/RyaxTech/cri-o
[13] https://github.com/RyaxTech/cri-o/compare/v1.22.1...image-layer-locality-scheduler
[14] https://github.com/Kubernetes/Kubernetes/issues/120672
[15] https://github.com/Kubernetes/community/issues/7379

5.4 Experimentation Outcomes

5.4.a Scheduling algorithms experimentation and performance evaluation

To evaluate our scheduling algorithms we have performed simulations based on Batsim-Simgrid. For that, we modeled the FaaS applications to be executed on top of Batsim. We created workloads specific for Serverless Functions, using the FunctionBench benchmarks, specifying their execution time, the containers name, tag, and list of layers - the different layers size along with the cpu processing time and memory required will be added in next steps. With that, on simulation time, we can know how long the function will take and how long it will take to download and deploy a container. With the information of the layers, we were able to compute a ratio with the existing layers in the different machines, to reduce their download and deployment time based on the amount of common layers. In this simulator we were also able to model different platforms, with a number of machines and nodes - network configuration will be added in next steps.

Concerning the modeling of the computing infrastructure, we simulated the same computing infrastructure used for the real FaaS execution experiments, which took place in the first phase. To do the simulation we made use of the SimGrid features which allows us to define in detail the different characteristics of our simulated environment. Hence with the usage of both Batsim-Simgrid we managed to execute simulated experiments on our scheduling policies and managed to get quite promising results.

We have run several experiments to compare the different implemented versions of CacheLocality scheduling Policy. In the following it is presented a simple example to illustrate how such policies can reduce the makespan of the platform, reducing the container or layers download. Figure 19 shows the execution time and the container download time, both in seconds, for three serverless functions, with specific input, used in our benchmarks. It shows the ratio is non-negligable between the time needed to download and deploy the containers and the time to execute the functions.

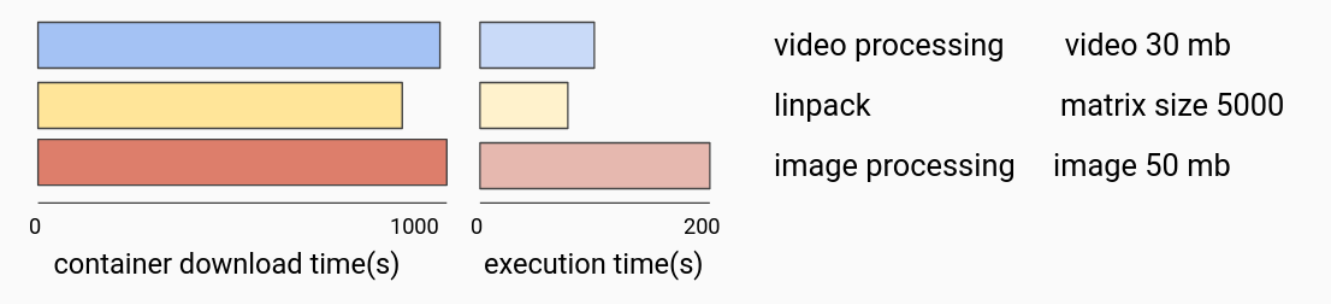

Figure 19. – Execution and container download time

In addition, it is important to emphasize that it was investigated that video processing and image processing container images share about 80% of their layers, and linpack shares 0% on the contrary. Figure 20 illustrates such functions grouped in a workload, with three invocations of video processing, one invocation of linpack and one of image processing, followed by their submission time.

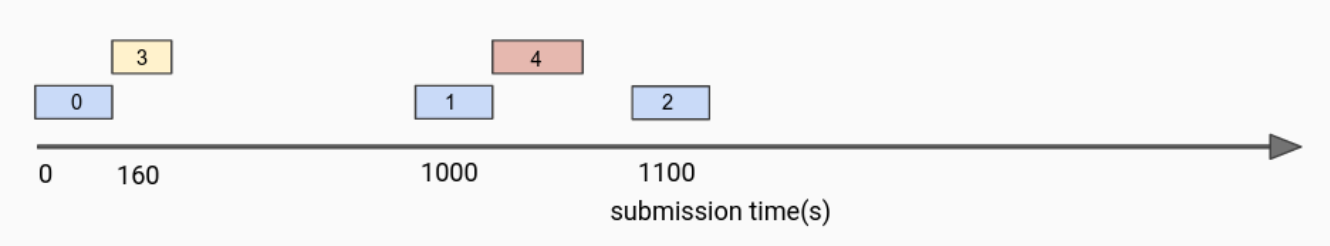

Figure 20. – Functions grouped

It was defined as a platform with two machines and it was simulated two different variations of the CacheLocality scheduling policy: c) CacheLocalityWithLayers and d) CacheLocalityWithLayersHard, in addition to the AlwaysDownload policy. The following figures will present the functions as the numbers on Figure 21, <function_id>, and their containers name will be composed by <container_name>_<job>_<function_id>_<container_counting>. For instance, function *0* will be preceded by *python3action_video_processing_job0_0*.

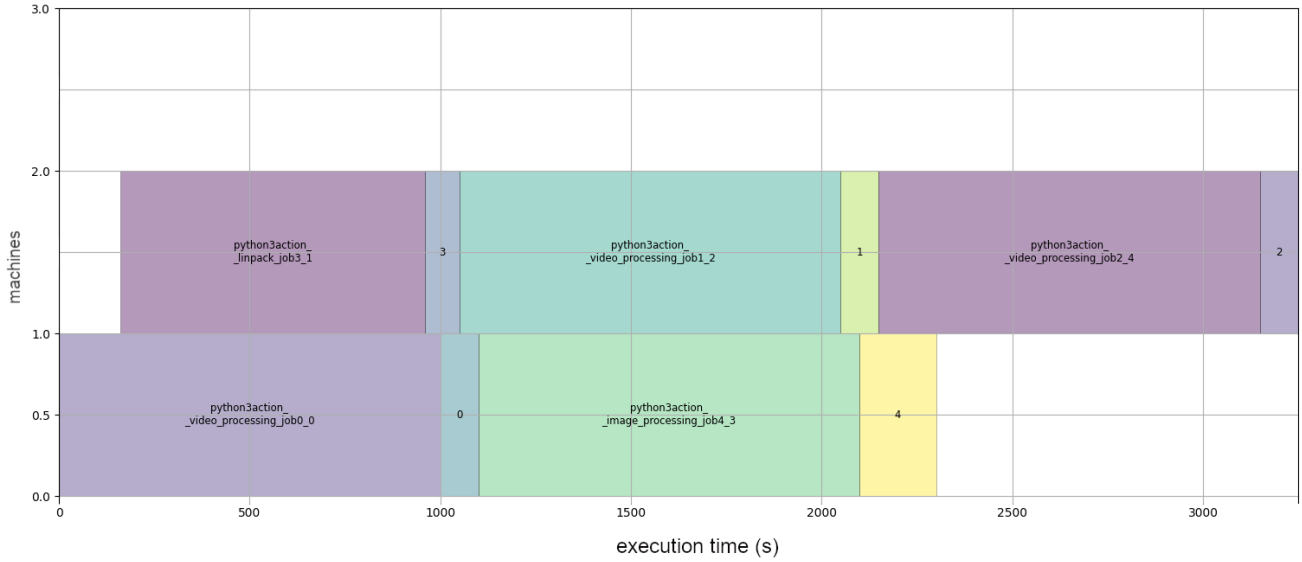

Figure 21. – Always policy result

Figure 21 shows the result of the Always Download policy. It is possible to see that as designed, all functions download their containers, even if the machine already executed the same function previously. This is basically to show the behavior in the worst case scenario which is something possible.

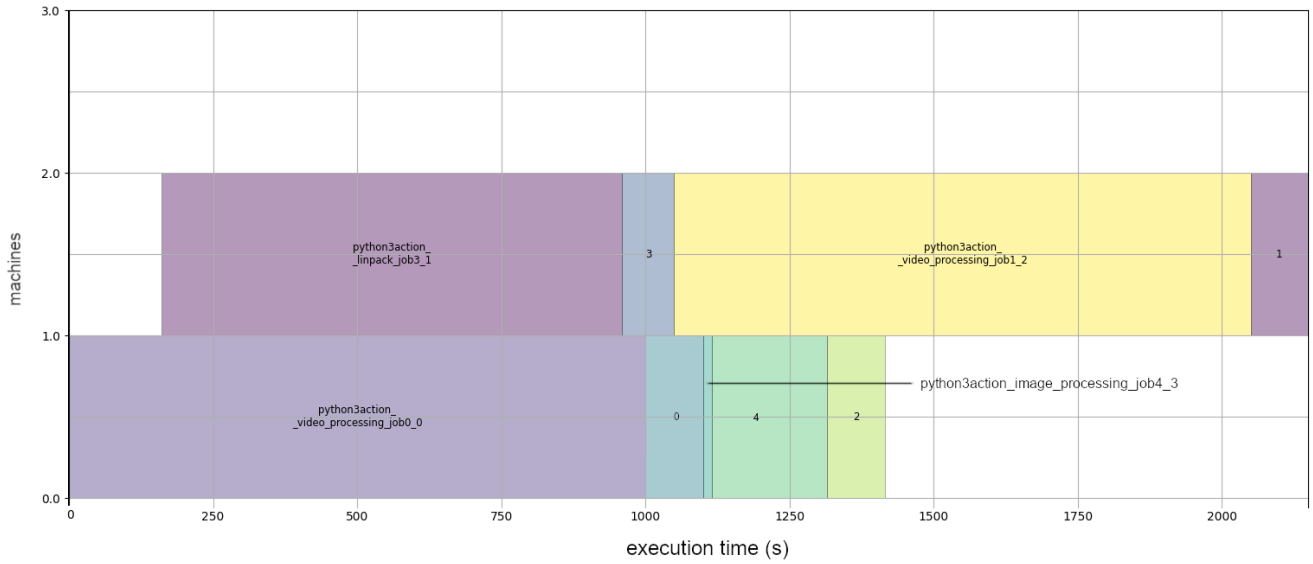

Figure 22. Cache locality with layers results.

Figure 22 shows the results of the Cache Locality with Layers policy. This policy searches at first for machines available and after for layers to be shared. If the machines available do not have any useful layer, the entire container will be download there anyway, as the python3action_video_processing_job1_2 of function 1, which found machine 0 busy when it was submitted, and then was allocated to machine_1 even without any useful container layer. On the contrary, it is possible to see that the container python3action_image_processing_job4_3 of function 4 had its download time considerably reduced in comparison with expected. It happened due to the sharing of 80% of layers with the python3action_video_processing_job0_0 of functions 0 and 2. And finally, it is visible that function 2 does not have any container directly before it because it shares the same container used by function 0

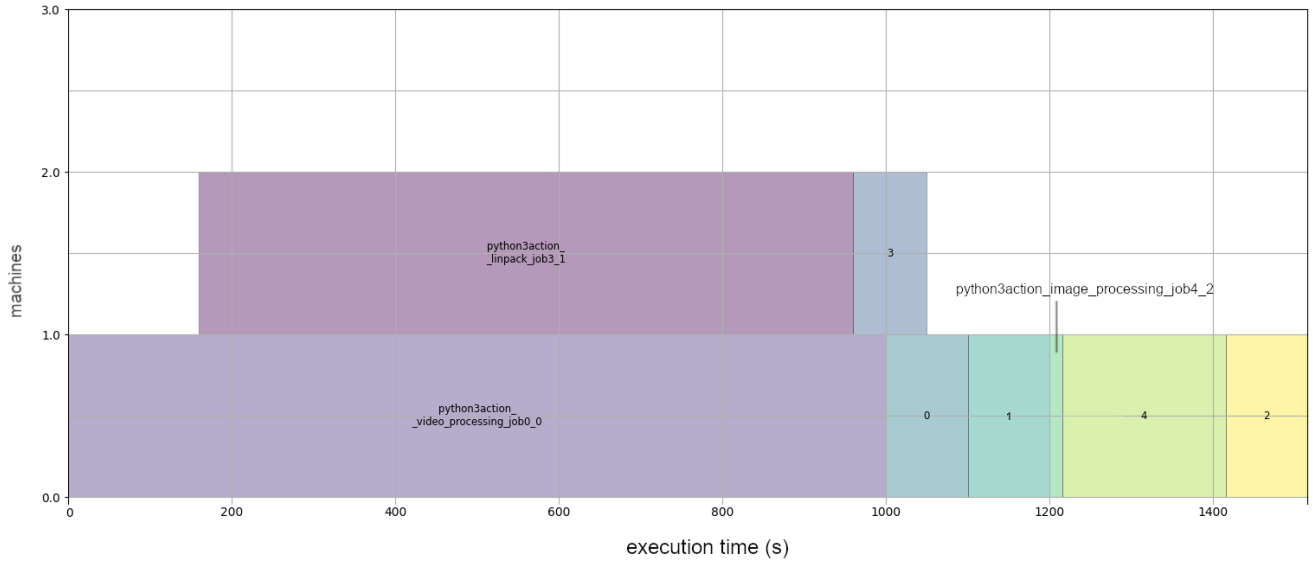

Figure 23. - Cache locality hard with layers results

Figure 23 shows the results of the CacheLocalityHardwithLayers policy. This policy searches at first for machines with layers to be shared and once found, select the best ones and check if they are available. If the selected machine is not available, the function will wait until the machine gets available and it will reduce the download of new containers or layers as much as possible.

For instance, we can see that only three containers were downloaded: *python3action_video_processing_job0_0*, *python3action_linpack_job3_1*, and *python3action_image_processing_job4_2*, which represent one container per type of function. It is possible to see that in comparison with Figure 22, the execution of function *1* was delayed to be executed on *machine_0* that already had the required container. Then the function *4*, image processing, followed the same behavior as on Figure 22 and was allocated to a machine that could share layers from other containers.

In addition to the different behavior of each illustrated policy, it is possible to observe that the makespan was reduced in both CacheLocality variantes in comparison to the AlwaysDownload baseline, which is around 3500s for the last one, and around 2500s for the CacheLocalityWithLayers and 1500s for the CacheLocalityHardWithLayers. So it is possible to conclude that the sharing of layers can reduce the makespan of any workload if there are functions requiring compatible container images. Furthermore, we can conclude with this simple example that a more rigid approach such as delaying the execution of functions to benefit the sharing of possible container layers can produce an interesting tradeoff between increased functions waiting time and the makespan of the platform. Of course this is something that we need to validate with simulations.

Hence the above results showed us the interest of our implemented algorithms and since the simulations have presented good results, the plan is to continue with the implementation of such policies in the real platforms. Kubernetes is the main layer where we are going to implement a new scheduler. Studying the Kubernetes Scheduler, which is based on predefined Policies and Profiles, we can modify the standard Profiles to use different Policies, and there is one Policy named Image-Locality that implements a behavior similar with the one we developed on top of our simulator. This will be adapted and enhanced to take into account the locality of the layers.

5.4.b Kubernetes LayersLocality Scheduler experimental validation procedure

Following the LayersLocality implementation on Kubernetes, we hereby provide the procedure for the experimental validation of the new scheduler. For this we will go through Kind [14] which is a tool for running local Kubernetes images based on Docker containers for “nodes”.

First we need to build CRI-O and Kubernetes using our forked versions as explained in detail in the tutorial we created here[16]. Then we can create a configuration for the kind cluster to use CRI-O and our custom images:

```
cat > kind-crio.yaml <<EOF
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  image: quay.io/aojea/kindnode:crio1639620432
  kubeadmConfigPatches:
  - |
    kind: InitConfiguration
    nodeRegistration:
      criSocket: unix:///var/run/crio/crio.sock
  - |
    kind: JoinConfiguration
    nodeRegistration:
      criSocket: unix:///var/run/crio/crio.sock
```

[16] https://github.com/RyaxTech/k8s-container-layer-locality#testing

```
  - |
    kind: ClusterConfiguration
    KubernetesVersion: 1.22.6
    imageRepository: docker.io/ryaxtech
- role: worker
  image: quay.io/aojea/kindnode:crio1639620432
  kubeadmConfigPatches:
  - |
    kind: JoinConfiguration
    nodeRegistration:
      criSocket: unix:///var/run/crio/crio.sock
- role: worker
  image: quay.io/aojea/kindnode:crio1639620432
  kubeadmConfigPatches:
  - |
    kind: JoinConfiguration
    nodeRegistration:
      criSocket: unix:///var/run/crio/crio.sock
EOF
```

Code 17 - KinD configuration for the validation of LayersLocality scheduler

Then create the cluster:

```
kind create cluster --name crio --config kind-crio.yaml
```

Code 18 - KinD cluster creation

Inside the cri-o repository, replace the Cri-o executable by ours with:

```
for n in $(kind get nodes --name crio); do
  docker cp ./result/bin/crio $n:/usr/bin/crio
  docker exec $n systemctl restart crio
  sleep 1
  docker exec $n systemctl status crio
done
```

Code 19 - CRI-O executable modification

Check if this work with:

```
docker exec -ti crio-control-plane crictl --runtime-endpoint unix:///var/run/crio/crio.sock images
-o json
```

Code 20 - CRI-O deployment

You should see this kind of annotations:

```
"annotations": {
          "imageLayer.sha256:b0e18b6da7595b49270553e8094411bdf070f95866b3f33de252d02c157a1bc7":
"15879307",
          "imageLayer.sha256:d256164d794efdde4db53b59b83dd6c13cabf639c7cac7b747903f8e921e32c9":
"23796084"
        }
```

Code 21 - CRI-O new annotations

Inside the Kubernetes repository, push the kubelet in all nodes:

```
for n in $(kind get nodes --name crio); do
  docker cp  _output/local/bin/linux/amd64/kubelet $n:/usr/bin/kubelet
  docker exec $n systemctl restart kubelet
  sleep 1
  docker exec $n systemctl status kubelet
done
```

Code 22 - kubelet update

Fix the coredns image name (use a hardcoded subpath `coredns/coredns` unsupported by dockerhub):

Inside the Kubernetes repository, push the kubelet in all nodes:

```
kubectl set image -n kube-system deployment/coredns coredns=docker.io/ryaxtech/coredns:v1.8.0
```

Code 23 - new Kubelet on nodes

And now we are ready to test with the following (or similar) scenario. We can consider two images which are going to be composed by at least one or two same layers for example:

- Img1: Layers: L1, L2, L3
- Img2: Layers: L1, L2, L4

Then we can consider three homogeneous nodes: N1, N2 and N3 and two pods: P1 using Img1 and P2 using Img2 both requesting all the resources of one single node. Now let's consider an empty cluster with no layers of Img1 and Img2 in cache.. At time `t0` we submit P1 which will be placed to node N1. Then at time `t1` we submit P2 which will be placed to node N2. Then at time `t2` we remove P2 and at time `t3` we submit P1 again. Now since it will be using the LayersLocality scheduler, the P1 will go to node N2 because some of the layers are already present and not on N3 which does not have any layers. This will accelerate the download of images, especially if the layers have large size.

The Kubernetes LayersLocality scheduler has been installed and configured on the PHYSICS testbed where it is validated through similar experiments and the real use case applications.

5.5 Next Steps

In previous sections, we described the different aspects of the studies that took place during the design, implementation and experimentation of the local-level scheduling algorithms of our global continuum. We have started with an initial analysis of the way FaaS applications are scheduled using default Openwhisk-Kubernetes scheduling techniques and we have tried to understand what is needed to further improve various aspects.

In addition we have provided an adapted simulator along with an experimental methodology to study the different scheduling algorithms. Based on our initial outcomes we have designed some first policies to address the delays due to downloads of containers in clusters where the containers or layers of the containers exist already. Hence the placement is adapted based on which resources provide particular containers or layers. We have implemented different variations of these policies and we have evaluated them using the simulator which showed promising results.

Furthermore, we have studied the Kubernetes scheduler and its internals and we have implemented the LayersLocality as extensions of the ImageLocality plugin of the Kubernetes scheduler, while integrating it with the currently designed Physics local-cluster architecture connecting with the webhook and the different Kubernetes APIs.

As we move forward, our focus is to follow-up, the effort that has already started, with the SIG-Scheduling group, in order to finalize the adoption of our new Kubernetes scheduler in the upstream version. This will allow us to minimize the cold start delays for FaaS applications and increase the impact of our scheduler.

Our studies will also continue after the end of the project to explore, design and implement new scheduling policies based on various new parameters as defined within the higher layers such as the Design Environment, the Global Continuum Placement, the Performance Evaluation Framework and the Orchestrator -- from WP4, see D4.2. One of the important aspects that may be explored in the future is the selection of warm and hot containers instead of cold ones to deploy the tasks/functions. For this particular adaptation, work will be needed not only within the Kubernetes scheduler, but also within the Openwhisk scheduler.

6. Optimized Service Co-location Strategies

The co-location strategies component, developed under Task T5.4, Optimized Service Co-location Strategies, is in charge of finding the most suitable placement decision for a new Pod to be deployed, taking into account the workloads that are running in a cluster in order to reduce the *noisy behavior effect.* The co-allocation strategies look at the complementarity of deployed Pods, that is, CPU intensive, memory intensive, or network intensive, but also at the requirements defined by the application at design time in WP3. Some functions may require a given hardware to be executed (e.g. GPU), some functions are not isolated, they are part of a workflow (e.g. a sequence of function invocations) that may run in the same node. Some functions may use a service and should be placed together with the service for improved performance. Those requirements are expressed with annotations by the applications designer in WP3 Functional and Semantic Continuum Services Design Framework. The application developer can also follow the Performance Pipeline Stage (described in D3.2) that leads to annotated information regarding the expected behavior of a function (CPU/memory/IO/network resources needed). All this information is passed to the infrastructure where a concrete pod will be deployed and the associated YAML file will be updated to include Kubernetes node and inter-pod affinity rules [53], which are used by the scheduler. The scheduler will decide the placement of a concrete pod based on these rules The node affinity constrains the nodes where the pod can be executed. For this purpose, nodes are labeled with key and value pairs, and the pod YAML will use that label and indicate if this is a required or preferred node. Inter-pod affinity and anti-affinity rules constrain the nodes where a pod can be deployed based on the labels of the pods that are already running on the cluster. The coallocation component will be in charge of generating these affinity rules. Moreover, statistics regarding the actual consumption of resources the pods in the cluster are collected and used to classify pods according to their CPU/IO/Network/Memory, so that future executions of the same type of pod can take this information into consideration. The infrastructure will also collect aggregated metrics of the resource usage for each node. This information will allow us to approximate the resource consumption of nodes. The co-allocation component will use this information to produce the affinity rules for each pod to be scheduled.

The rest of the section presents the design of this component, and its implementation, the information it uses and produces, and an example to understand its behavior.

6.1 Design Specification

The goal of the co-location is to find a set of candidate nodes for the deployment of new pods based on the current running nodes (workload) in a given cluster. The co-location component is made out of 6 subcomponents as depicted in Figure 24. The subcomponents are: a database, processes that run periodically (cron job pods) and processes that run when a new pod is created (Section 4.1.g). The cluster status and function execution metrics are stored in the *co-location database,* a Prometheus database. Three subcomponents run periodically: *Cluster information*, *Cluster status*, and *Function metrics* and *interferences.* The d*ata collection* and the *Rules generator* are executed before a new pod is created.

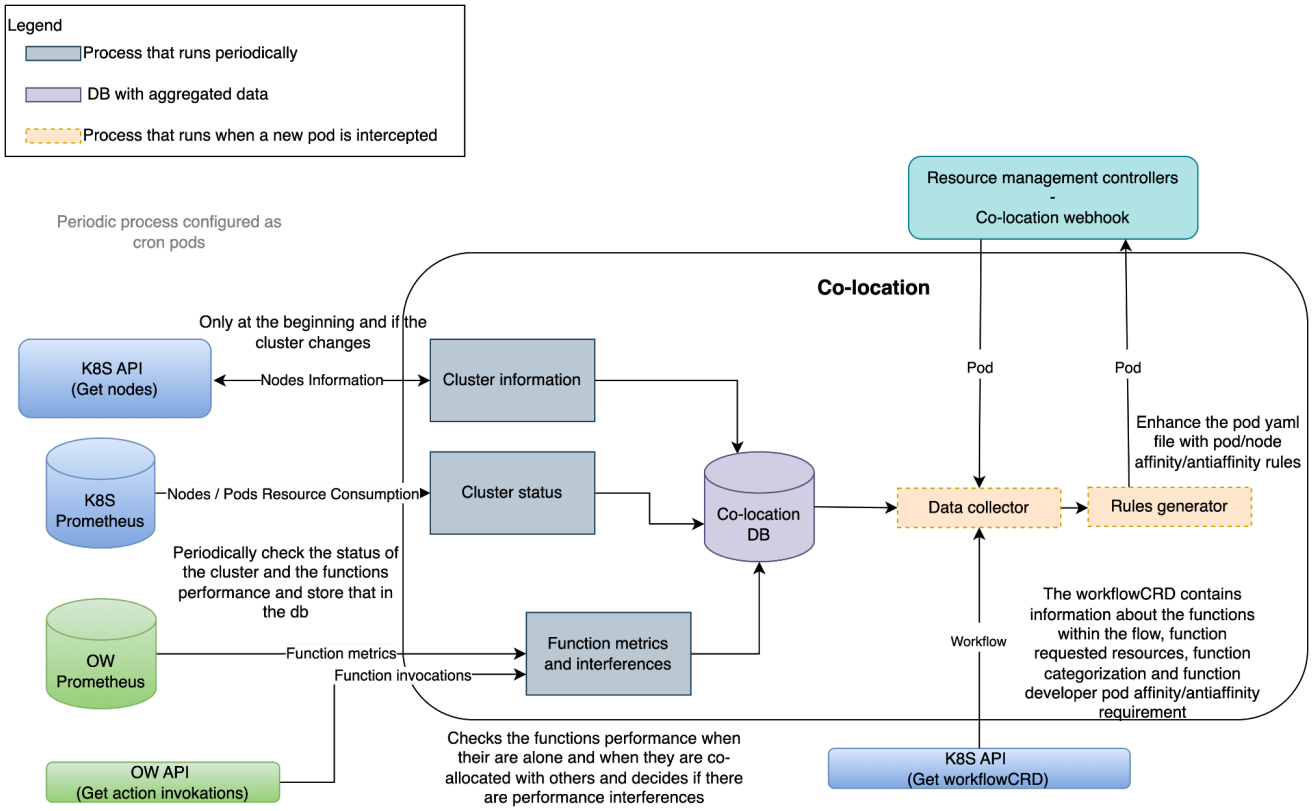

Figure 24. Co-location component architecture

The *Cluster information* process gets information about the nodes available in the Kubernetes cluster and their technical specifications (CPU, memory, disk type, GPU and network bandwidth). This information is obtained through the Kubernetes API. This process is executed when the Co-location component is deployed and periodically to check if the cluster status has changed. The information is stored into the *Co-location database* to be used by other subcomponents. The *Cluster status* subcomponent periodically checks the resource consumption of the nodes and the pods that are running in the Kubernetes cluster. This information is obtained from the Kubernetes Prometheus instance and is stored into the *Co-location database*. The *Function metrics and interferences* subcomponent gets information about the functions executed in the FaaS platform such as the execution time and the invocation time of the function to detect interferences with other functions executed at the same time in the same node. This information is kept in memory and also stored in the Co-location database to be used to generate the affinity/anti-affinity rules.

The *Data collector* subcomponent is executed each time the webhook intercepts the creation of a new Pod. This subcomponent retrieves the information from the Co-location database and the Workflow CRD and selects the set of nodes where the function can be allocated so that performance is not degraded. The subcomponent finds the nodes with the needed hardware resources and among them selects the nodes in which the function executed previously with the lower response time if the same functions are running concurrently. Otherwise, if there is no previous information, the less loaded node is selected. If none of these conditions are met, the nodes that have more available resources are recommended. The Rule generator subcomponent updates the pod yaml object to add the affinity and anti-affinity rules that will be used by the scheduler to deploy the pod.

6.1.a Cluster information

The *Cluster information* subcomponent retrieves the features of the nodes that belong to the cluster. This subcomponent is executed periodically, i.e., once per day. The component uses the Kubernetes Python library, this a Python client for the Kubernetes API. The component gets features of each node registered in the Kubernetes cluster, the memory, the number of cores and all the labels set to the nodes like GPU, disk type and the DMS component keydb label. The labels have to be set by the cluster administrator. This information is stored in the Co-location database.

6.1.b Cluster status

The *Cluster status* subcomponent collects the CPU, memory, network, and disk metrics of each node within the Kubernetes cluster. This component is executed periodically, i.e. every 5 minutes. The metrics are provided by Kubernetes and accessed using the *prometheus_api_client* and *Kubernetes* Python libraries. These metrics are stored in the Co-location database.

6.1.c Function metrics and interferences

The *Function metrics and interference* subcomponent is executed periodically, i.e., every 5 minutes. This component gets information about the execution and starting time of the functions executed in the FaaS platform during the last 5 minutes and updates the database.

The OpenWhisk(OW) client(*wsk*) is used to retrieve the information about the functions executed in the last 5 minutes:

```
$ wsk -i activation list --since 1694428612 --upto 1649928912 --limit 200
```

This command returns the list of functions (up to 200) executed from timestamp 169442861 till timestamp 1649928912. Once the activation list is collected, the *function metrics and interference* sub-component analyzes the activations of each function and checks if the function has been executed alone or collocated with other functions. The component gets the information about the pod which runs the function and where the pod runs. Then, the latency, the CPU usage, the number of vcores, memory and network I/O used by the function are stored into the database.

6.1.d Co-location Database

The Co-location database is a central component in the co-location component used by the rest of the subcomponents. The information is stored into the temporal series Prometheus database each time the process that collects the information (Cluster information, cluster status and Function metrics and interference components) stores their output in the database.

6.1.e Data collection

The *Data collection* subcomponent is executed each time a pod creation is intercepted by the mutating webhook. This component receives the pod YAML object, which contains the workflow CRD name in the metadata>annotations>workflow field. This sub-component accesses the workflow CRD object the pod belongs to. The workflow CRD stores the workflow structure (function execution order) and annotations related to the workflow as shown in Code 24. For instance, if all functions must or should be executed in the same node (affinities) or in different ones (anti-affinities).

The relevant information extracted from the workflow CRD is:

- Cluster: This annotation is required to know if all the functions of the workflow are executed in the same cluster. In case the functions are executed in the same cluster, the component has to take into consideration interferences of the different functions with the ones running in the cluster.
- optimizationGoal and importance: If the optimization goal is performance, the component has to minimize the interferences of the function with the ones that are deployed in the cluster.
- extraResources: Allows to filter the nodes from the available ones and create nodes affinity/anti-affinity rules.

```
apiVersion: wp5.physics-faas.eu/v1alpha1
kind: Workflow
metadata:
  name: hello-sequence
  namespace: physics-namespace
  annotations:
       id: "19fe4293742e0b2c"
       version: "1"
       cluster: cluster1
spec:
  execution: NativeSequence
  listOfActions:
       - id: 339d2ef8b0b29795
       - id: 3a807141f16764a5
  native: true
  platform: openWhisk
  type: flow
  actions:
  - name: hello
       description: "hello"
       id: 339d2ef8b0b29795
       version: 1.0.0
       runtime: nodejs
       code: "function main(msg) {\nmsg.payload=msg.payload+' hello';\nreturn msg;}"
       annotations:
       optimizationGoal: Performance
       importance: "High"
       resources:
       limits:
       memory: 128
       requests:
       cpu: 1
       memory: 128
       extraResources:
       gpu: true
       diskType: ssd
       performanceProfile:
         cpu: medium
         memory: low
         networkTransmitted: low
  - name: world
       description: "world"
       version: 1.0.0
       id: 3a807141f16764a5
       runtime: nodejs
```

```
      code: |
      function main(msg) {
      //implies affinity with the other function in the sequence
      msg.payload=msg.payload+' world';
      return msg;
      }
      resources:
      limits:
      memory: 256
```

Code 24: WorkflowCRD example

Then the algorithm first restricts the candidate nodes based on the hardware requirements to reduce the search space. For instance, if the new pod requires a GPU, nodes in the cluster without a GPU are not taken into consideration. The nodes that fulfill the hardware requirements are considered as candidates.

Next, the algorithm gets the functions that are running in candidate nodes at this time and checks for interferences. If the optimization goal specified for the Pod to be deployed is **performance** and the importance is *high* the algorithm discards the nodes where functions that may cause interference are running. If there are not enough nodes or the importance is lower the algorithm minimizes the interference selecting the nodes with no interferences or with functions that can cause a low impact in the performance of the function running in the pod to be deployed.

When a new function arrives to the cluster and there are no records about how the function behaves in this cluster, the profile annotations available in the WorkflowCRD are considered. First, the algorithm is going to consider executing the function alone to analyze the resources consumption. If this is not possible due to all nodes in the cluster having other functions deployed, the algorithm selects the nodes with functions that have different behavior/profile to minimize the impact in the performance. From then on, this new function will be monitorized by the *Function metrics and interferences component.* If there are records about the function the algorithm uses the interference registry to select the most suitable nodes.

6.1.f Rules generator

Based on the outcome provided by the *Data collection* subcomponent, a set of affinity/antiaffinity rules are created. The hardware requirements are translated into node/pod affinities/antiaffinities that depending if they are strong constraints, the affinity will be *required* (*requiredDuringSchedulingIgnoredDuringExecution* in Kubernetes) or optional (preferred*DuringSchedulingIgnoredDuringExecution* in Kubernetes)[42]. Each node that is an outcome of the *Data collection* algorithm (candidate to deploy the new pod) are translated node affinity rules. The interferences detected by the Data collection algorithm are set as pod antiaffinity rules. Other requirements of the pod that come in the workflow such as some functions that should (not) be co-allocated, are added to the pod YAML file as inter-node affinity (anti-affinity).

An example of affinity and antiaffinity rules is represented in Code 25 where two node affinity and a pod antiaffinity rules have been set for the pod intercepted by the Mutating webhook that is going to deploy the hello function. The Node affinity rules have been set to satisfy extra resources requests of GPU and SSD disk. Then, the pod antiaffinity rule has been set to avoid the pod to be deployed within a node that has a function with the label openwhisk/action: Model_training. This type of antiaffinity rule is set due to interferences detected when the two functions are executed together. The algorithm detects that running the function to be deployed with model_training functions decreases the performance.

```
apiVersion: v1
kind: Pod
metadata:
```

```
  name: wskowdev-invoker-00-1-guest-hello
spec:
  affinity:
       nodeAffinity:
       requiredDuringSchedulingIgnoredDuringExecution:
       nodeSelectorTerms:
       - matchExpressions:
       - key: disktype
              operator: In
              values:
              - ssd
       - key: gpu
              operator: In
              values:
              - yes
       podAntiAffinity:
       requiredDuringSchedulingIgnoredDuringExecution:
       - labelSelector:
       matchExpressions:
              - key: openwhisk/action
              operator: In
              values:
              - Model_training
       topologyKey: topology.Kubernetes.io/zone
  containers:
  - name: wskowdev-invoker-00-3278-guest-hello
       image: "gkousiou/NodeREDhelloaction"
```

Code 25. Affinity/Antiaffinity rules example.

6.2 Co-location evaluation

We have run the Function Bench benchmark [54] using OpenWhisk in a Kubernetes cluster to measure the effect on performance of co-locating functions that compete for the same resource. Function Bench defines a set of functions (workload) that compete for resources. More concretely, we have run the functions on OpenWhisk both isolated, being the single pod in a node, and co-located in the same node with other function to show the effect on the function execution time of the co-location of functions that compete for the same resources (CPU and memory).

The evaluation was run in a Kubernetes cluster composed of 3 Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz Processor instances (24 virtual CPUs and 128GB memory), a TCP/IP emulation with 1 Gbps bandwidth. One machine is the master node and the other two are workers nodes. One of the workers executes all OpenWhisk master pods such as the controller. The other node was labeled as openwhisk-role=invoker and is where the OpenWhisk invoker is deployed and thus where all function pods will be scheduled. Some of the tested functions, like Model_training and video processing, require an external storage. To satisfy that requirement a MinIO [55] instance was deployed in the same node as the invoker to reduce IO latency.

The functions used for this evaluation are:

- Float[17]: This function performs the operations sine, cosine and square root in a loop of 100M iterations. The amount of memory is the default one, 256MB.
- Video_processing[18]: This function loads a 30 MB video stored in MinIO, transforms the video into gray frame and stores the new video into MinIO. The function was configured with 512 MB RAM.
- Model_training[19]: This function trains a scikit-learn linear model[20], LogisticRegression, with a CSV file of 50 MB. The CSV file is loaded at the beginning of the function execution and the trained model is stored into MinIO once the train process finishes. The function was configured with 512MB RAM.
- Model_serving[21]: This function loads a 5 MB video and a model available in MinIO and runs a model for face recognition. The RAM function is the default one, 256MB.

The assigned memory is enough to avoid running out of memory during the execution of the corresponding function.

A client invokes the function for 30 minutes. The average execution time, CPU usage, number of cores used, memory and network IO metrics were monitored during the standalone evaluation of each function. Table 1 shows the results.

	Execution Time (s)	CPU usage (%)	# vcores	Memory (MB)	Network Received	Network Transmitted
Float	6.13	5.36	0.16	25.6	26.5 B	17.24 B
Video_processing	97.9	5.82	1	219	201.8 KB	329.11KB
Model_training	36.5	27	4.5	201	223 KB	1.68KB
Model_serving	33.5	19	4.1	109.8	63KB	114KB

Table 1. Functions standalone execution results.

Next, functions have been executed in pairs monitoring the same metrics. Table 2 shows the results:

		Execution Time (s)	CPU usage (%)	# vcores	Memory (MB)	Network Received	Network Transmitted
Video_processing + Model_serving	Video_processing	120.5	30.3	0.89	241	159 KB	264 KB
	Model_serving	33.5		5.6	213.5	541 KB	161 KB
Video_processing + Model_training	Video_processing	121.4	38.04	0.93	219	174.2 KB	275.2 KB
	Model_training	36.6		7.82	262	725 KB	3.5 KB
Video_processing + Float	Video_processing	103.22	8.9	0.87	240	167 KB	269 KB
	Float	6.4		0.17	26.4	26.4 B	17.24 B
Model_serving +	Model_serving	38.9	52.67	5.13	212	706 KB	174 KB

[17] https://github.com/ddps-lab/serverless-faas-workbench/tree/master/openwhisk/cpu-memory/float_operation
[18] https://github.com/ddps-lab/serverless-faas-workbench/tree/master/openwhisk/cpu-memory/Video_processing
[19] https://github.com/ddps-lab/serverless-faas-workbench/tree/master/openwhisk/cpu-memory/Model_training
[20] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
[21] https://github.com/ddps-lab/serverless-faas-workbench/tree/master/openwhisk/cpu-memory/Model_serving

Model_training	Model_training	38.4		7.12	261	681 KB	285 KB
Model_serving + Float	Model_serving	33.61	27.9	5.7	210.7	495 KB	162 KB
	Float	7.3		0.18	24.8	26.8 B	27.9 B
Model_training + Float	Modelt_training	37.49	35.34	7.6	250	699 KB	288KB
	Float	8.3		0.2	25.2	25 B	16.3B

Table 2. Functions co-located execution metrics.

Figure 25 shows the latency in seconds obtained from the execution of the Video_processing function standalone, and co-allocated with other functions. The average latency of this function, during the standalone execution, is 97.9 seconds. This latency increases 5%(Co-located Float), 19% (Co-located Model_training) and 18%(Co-located Model_serving), when it's co-located. The number of cores used by this function remains stable (1 vcore) in all executions, instead, the network I/O decreases 21% and 13% when the function runs co-located with the Model_serving and Model_training functions respectively. Those functions also read and write in MinIO so the Video_training function is competing for the same resource. Analyzing the evolution of the network IO metric during the evaluations, when the video_processing function running in standalone the network consumption reaches 56% of capacity, when co-located with the other functions it is 54%(Co-located Float), 95% (Co-located Model_training) and 75% (Co-located Model_serving). This function must therefore be prevented from being executed with either of the other two functions. The float function response time increases only 5% when co-located with other functions. These two functions do not cause interference since the nodes have enough resources to run the two functions at the same time.

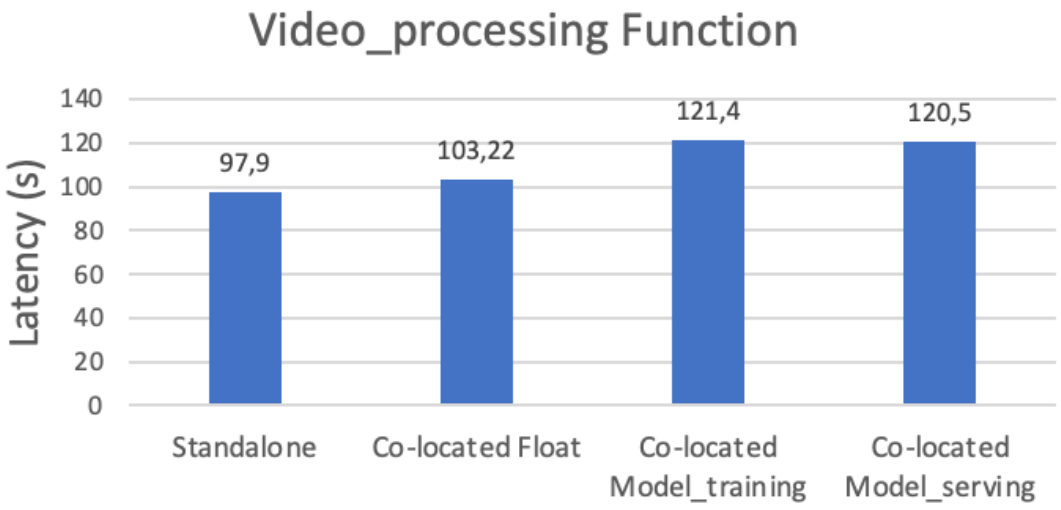

Figure 25. Video_processing function co-location.

Figure 26 shows execution time of the Model_training function standalone, and co-located with the float, Video_processing and Model_serving functions. This function running alone has an average response time of 36.5 seconds. The latency increases when co-located in 2% (float), 0.2% (Video_processing) and 4% (Model_serving). The increment is low in comparison with increment suffered by the Video_processing function. However, it is remarkable the increased usage of vcores when the function is co-located. The function running alone consumes an average of 4.5 vcores. However, when the function is running co-located, the Model_training function consumes 7.6 vcores (an increment of 40%) when running with

the Float function. Consumes 7.82 vcores when co-located with the Video_processing function (42%) and 7.12 vcores when running with the Model_serving function (37%).
This behavior occurs because the node has enough resources to execute all functions and increase CPU usage, a resource that is not limited by OpenWhisk functions. We have limited the CPU usage of the Model_training function to 2 vcores. For this purpose the Muttating webhook intercepts the pod YAML and adds the resource limit to 2 vcores. Figure 27 shows that the latency of the function when the resource usage is limited has increased by 36% compared to the execution without resource limit. If the user indicates that the performance is not so crucial , the function could be placed on a node that has less available resources or run it with another type of function competing for the same resources would increase the latency.

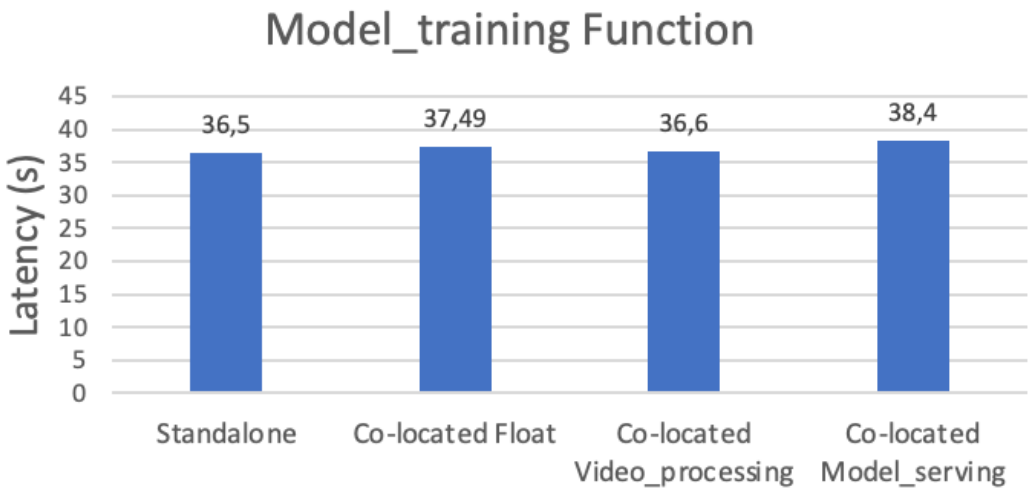

Figure 26. Model_training function co-location

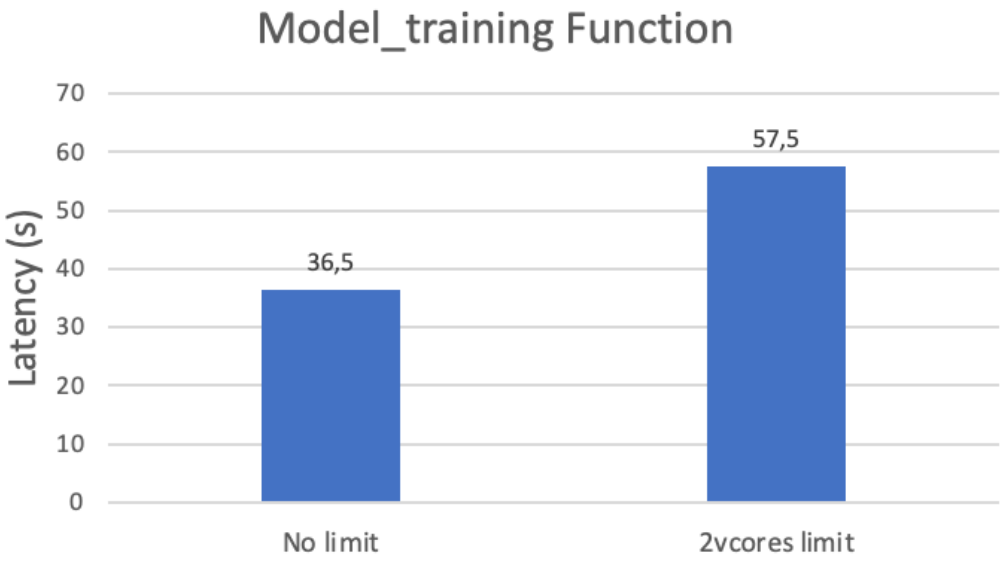

Figure 27. Model_training function without and with CPU usage limit

Figure 28 shows the performance for the Model_serving function when it is running alone and when it is co-located with the float, Model_training and Video_processing functions. The average latency of the Model_serving function is 33.5 seconds. However, the latency increment of this function when co-located is very small (float 0.3%, Model_training 13% and 0%, Video_processing). The highest increment is obtained when it is co-allocated with the Model_training function 5.4 seconds. This function has the same behavior

as the Model_training function when it is co-allocated with another function, it increments the number of cores usage.

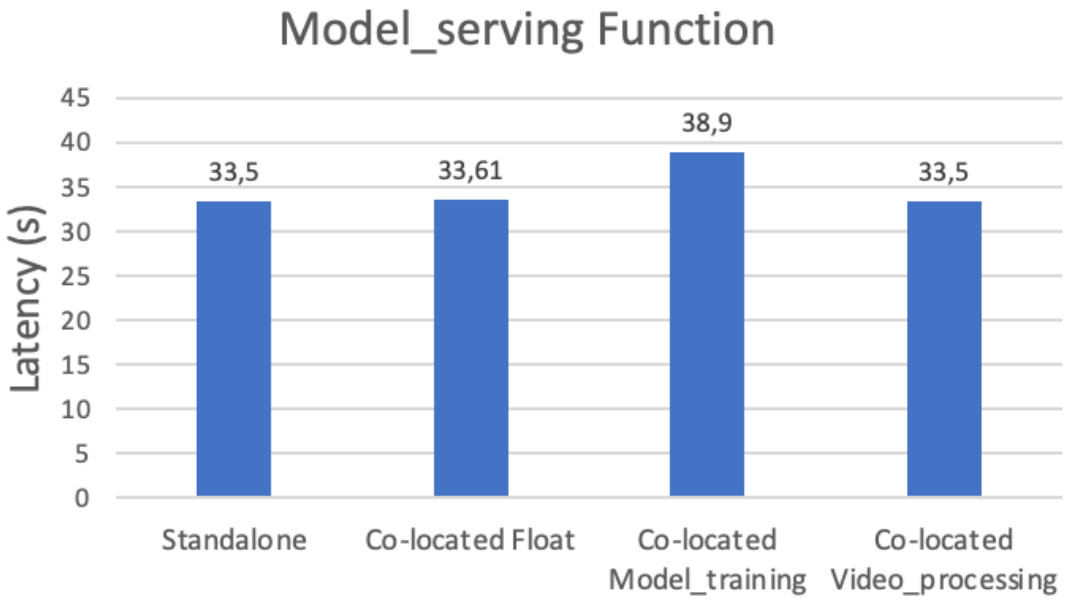

Figure 28. Model_serving function co-location.

This evaluation shows how interferences are detected taking into consideration the standalone execution of the function in the cluster and comparing the usage of the resources when the function is co-located with other functions in the same node of the cluster.

6.3 Profiling Process and Annotations from the Performance Pipeline Outputs

In this section the background details of the Profiling Process for characterization of deployed functions are portrayed. The aim of profiling is to be applied in the context of the Performance Pipeline, presented in WP3 and D3.2, following which a function will be annotated with relation to its usage of underlying resources. The goal is to categorize functions with respect to the usage of the resources in order to provide information to the co-location service described in this chapter. Thus the latter will not schedule functions that require the same type of resource on the same node, avoiding severe bottlenecks and making informed decisions for the placement of functions on the available nodes.

In order to design the profiling process, it was considered that this should be performed **relatively** to the other available functions. Thus the platform, at the WP4 level, can exploit a kind of function profiles **crowdsourcing** from all the available executions of the performance pipeline. This would remove the need to define arbitrary boundaries on what is considered to be a low/medium/high behaviour in terms of e.g. CPU usage, while these limits will be determined based on the available and observed traces.

While the function executes in the context of the performance pipeline, the averages for pod CPU, memory, file system and network data are received. An example trace appears in the following figure and is available through the PEF APIs described in D4.2.

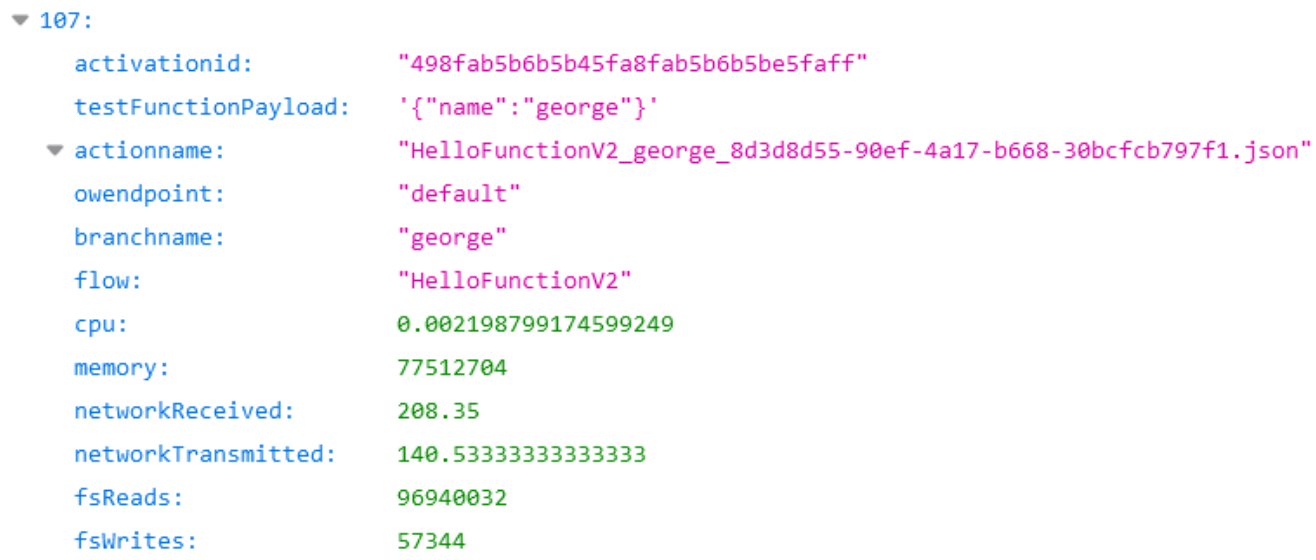

Figure 29. Example trace

Given that the function categorization needs to be performed in a relative manner, and based on the available function traces, we have selected an unsupervised learning algorithm for determining the profile boundaries. The process is based on an implementation of the well known "k-means" algorithm, in which k centroids of areas are determined from the available data. A k value of 3 has been selected, indicating three categories of resource usage: **low**, **medium** and **high**. Although this process can be extended with a silhouette analysis in order to determine the optimal value of k [56], in our case this is not considered practical for two reasons. Initially, the dataset on which the clusters are going to be applied is expected to grow with time, as more functions get registered and executed in the platform. Furthermore, the categories and according annotations are used by the underlying infrastructure components in order to perform the function scheduling. This implies an agreement on the specification of the potential annotations between the performance pipeline and the lower management layers, such as the co-location strategies. Hence, if from time to time the k value changes based on the silhouette analysis, this implies that this specification would be broken and further adaptations would need to be performed across the platform.

The clustering process is performed individually for each resource metric. This means that there is a separate categorization for each resource metric (CPU usage, function memory usage, file system usage, network data sent and received). This enables us to have more detailed annotations and potentially map them to the capabilities of the nodes. The determination of the resource usage clusters (k-means re-training) is performed in a periodic manner, set as a Kubernetes CronJob in the supporting framework while utilizing a respective clustering function registered in Openwhisk. More details on the according function implementation are provided in D4.2, as part of the Performance Evaluation Framework (PEF).

Once the newly analyzed function has been executed through the performance pipeline, we need to determine the clustering category to which it belongs for each resource metric. For this categorization, a relevant classifier function is used. This is based on a euclidean distance calculation for each metric in the function trace against the 3 available cluster centroids for that metric. The lowest distance from a centroid, compared to the other 2, means that this function is closer to this category than the others for this metric.

An overview of the approach appears in Figure 30.

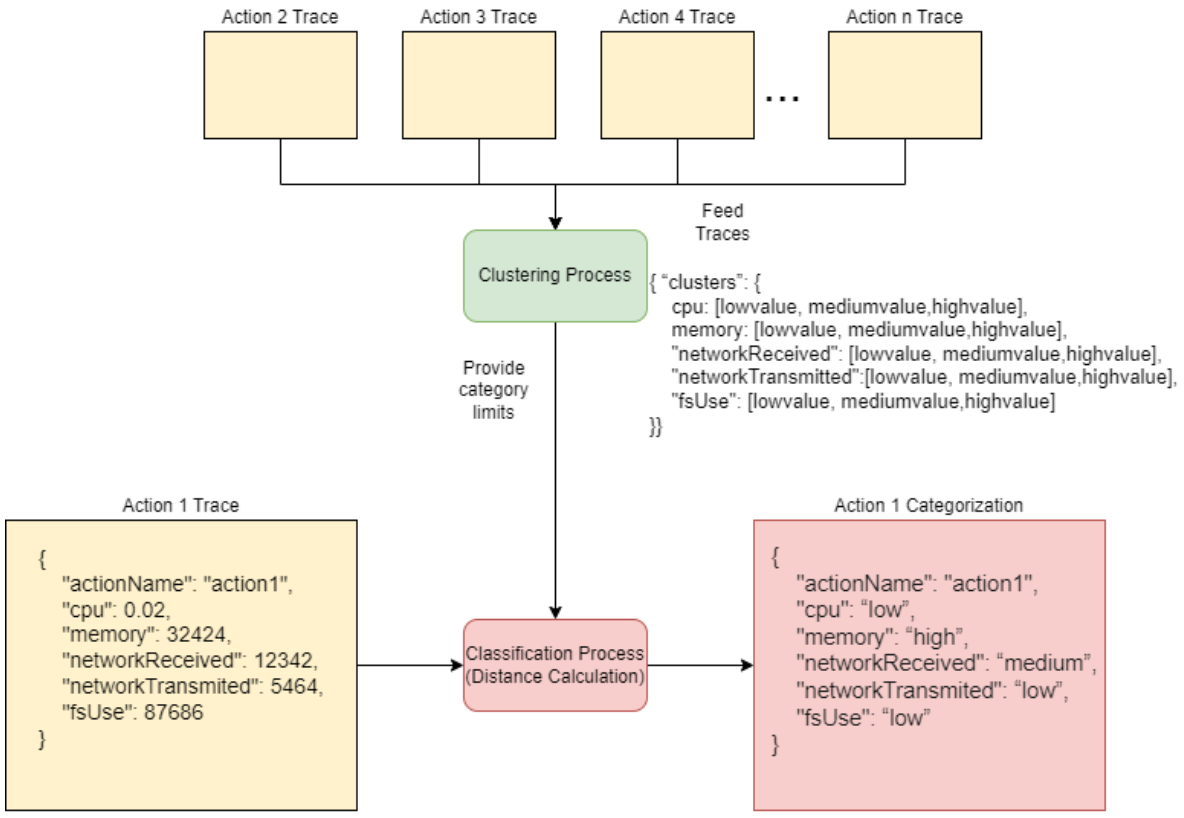

Figure 30. Overview of the Profiling and Classification Process for Function Characterization

6.3.a Experiment for the identification of function profiles

To evaluate the execution and outcome of the profiling and categorization process in terms of identifying and reducing bottlenecks from similar behavior, we selected four functions with varying computational requirements, implemented in native Python. These include sorting and Fibonacci functions, a large list generation function and a file read and write function for I/O-bound operations. Each of these functions accepts an integer as an input parameter, which determines the n-th number in the Fibonacci sequence, the length of the list to be sorted or generated, and the size of the random data read or written to the file (in bytes). Default values used for n were 30 for Fibonacci, 1 million for List, 1048576 bytes for fileRW and 10.000 for the sorting function.

Following their execution through the Performance Pipeline described in D3.2, the resource usage footprints of the pods executing each function in its main run were collected and appear in Figures 31, 32, 33, 34 and 35 for different resource types.

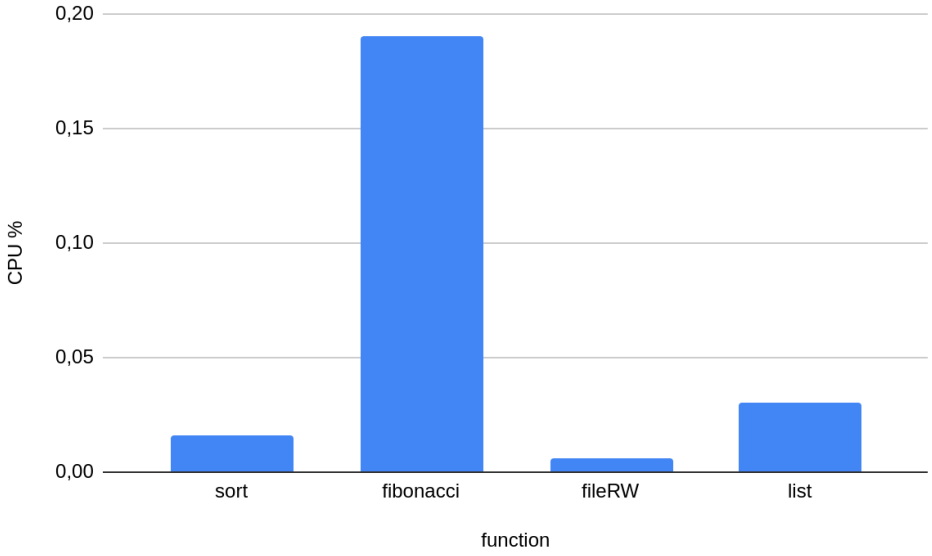

Figure 31. CPU Usage per Function

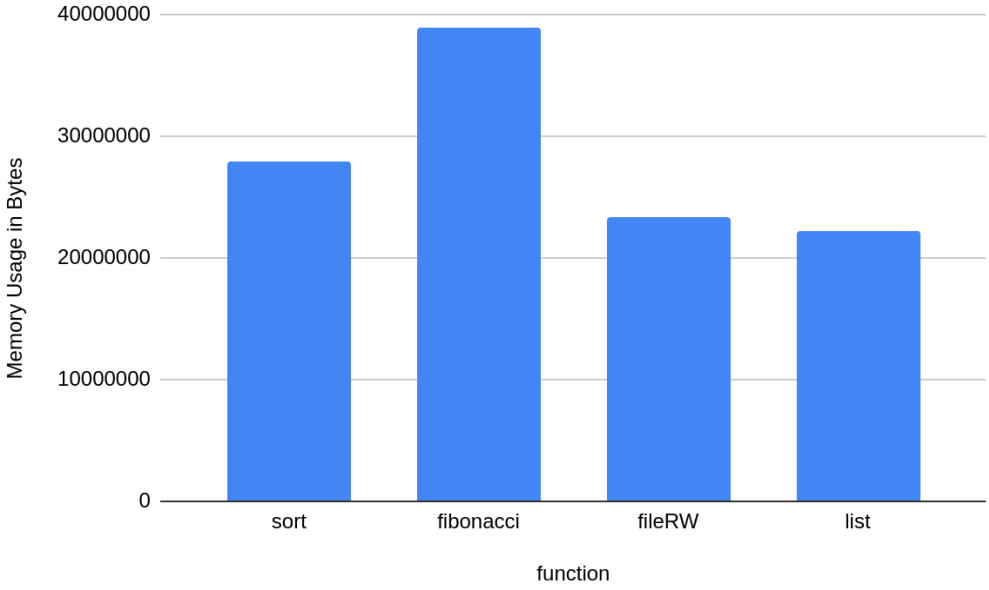

Figure 32. Memory Usage per Function

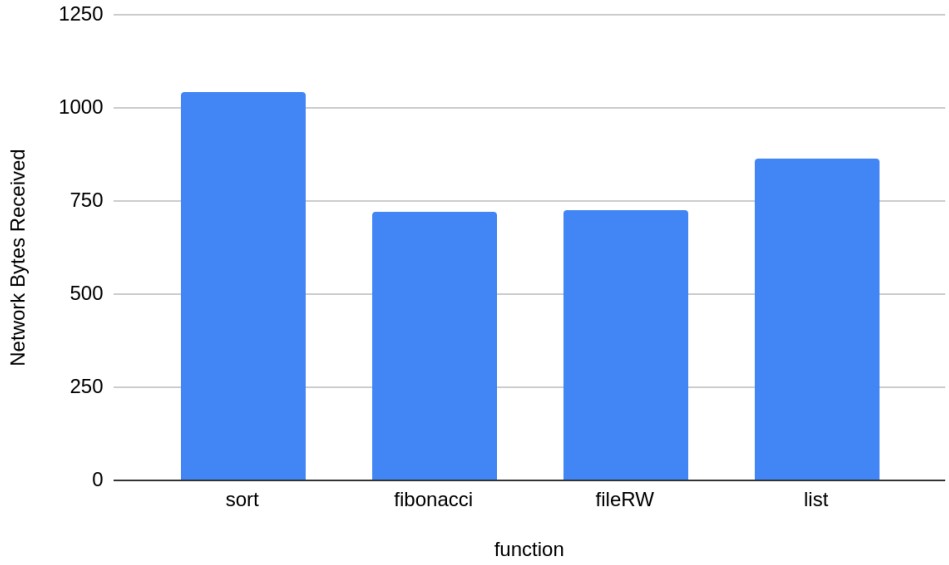

Figure 33. Network Received per Function

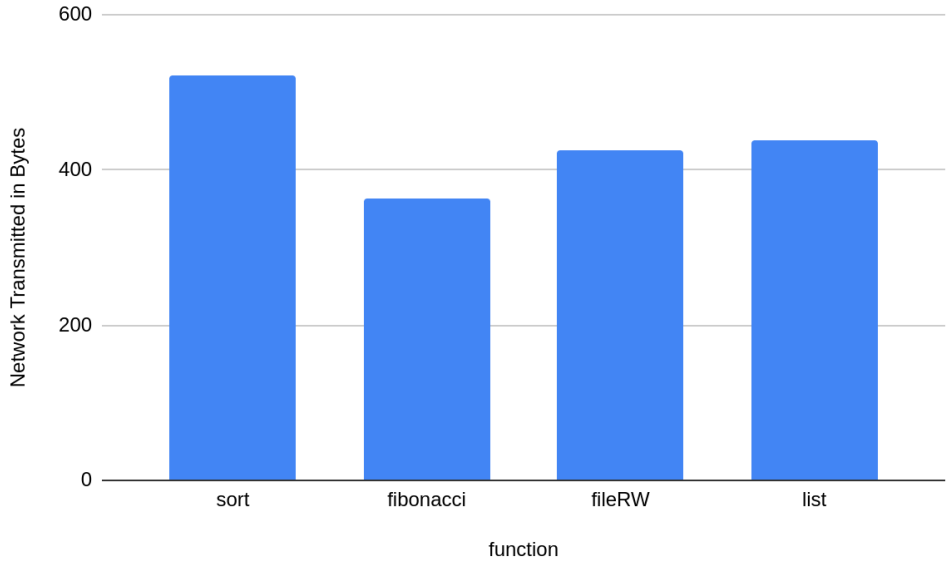

Figure 34. Network Transmitted per Function

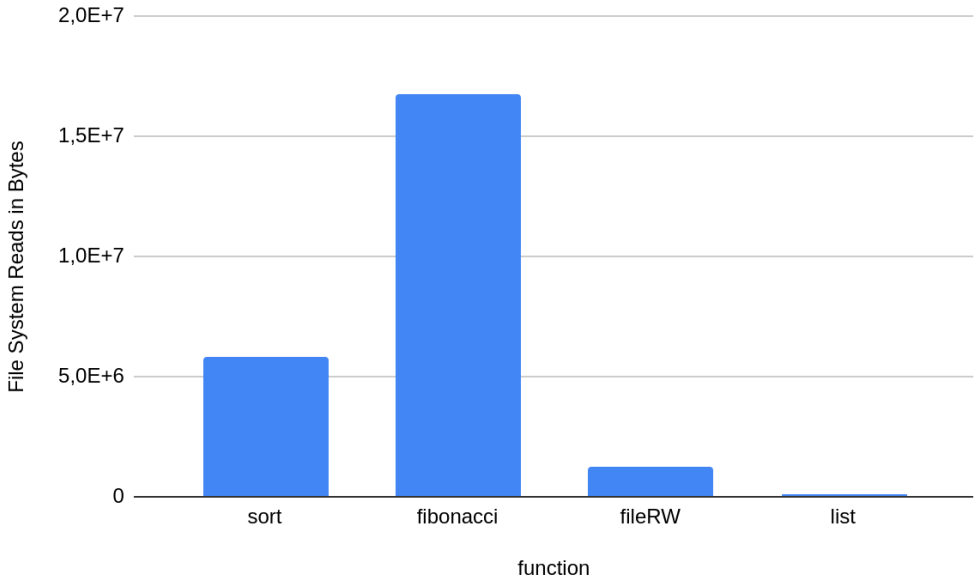

Figure 35. File System Use per Function

The related results of the clustering process and the according centroids for each resource usage category appear in Table 3, while the function classification per utilized resource intensity are presented in Table 4. From these we can identify which one of the functions is more resource intensive for each type of resource. In this case some profiles seem to be misleading since for example the fileRW case has lower filesystem use than the fibonacci case. This is due to the small input argument used for its function execution. In the case where more indicative executions are available for a given function (i.e. with smaller or larger arguments), the mechanism can acquire the average values of all these runs to get a better insight. But this is also indicative of misconceptions that might exist for the computational nature of a function. For example, if the developer was asked to annotate by hand these functions, they would annotate the fileRW case as disk intensive, whereas it appears from the acquired profiles that in fact other function types that would not appear as such use more the specific resource. Thus the usage of the presented mechanism can help avoid such stereotyping and base our decisions on the acquired data and compared relative usage of a resource.

Resource	Low	Medium	High
CPU	0.00580	0.02965	0.18521
Memory (Bytes)	22821091	27701589	47930026
Network (Bytes) Received	173.33377	663.13804	983.72398
Network (Bytes) Transmitted	196.98828	359.94572	495.84016
File System (Byte) Reads	1494016	12274688	49229824

Table 3. Resource Usage Category Centroids Determined from the Clustering Process

Resource	sort (n=10000)	fibonacci (n=30)	fileRW (n=1 MB)	list (n=1 million)
CPU	low	high	low	medium
Memory	medium	high	low	low
Network Received	medium	medium	medium	high
Network Transmitted	medium	medium	medium	high
File System Reads	low	medium	low	low

Table 4. Function Category Classification per Resource Type for Default Inputs of the Test Functions

6.3.b Function Input Dependencies on Execution Time

One question during the performance pipeline testing was how much a given function's input can actually affect the resulting categorization of that function. Given that the input may determine how many computations are performed or how much memory is needed, it is necessary to investigate that aspect. This is especially true for the tested functions in our case. For this reason a number of tests were performed, altering the default values of n applied in the test functions. The results are presented in Figure 36. From these it is evident that it's possible for a function profile to shift between categories for different input values. In all examined cases specific values of n caused this category shift. On the other hand, there are types of functions (e.g. model inference ones such as the one in [57]) in which the input applied does not actually affect the resources used. In that case the input vector, regardless of the contents, is applied to a neural network in order to get a predicted output. The amount of computation applied is the same regardless of the actual input vector in that case.

The main conclusion from the analysis in this section is the fact that there are functions that are severely influenced by the type or size of inputs that are applied. Through the defined process, the developers can investigate whether this applies to their given function. This also implies that the framework may need to categorize not only by the function name (thus one profile for each function), but a combination of the function name and the input applied.

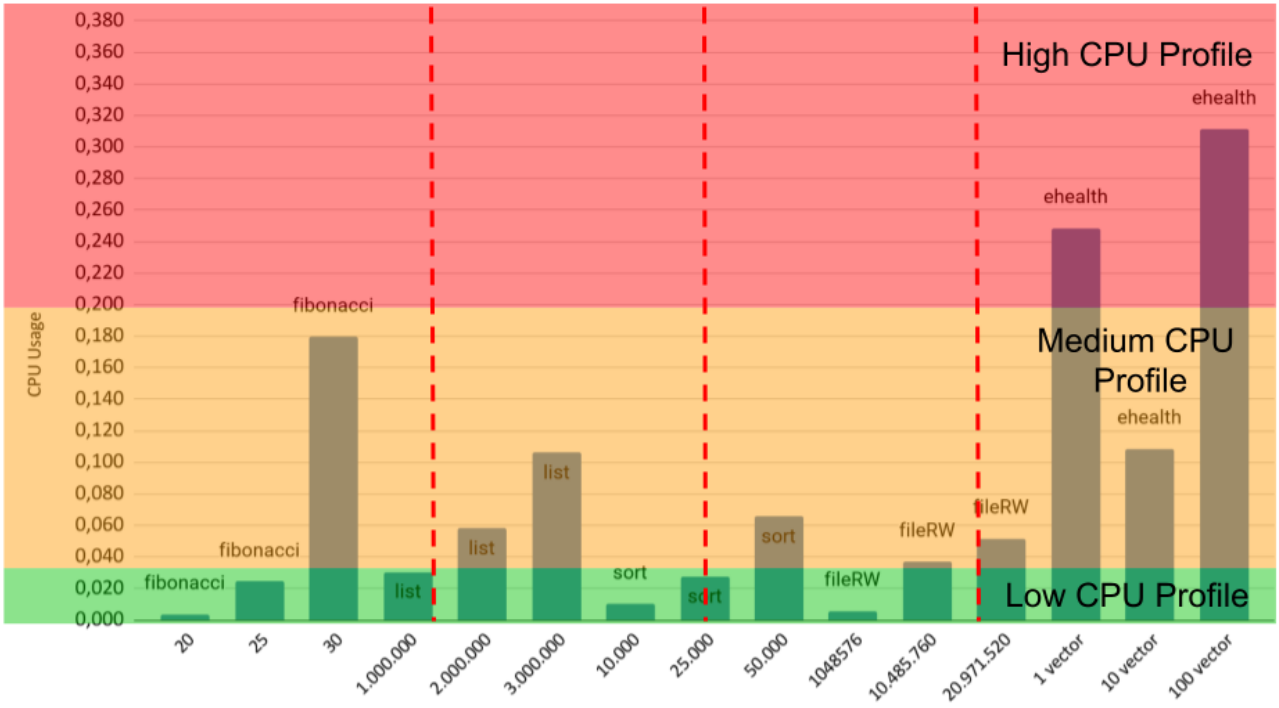

Figure 36. CPU usage and according classification for each function with different inputs

6.3.c Co-location experiments and performance degradation based on profiles

In order to check the effect of the categorization and validate its usage, the following experiment was designed and implemented. The execution of the functions was limited to a node with 4 cores and 16 GB of RAM and the aim was to measure the concurrency overhead of different combinations (e.g. low-low, low-medium, low-high etc) of characterized functions. In each scenario, the node would execute around 16 container functions, in which half should be from each category. The goal is to demonstrate that with the use of the annotations, the provider may reduce the observed concurrency overhead.

A blocking thread client based on JMeter was developed in order to ensure that at any given time, only the defined number of concurrent requests would be active towards the FaaS system from each category defined in a scenario. The JMeter client appears in the following Figure. It has been parametrically designed so that the configuration can be performed for each test series through user defined variables that are propagated to the relevant fields in the workload structure. Through this we are able to easily change configurations and adapt the designed load.

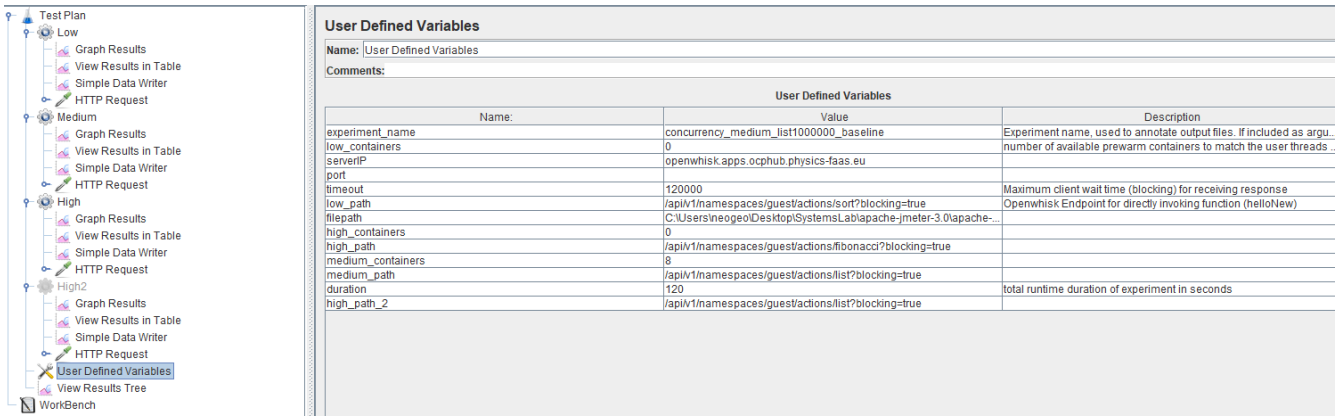

Figure 37. JMeter client snapshot

The measured scenarios included:

- Scenario 1 (High-Medium): 8 threads targeting a high CPU function like fibonacci (with n=30) and 8 threads targeting a medium CPU profile function (list with n=1000000)
- Scenario 2 (High1-High1): 16 threads targeting a high CPU function like fibonacci (with n=30). In this case we consider that this should be the worst case, since it is a high-high coallocation scenario in which the 16 functions are of the same type thus exemplify the largest possible competition for similar resources
- Scenario 3 (High1-High2): 8 threads targeting a high CPU function like fibonacci (with n=30) and 8 threads targeting another high CPU function (but of different type) like List with n=3000000
- Scenario 4 (High1-Low1): 8 threads targeting a high CPU function like fibonacci (with n=30) and 8 threads targeting a low CPU profile function (sort with n=10000)
- Scenario 5 (High1-Low2): 8 threads targeting a high CPU function like fibonacci (with n=30) and 8 threads target- ing a low CPU profile function (fileRW with n=1048576 bytes)
- Scenario 6 (Low1-Low2): 8 threads targeting a low CPU function (sort with n=10000) and 8 threads targeting an- other low CPU profile function (fileRW with n=1048576 bytes)
- Scenario 7 (Medium-Low1): 8 threads targeting a medium CPU function (list with n=1000000) and 8 threads targeting a low CPU profile function (fileRW with n=1048576 bytes)
- Scenario 8 (Medium-Low2): 8 threads targeting a medium CPU function (list with n=1000000) and 8 threads targeting a low CPU profile function (sort with n=10000)

For all used tests, the baseline times were extracted from 8 threads of the load running as standalone (without any other function type executing on the node). The runs were set to last for 2 minutes in each case.

The results appear in Figures 38, 39, 40 and 41 and relate to the response time from the client. Initially we plot the response time of the function that was categorized as high (Fibonacci with n=30), in the various scenarios where it participates (38). From this it is evident that while its baseline time is around 2036 milliseconds, its coallocation with a low function incurs a degradation of around 15% (2329 for the case of FileRW and 2387 for Sort). Going to a medium-high combination raises the degradation to around 39.68% (or 2844 milliseconds of runtime). The largest degradation appears for the High1-High2 case (3387 milliseconds of response time or 66.3%) and finally for the High1-High1 case (3865 milliseconds or 89.83%).

For the low function cases, fileRW times appear in Figure 39. In this case, the baseline time is 101 milliseconds, while its co-location with another low function (sort 10000) causes it to reach 165 milliseconds, thus increasing by 63.36%. Assigning it with a medium case has a mediocre effect, since it reaches 171 milliseconds or 70.29%, very similar to the low-low case. However, collocating it with a high function like Fibonacci causes the delay to skyrocket to 659 milliseconds or 552% of degradation.

Thus although this combination is beneficial for the high function, as seen in the according Figure 38, it takes a heavy toll on the low function.

A similar behavior appears in Figure 40 for the medium case (List with n=1 million), that portrays a 43% degradation when collocated with the low fileRW and a 24% one when collocated with the low sort. However when the high function is used, the according degradation reaches 467%. In Figure 41 the low sort case is portrayed. In this we observe similar behaviors, with the only exception being that the medium-low case has a slightly better performance on the low sort than the low-low combination.

Thus it is evident that the produced categorization from the proposed mechanism can inform the provider on potential overheads and enable them to apply optimizations of co- allocation in order to minimize that effect. Through this they can achieve both competing goals, i.e. further utilize their infrastructures while reducing the deterioration of the user experienced QoS. Another approach would be to offer Quality of Service classes to customers. It is evident in some combinations (e.g. High-Low) that while the combination is good for the High case (producing the least overhead), it is not beneficial for the low case. Thus low characterized functions that belong to a cheaper QoS could be scheduled with High functions, whereas more expensive offerings may prevent these low categorized functions from participating in such combinations.

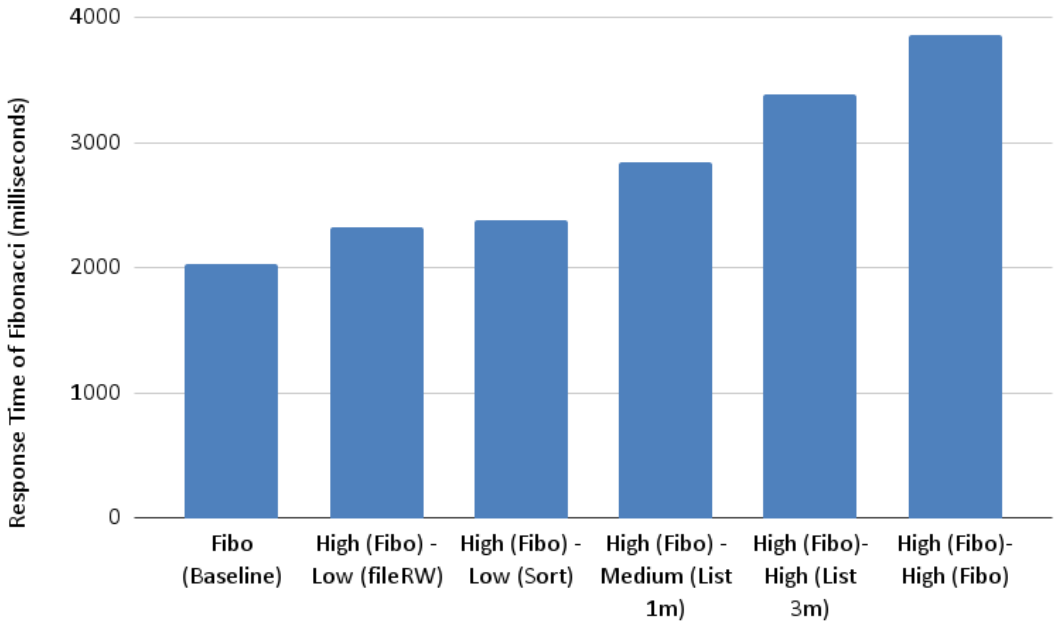

Figure 38. Response time of Fibonacci function when paired with functions of different category

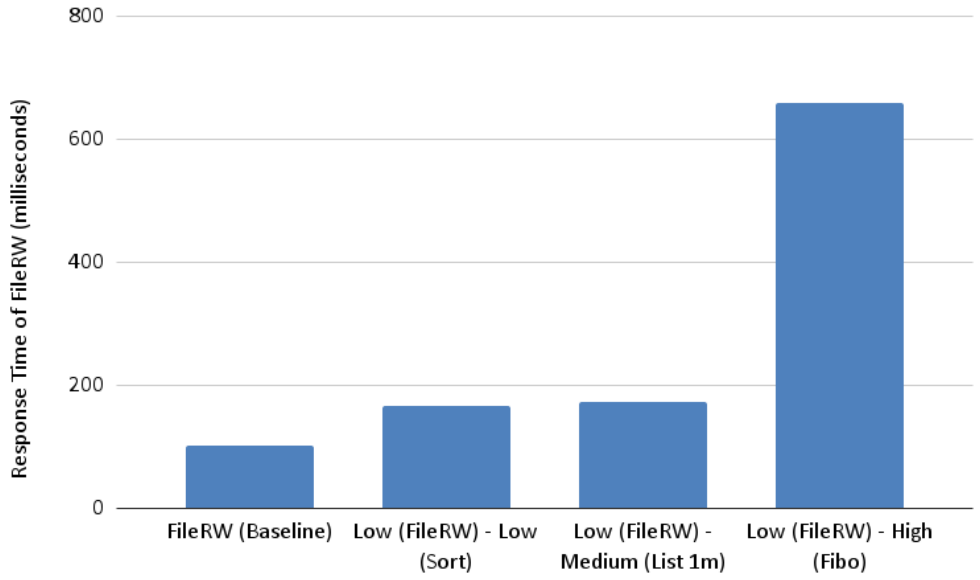

Figure 39. Response time of fileRW function when paired with functions of different category

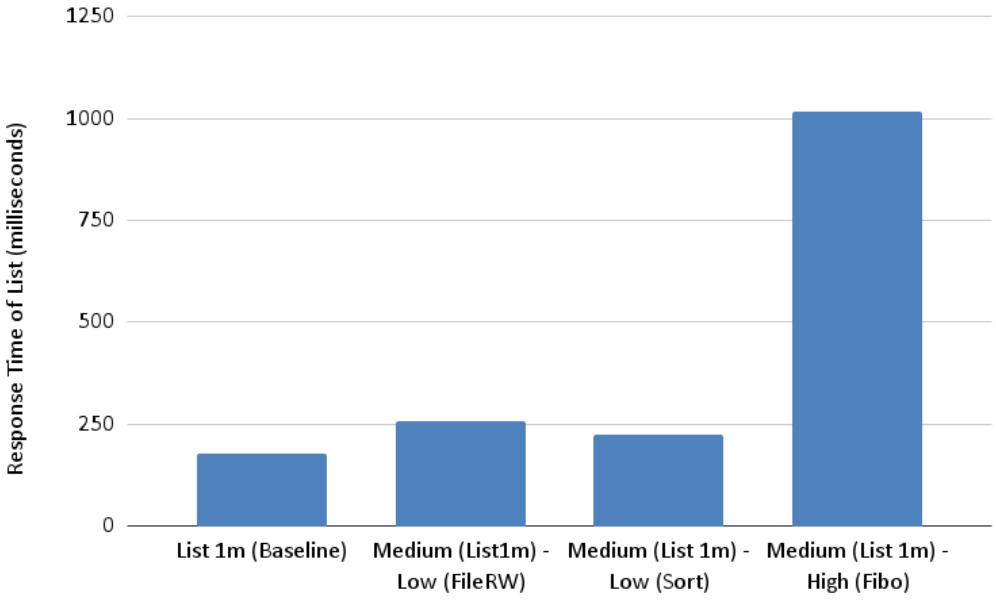

Figure 40. Response time of list function when paired with functions of different category

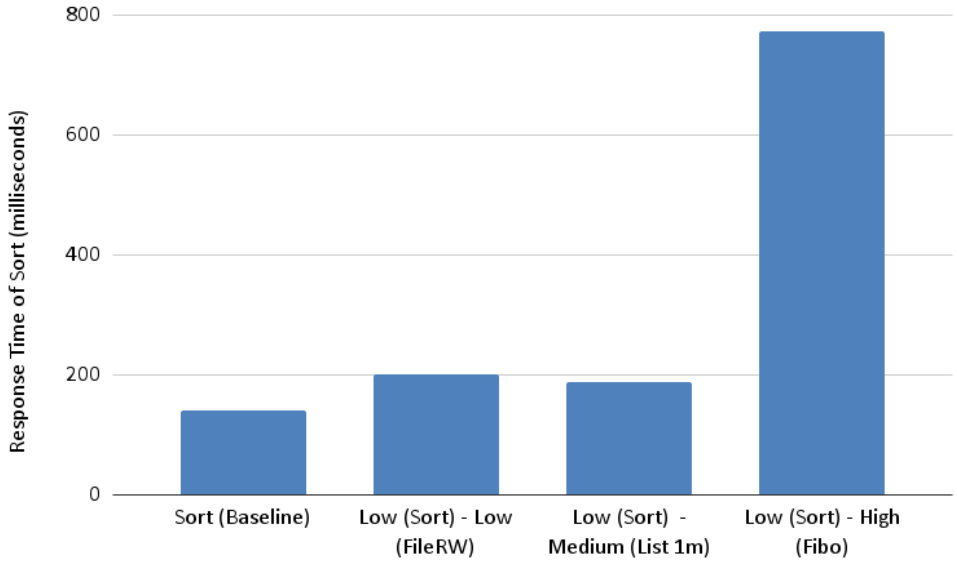

Figure 41. Response time of sort function when paired with functions of different category

7. Conclusions

This deliverable presents the final version of the PHYSICS Extended Infrastructure Services. The main building blocks are described, together with the design decisions taken, its implementation and interaction details, initial results. Some next steps that will happen during the last month of the project and even after it finishes (such as upstream work) are also described.

This deliverable describes:

- the upstream components selected as main building blocks for the infrastructure layer, including multicluster setups.
- the new APIs created at the infrastructure layer that the upper layers (WP3 and WP4) components leveraged.
- the components (i.e. operators) behind those new APIs, providing the PHYSICS functionality
- the other components that improve the performance at the infrastructure layer (scheduler and co-location)
- the semantic component that provides needed information to WP4 to operate.

In addition the interactions between the components of the infrastructure layers, as well as with other WP's components are detailed.

The work presented in this document demonstrates the cooperation and synchronization with upstream communities, towards enhancing specific projects. It also highlights that the project has followed the best upstream Kubernetes practices on the developed components (e.g. by using Webhooks, CRDs and Operators).

Before submitting a Deliverable, please check out the Deliverable Checklist.

List of MUST-DO	Check
Carefully compile the Table & Deliverable Title at page 1	☐
Modify the footer of the document	☐
Fill in Contributing Partners Table at page 2	☐
Fill in Revision History Table at page 2	☐
Compile List of Abbreviations at page 3 (check twice!)	☐
Carefully re-read the Executive Summary at page 4	☐
Update the Table of Contents at page 5	☐
Check again the overall document to ensure quality of contents	☐
Check again the overall document to ensure guidelines are respected	☐
Check all the Tables, Figures and Codes to verify captions	☐
Carefully re-read the Conclusions at the end of the document	☐
Check the References referring to IEEE style	☐
Save the document following the Naming Convention of PHYSICS	☐
Send both .docx and .pdf files to the Coordinator in advance	☐

References

[1] OpenShift documentation. (Accessed: 2023, Sep. 27). [Online]. Available: https://docs.openshift.com/

[2] https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

[3] Operators development. (Accessed: 2023, Sep. 27). [Online]. Available: https://Kubernetes.io/docs/concepts/extend-Kubernetes/operator/

[4] Open Cluster Management. (Accessed: 2023, Sep. 27). [Online]. Available: https://open-cluster-management.io/

[5] Submariner. (Accessed: 2023, Sep. 27). [Online]. Available: https://submariner.io/

[6] https://kubernetes.io/

[7] Prometheus. (Accessed: 2023, Sep. 27). [Online]. Available: https://prometheus.io/

[8] Open Whisk. (Accessed: 20223 sep. 27). [Online]. Available: https://openwhisk.apache.org/

[9] Knative. (Accessed: 20223 sep. 27). [Online]. Available: https://knative.dev/docs/

[10] Kepler. (Accessed: 20223 sep. 27). [Online]. Available: https://sustainable-computing.io/

[11] Cloud Native Computing Foundation. (Accessed: 2023, Sep. 27). [Online]. Available: https://www.cncf.io/

[12] Skupper. (Accessed: 20223 sep. 27). [Online]. Available:https://skupper.io/

[13] Microshift. (Accessed: 2023, Sep. 27). [Online]. Available: https://next.redhat.com/project/microshift/

[14] KinD. (Accessed: 2023, Sep. 27). [Online]. Available: https://kind.sigs.k8s.io/

[15] Submariner network plugin syncer issue. (Accessed: 2022, Jan. 27). [Online]. Available: https://github.com/submariner-io/submariner/issues/1631

[16] Submariner broken service connectivity with ovn-Kubernetes. (Accessed: 2022, Jan. 27). [Online]. Available: https://github.com/submariner-io/submariner/issues/1608

[17] Broken OVNKubernetes to after deploying submariner. (Accessed: 2022, Jan. 27). [Online]. Available: https://github.com/submariner-io/submariner/issues/1625

[18] HPA. (Accessed: 2023, Sep. 27). [Online]. Available: https://Kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[19] VPA. (Accessed: 2023, Sep. 27). [Online]. Available: https://github.com/Kubernetes/autoscaler/tree/master/vertical-pod-autoscaler#intro

[20] Resource Limit. (Accessed: 2023, Sep. 27). [Online]. Available: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

[21] KEDA. (Accessed: 2023, Sep. 27). [Online]. Available: https://keda.sh/

[22] K8s Custom resource definitions. (Accessed: 2023, Sep. 27). [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

[23] ETCD Database (Accessed: 2023, Sep. 27). [Online]. Available:

[24] Grid 5000. (Accessed: 2022, Jan. 27). [Online]. Available: https://www.grid5000.fr/w/Grid5000:Home

[25] Configuring multiple schedulers. (Accessed: 2023, Sep. 27). [Online]. Available: https://Kubernetes.io/docs/tasks/extend-Kubernetes/configure-multiple-schedulers/

[26] Extensible controllers. (Accessed: 2023, Sep. 27). [Online]. Available: https://Kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/

[27] Poulakis, Y., Fatouros, G., Kousiouris, G., Kyriazis, D. (2023). HOCC: An Ontology for Holistic Description of Cluster Settings. In: Economics of Grids, Clouds, Systems, and Services. GECON 2022. Lecture Notes in Computer Science, vol 13430. Springer, Cham. https://doi.org/10.1007/978-3-031-29315-3_4

[28] Tosca. (Accessed: 2022, Jan. 27). [Online]. Available: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca#overview

[29] Camel. (Accessed: 2022, Jan. 27). [Online]. Available: http://camel-dsl.org/

[30] Owl. (Accessed: 2022, Jan. 27). [Online]. Available: https://www.w3.org/OWL/

[31] Musen, M.A. The Protégé project: A look back and a look forward. AI Matters. Association of Computing Machinery Specific Interest Group in Artificial Intelligence, 1(4), June 2015. DOI: 10.1145/2557001.25757003.

[32] Flask. (Accessed: 2022, Jan. 27). [Online]. Available: https://flask.palletsprojects.com/en/2.0.x/

[33] Owl ready. (Accessed: 2022, Jan. 27). [Online]. Available: https://owlready2.readthedocs.io/en/v0.35/

[34] Ontospy. (Accessed: 2022, Jan. 27). [Online]. Available: https://github.com/lambdamusic/Ontospy

[35] Hasan, M.M., Kousiouris, G., Anagnostopoulos, D., Stamati, T., Loucopoulos, P., Nikolaidou, M.: CISMET: A Semantic Ontology Framework for Regulatory-Requirements-Compliant Information Systems development and Its Applicationin the GDPR Case. International Journal on Semantic Web and Information Systems(IJSWIS) 17(1), 1–24 (January 2021)

[36] PHYSICS Cluster onboarding public repo: https://github.com/luis5tb/physics-cluster-registration

[37] OCM architecture. (Accessed: 2023, Sep. 27). [Online]. Available: https://open-cluster-management.io/concepts/architecture/

[38] Cloud Events. (Accessed: 2023, Sep. 27). [Online]. Available: https://cloudevents.io/

[39] Red Hat Device Edge. (Accessed: 2023, Sep. 27). [Online]. Available: https://cloud.redhat.com/blog/introducing-the-new-red-hat-device-edge

[40] Extending Kubernetes API. (Accessed: 2023, Sep. 27). [Online]. Available: https://Kubernetes.io/docs/concepts/extend-Kubernetes/api-extension/custom-resources/

[41] Machine Custom resource definitions. (Accessed: 2023, Sep. 27). [Online]. Available: https://github.com/Kubernetes-sigs/cluster-api/blob/main/config/crd/bases/cluster.x-k8s.io_machines.yaml

[42] Affinity and anti affinity. (Accessed: 2022, Jan. 27). [Online]. Available: https://Kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#affinity-and-anti-affinity

[43] DevConf Workshop. (Accessed: 2023, Sep. 27). [Online]. Available: https://research.redhat.com/blog/2023/06/09/red-hat-research-engineers-will-lead-the-workshop-on-k8s-operator-for-faas-at-devconf-cz-2023

[44] Luc Angelelli, Anderson Andrei Da Silva, Yiannis Georgiou, Michael Mercier, Grégory Mounié, Denis Trystram: Towards a Multi-objective Scheduling Policy for Serverless-based Edge-Cloud Continuum. CCGrid 2023: 485-497

[45] Knative Sequences. (Accessed: 2023, Sep. 27). [Online]. Available: https://knative.dev/docs/eventing/flows/sequence/

[46] CODECO EU Project. (Accessed: 2023, Sep. 27). [Online]. Available: https://he-codeco.eu/

[47] Batsim. (Accessed: 2022, Jan. 27). [Online]. Available: https://batsim.readthedocs.io/en/latest/

[48] Serverless faas workbench. (Accessed: 2022, Jan. 27). [Online]. Available: https://github.com/kmu-bigdata/serverless-faas-workbench

[49] OpenWhisk serverless faas. (Accessed: 2022, Jan. 27). [Online]. Available: https://gitlab.com/andersonandrei/serverless-faas-workbench/-/tree/openwhisk/openwhisk

[50] Scheduling functions on serverless computing. (Accessed: 2022, Jan. 27). [Online]. Available: https://gitlab.com/andersonandrei/scheduling-functions-on-serverless-computing/-/tree/main/dev/openwhisk_env

[51] Casanova, H., Giersch, A., Legrand, A., Quinson, M., Suter, F.: Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms. Journal of Parallel and Distributed Computing 74(10), 2899–2917 (Jun 2014)

[52] Pierre-François Dutot, Michael Mercier, Millian Poquet, Olivier Richard: Batsim: A Realistic Language-Independent Resources and Jobs Management Systems Simulator. JSSPP 2016: 178-197

[53] Assign pods to nodes. (Accessed: 2022, Jan. 27). [Online]. Available: https://Kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/

[54] Workbench. (Accessed: 2023, Sep. 27). [Online]. Available: https://github.com/ddps-lab/serverless-faas-workbench

[55] MinIO. (Accessed: 2023, Sep. 27). [Online]. Available: https://min.io/

[56] F. Wang, H.-H. Franco-Penya, J. D. Kelleher, J. Pugh, and R. Ross, "An analysis of the application of simplified silhouette to the evaluation of k-means clustering validity," in Machine Learning and Data Mining in Pattern Recognition: 13th International Conference, MLDM 2017, New York, NY, USA, July 15-20, 2017, Proceedings 13. Springer, 2017, pp. 291–305

[57] G. Kousiouris, "A self-adaptive batch request aggregation pattern for improving resource management, response time and costs in microservice and serverless environments," in 40th IEEE International Performance Computing and Communications Conference (IPCCC 2021). IEEE, 2021.

Disclaimer

Copyright Message