PHYSICS

oPTIMIZED HYBRID SPACE-TIME SERVICE CONTINUUM IN FAAS

D6.1 – PROTOTYPE OF THE INTEGRATED PHYSICS SOLUTION FRAMEWORK AND RAMP V1

Lead Beneficiary	HPE
Work Package Ref.	WP6 – Use Cases Adaptation, Experimentation, Evaluation
Task Ref.	T6.1 – Solution Services Integration and Reusable Artefacts Marketplace Platform (RAMP) Creation
Deliverable Title	D6.1 – Prototype of the Integrated PHYSICS solution framework and RAMP V1
Due Date	2022-03-31
Delivered Date	2022-03-31
Revision Number	3.0
Dissemination Level	Public (PU)
Type	Demonstrator (DEM)
Document Status	Release
Review Status	Internally Reviewed and Quality Assurance Reviewed
Document Acceptance	WP Leader Accepted and Coordinator Accepted
EC Project Officer	Mr. Stefano Foglietta

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

CONTRIBUTING PARTNERS

Partner Acronym	Role[1]	Name Surname[2]
HPE	Lead Beneficiary	Alessandro Mamelli, Domenico Costantino
UPM	Contributor	
HUA	Contributor	
GFT	Contributor	
RHT	Contributor, Internal Reviewer	Josh Salomon, Luis Tomas Bolivar
INNOV	Contributor	
BYTE	Contributor	
RYAX	Contributor	
ATOS	Contributor, Internal Reviewer	Antonio Castillo Nieto, Carlos Sánchez Fernández
FTDS	Quality Assurance	

REVISION HISTORY

Version	Date	Partner(s)	Description
0.1	2022-01-10	HPE	ToC Version and preliminary contents
0.2	2022-01-21	HPE	Refined ToC Version and additional preliminary contents
1.0	2022-02-28	HPE, all Contributors	1st integrated version
1.1	2022-03-17	HPE, all Contributors	2nd integrated version
1.2	2022-03-20	HPE	Version for Peer Reviews
1.3	2022-03-24	ATOS-RHT	Peer Reviews completed
2.0	2022-03-28	HPE	Version for Quality Assurance
2.1	2022-03-29	FTDS	Quality Assurance completed
3.0	2022-03-30	HPE	Version for Submission

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

LIST OF ABBREVIATIONS

2FA	Two-Factor Authentication
ACM	RedHat Advanced Cluster Management
API	Application Programming Interface
AWS	Amazon Web Services
CI/CD	Continuous Integration / Continuous Delivery
CPU	Central Processing Unit
CRD	Custom Resource Definition
CRI	Container Runtime Interface
CVE	Common Vulnerabilities and Exposures
DevOps	Development Operations
DL	Deep Learning
DMS	Distributed Memory Service
DNN	Deep Neural Net
DoS	Denial of Service
DRAM	Dynamic Random Access Memory
DSL	Domain Specific Languages
EKS	Amazon Elastic Container Service for Kubernetes
FaaS	Function as a Service
HPA	Horizontal Pod Autoscaler
HSM	Hardware Security Module
HTTP/S	HyperText Transfer Protocol / Secure
I/O	Input/Output
IaC	Infrastructure as Code
IAM	Identity and Access Management
IoT	Internet of Things
JSON	JavaScript Object Notation
JSON-LD	JSON for Linking Data
JWT	JSON Web Token
KMS	Key Management Service
LDAP	Lightweight Directory Access Protocol
MARLA	MApReduce on Lambda
MCSC	Multi-Cloud Service Composition
MILP	Mixed-Integer Linear Programming
ML	Machine Learning
MVP	Minimum Viable Platform
NVMe	Non-Volatile Memory Express
OCI	Open Container Initiative
OCM	Open Cluster Management
OIDC	Open ID Connect
OWASP	Open Web Application Security Project
QoS	Quality of Service
RA	Reference Architecture
RAMP	Reusable Artefacts MarketPlace
RDF	Resource Description Framework
REST	REpresentational State Transfer
RWX	Read Write Many
SAML	Security Assertion Markup Language
SDK	Software Development Kit

SFG	Serverless Function Generator
SGX	Software Guard Extensions
SLA	Service-Level Agreement
SOA	Service Oriented Architecture
SPT	Shortest Processing Time
SSH	Secure Shell protocol
SSL	Secure Sockets Layer
SSO	Single Sign-On
TLS	Transport Layer Security
UI	User Interface
UML	Universal Modelling Language
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
XML	Extensible Markup Language
XSS	Cross Site Scripting
XXE	XML External Entity
YAML	YAML Ain't Markup Language

EXECUTIVE SUMMARY

Within the scope of PHYSICS Work Package 6 (Use Cases Adaptation, Experimentation, Evaluation), this document describes the results of Task T6.1 (Solution Services Integration and Reusable Artefacts Marketplace Platform (RAMP) Creation) achieved during the first phase of the project, and provides the first version (D6.1) of this deliverable (out of the two iterations foreseen in the whole work plan for WP6).
With respect to the general WP6 objectives, the deliverable mainly focuses on two of them, i.e.:

- To integrate the various technical artefacts of the technical Work Packages (WP3-4-5) to the 3 logical bundles, enabling their use as one vertical solution or separate per case bundle
- To provide the finally available demonstrator executions demonstrating the effectiveness of the approach as well as the operational version of the RAMP marketplace, to be used in WP7 activities

The achieved results provide key contributions for the fulfilment of the 2nd major WP6 milestone (MS5 – PHYSICS 1st integrated platform release – foreseen for M15 of the project) and provide the first release of the proposed solution.
The document is the companion textual specification of the major result of the deliverable and the task: the first version of the prototype of the integrated PHYSICS solution framework and RAMP, which have been deployed into the PHYSICS blueprint reference target infrastructure. The document and the integrated PHYSICS solution framework and RAMP setup constitute the overall deliverable and task output.
As consistently done since the beginning of WP6 T6.1 activities, the work has been carried out in close cooperation and coordination with the other PHYSICS WP6 tasks and Work Packages 2-3-4-5 tasks and partners, taking into account and integrating the delivered results and concepts (e.g. the PHYSICS Reference Architecture proposed by WP2 and the solution framework major components and services artefacts proposed by WP3, WP4 and WP5) in a coherent and uniform manner.
Moreover, the outcomes of T6.1 will continue to feed the work of the remaining WP6 tasks (mainly the tasks dedicated to Use Cases adaptation, experimentation and evaluation) for the upcoming 1st iteration of the PHYSICS Pilots and Use Cases Operations and Stakeholders' Evaluation of the proposed solution framework.
Finally, the delivered integrated PHYSICS solution framework and RAMP marketplace will be fundamental inputs and drivers for the Work Package dedicated to Exploitation, Dissemination and Impact Creation, with special emphasis on the task related to Business Innovation Development & Exploitation.

CONTENTS

FIGURES

TABLES

1. INTRODUCTION

The PHYSICS project aims to enable European Cloud Service Providers (CSPs) to exploit the most modern, scalable and cost-effective cloud model (FaaS), operated across multiple service and hardware types, provider locations, edge, and multi-cloud resources. To this end, it applies a unified continuum approach, including functional and operational management across sites and service stacks, performance through the relativity of space (location of execution) and time (of execution), enhanced by semantics of application components and services. PHYSICS applies this scope via a vertical solution consisting of a:

- Cloud Design Environment, enabling design of visual workflows of applications, exploiting provided generalised cloud design patterns functionalities with existing application components, easily integrated and used with FaaS platforms, including incorporation of application-level control logic and adaptation to the FaaS model
- Optimised Platform Level FaaS Service, enabling CSPs to acquire a cross-site FaaS platform middleware including multi-constraint deployment optimization, runtime orchestration and reconfiguration capabilities, optimizing FaaS application placement and execution as well as state handling within functions, while cooperating with provider-local policies
- Backend Optimization Toolkit, enabling CSPs to enhance their baseline resources performance, tackling issues such as cold-start problems, multitenant interference and data locality through automated and multi-purpose techniques.

PHYSICS also delivers a Reusable Artefacts MarketPlace (RAMP), in which internal and external entities (developers, researchers etc.) will be able to contribute fine-grained, reusable and tested artefacts (functions, flows, controllers, etc.).
Furthermore, the project designs and implements a range of pilots and use cases that aim at validating these technologies in real-life scenarios of three vertical sectors (eHealth, Agriculture and Manufacturing).
Within PHYSICS, WP6 (Use Cases Adaptation, Experimentation, Evaluation) aims to achieve the following objectives:

1. Integrate the various technical artefacts of the technical Work Packages (WP3-4-5) to the 3 logical bundles, enabling their use as one vertical solution or separate per case bundle
2. Define and implement the necessary application scenarios, application adaptation and experimentation through which the relevant KPIs (use case driven and component driven) are assessed, evaluated and reported back to the component or application owners and the external communities
3. Provide the overall demonstrator executions, aiming to show the effectiveness of the approach as well as the operational version of the RAMP marketplace, to be used in WP7 activities
4. Gather the experiences report from the tests and documenting their outcomes, providing the input for the road mapping activities of the project

1.1 Objectives of the Deliverable

This document describes the preliminary results of PHYSICS WP6 Task T6.1 “Solution Services Integration and Reusable Artefacts Marketplace Platform (RAMP) Creation” and provides the first version (D6.1) of the deliverable (out of the two foreseen in the WP6 work plan for this task). With respect to the general WP6 objectives mentioned before, this deliverable mainly focuses on objectives 1. and 3. (in the latter only for the RAMP related activities).
The results that have been achieved during the work provide key contributions for the fulfilment of the 2nd major WP6 milestone (MS5 – PHYSICS 1st integrated platform release – foreseen for M15 of the project) and provide the first release of the proposed solution.

The document is the companion textual specification of the major result of the deliverable and the task: the first version of the prototype of the integrated PHYSICS solution framework and RAMP, which have been deployed into the PHYSICS blueprint reference target infrastructure.
The document and the integrated PHYSICS solution framework and RAMP setup constitute the overall deliverable and task output.

1.2 Insights from other Tasks and Deliverables

The following picture shows the high level interconnections between Work Package 6 (and T6.1) and the other technical Work Packages that provide the more relevant inputs to the task:

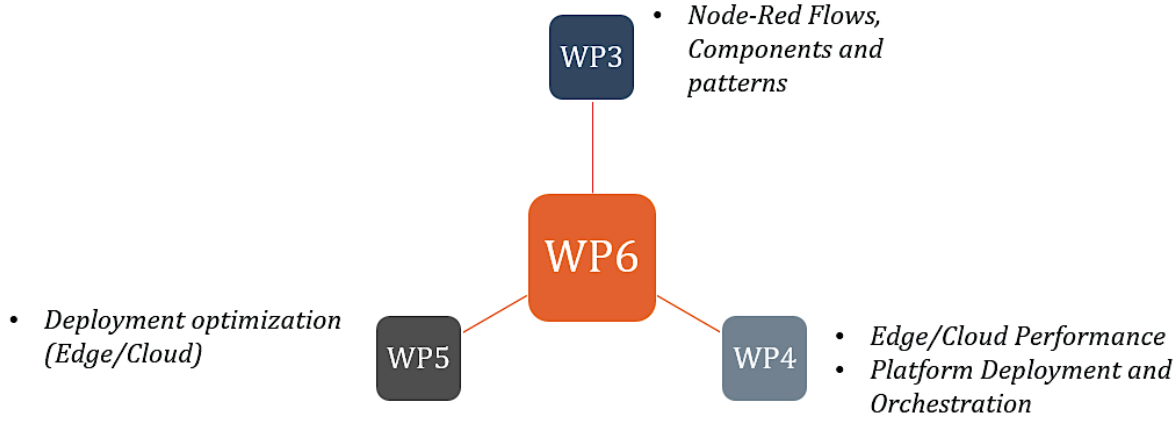


Figure 1 - High level relations between WP6 and T6.1 and the other technical WPs

In particular, important and relevant inputs for WP6 Task 6.1 are the outcomes of:

- WP3 tasks and their first available deliverable "D3.1 – Functional and Semantic Continuum Services Design Framework, Scientific Report and Prototype Description V1"
- WP4 tasks and their first available deliverable "D4.1 – Cloud Platform Services for a Global Continuum Space-Time Continuum Interplay, Scientific Report and Prototype Description v1"
- WP5 tasks and their first available deliverable "D5.1 – Extended Infrastructure Services with Adaptable Algorithms Scientific Report and Prototype Description V1"

Moreover, the outcomes of T6.1 will continue to feed into the remaining WP6 tasks, mainly T6.3 (Use Cases Adaptation & Experimentation) and T6.4 (Use Case Evaluation). These will be used for the 1st iteration of the PHYSICS Pilots and Use Cases Operations and Stakeholders' Evaluation of the proposed solution framework.
Furthermore, as consistently done since the beginning of WP6 T6.1 activities, the work delivered in the task embodies a strict and continuous collaboration and alignment with WP2 tasks and partners, towards the integration of the delivered outcomes (with special focus on the full compliance with the PHYSICS Reference Architecture).
Finally, the delivered integrated PHYSICS solution framework and RAMP marketplace will be fundamental inputs and drivers for WP7 (Exploitation, Dissemination and Impact Creation), with special emphasis on T7.2 (Business Innovation Development & Exploitation).

1.3 Structure

The deliverable consists of the following chapters:

- Chapter 2 "Integrated PHYSICS solution framework and RAMP Overview" provides an overview of the features of the first version of the prototype, its architecture and relationship to the initial version of the general PHYSICS Reference Architecture, with a concrete example

of the Functions Design, Deployment and Execution and a sample sequence flow, via a sample application that uses the capabilities of the integrated PHYSICS solution framework that was created ad hoc

- Chapter 3 “Integrated PHYSICS solution framework Implementation” describes the design and implementation of the components and tools that together form the first version of the prototype of the integrated PHYSICS solution framework
- Chapter 4 “Reusable Artefacts MarketPlace Implementation” the design and implementation of the the first version of the prototype of the RAMP application
- Chapter 5 ”PHYSICS solution framework Integration environment” describes the integrated development and testing environment upon which the PHYSICS solution framework is built, including the Continuous Integration/Continuous Delivery and agile processes put in place to support all the development, testing and integration activities
- Chapter 6 “Conclusions” summarizes the results of the work done in the deliverable and the next steps foreseen for the related tasks
- Chapter 7 “References” provides details of all the cited work

2. INTEGRATED PHYSICS SOLUTION FRAMEWORK AND RAMP OVERVIEW

This chapter provides an overview of the features of the first version of the prototype of the integrated PHYSICS solution framework and RAMP, through its architecture and relationship to the initial version of the general PHYSICS Reference Architecture, with a concrete example of the Functions Design, Deployment and Execution and a sample sequence flow, and finally with a sample application that uses the capabilities of the integrated PHYSICS solution framework that was created ad hoc.

2.1 PHYSICS solution framework architecture overview

This section summarises the architecture of the integrated PHYSICS solution framework. The PHYSICS architecture was described in deliverable D2.4 Reference Architecture SpecificationV1 [1].

The first version of the prototype implementation of the integrated PHYSICS solution framework can be referred to as PHYSICS Minimum Viable Platform (MVP) and is fully aligned to the PHYSICS architecture. The main components of the PHYSICS architecture, implemented in the PHYSICS MVP, are shown in Figure 2.

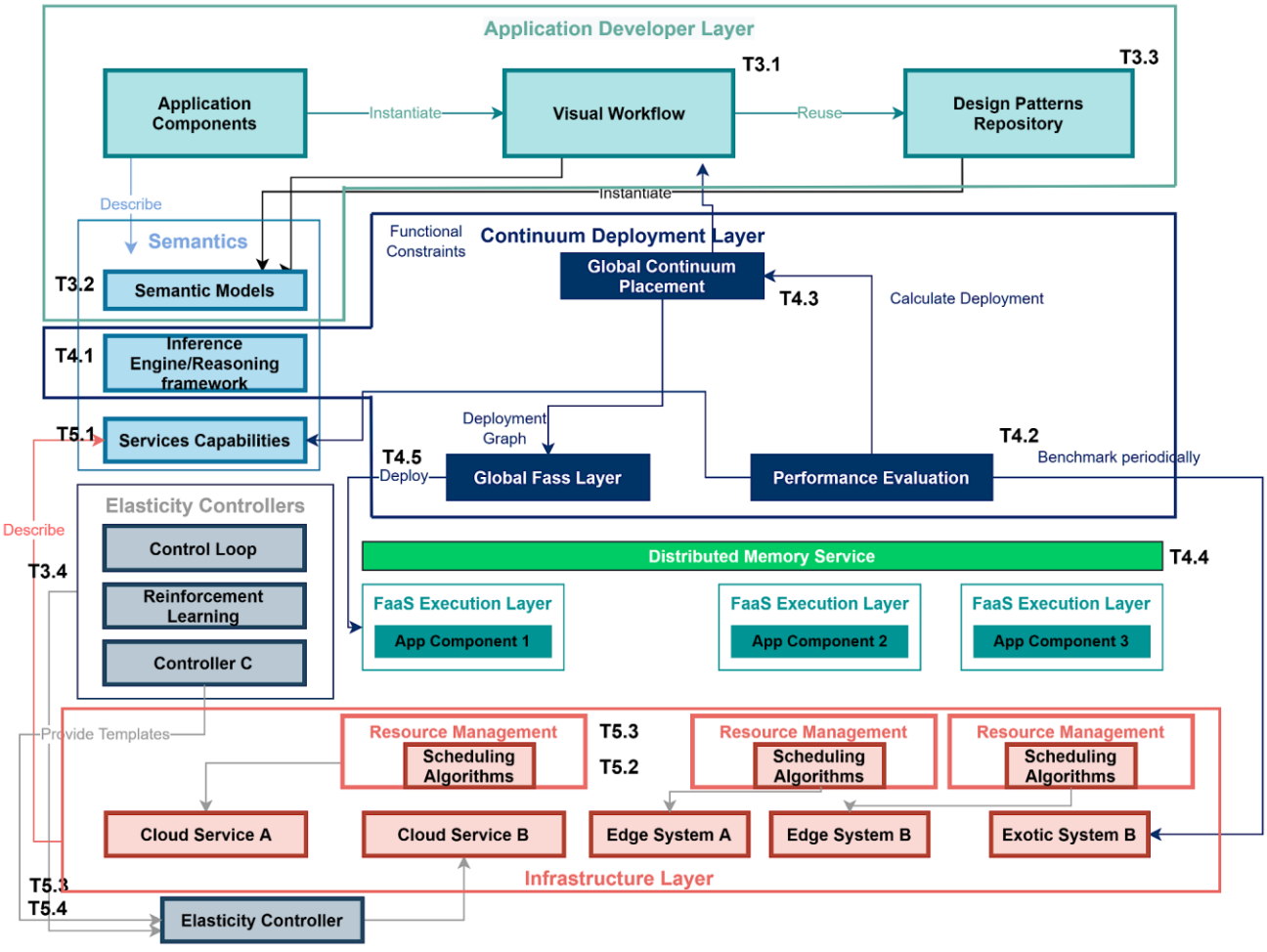


Figure 2 - PHYSICS MVP architecture from RA

The figure presents three layers from top to bottom: Application Developer Layer, Continuum Deployment Layer and Infrastructure Layer, which correspond to the developments in the three technical work packages (WP3 Functional and Semantic Continuum Services Design Framework, WP4 Cloud Platform Services for Global Space-Time Continuum Interplay and WP5 Extended Infrastructure Services and with Adaptable Algorithms).

The top layer, Application Developer Layer, is the entry point for users that design their applications using a Visual Workflow tool. The design of applications is eased by reusing common design patterns such as split-join for function parallelisation, batch processing, data collection, and more, provided by the Design Patterns Repository. Application components (e.g. functions) can be semantically annotated providing information to lower layers that may affect the placement, deployment, operation and configuration of the application (Semantic Models). Application components may have elasticity controllers that regulate the algorithms and resources needed for scaling a component.

The Continuum Deployment Layer is in charge of providing uniform access to the diverse cloud services provided by one or more cloud providers. The Global Continuum Placement is in charge of deciding on the most suitable deployment of applications taking into account the performance of the services, costs, affinity constraints of components: for that purpose it receives the list of candidate services the Reasoning Framework has filtered taking into account the application graph needs and the performance of the services provided by the cloud services and edge devices Performance Evaluation component. The placement of the components is done by the Global FaaS Layer component. The Global FaaS layer abstracts the usage of different data centers from one or more cloud providers. The management of data shared by functions of applications is provided at this level by the Distributed Memory Service.

The Infrastructure Layer provides a view and interface for enabling an optimised operation of the edge and cloud services utilised for the realisation of the application service graph. To this end the Service Capabilities component depicts and models the abilities of each service and resource type. The analysis of different algorithmic approaches for adaptive and real-time provider level scheduling (Scheduling algorithms) so that resources are adapted to current application needs while maintaining overall QoS levels is done by the Resource Management component. The Co-allocation strategies component provides, on behalf of the provider strategies, optimizations to maximise performance.

2.1.1 Design, Deployment and Execution of Functions

Figure 3 presents the steps and components of the PHYSICS framework platform involved in the design, deployment and execution of a function. The infrastructure includes two clusters (Cluster A and Cluster B) each of them with four nodes. We assume that the platform has been previously deployed. The figure presents the minimal number of components involved in each step and focuses on two main processes:

- Process A: Building and testing a flow function (Steps 1 to 3)
- Process B: Creating an app graph that consists of a set of flows and deploying them in the production environment (Steps 4 to 9)

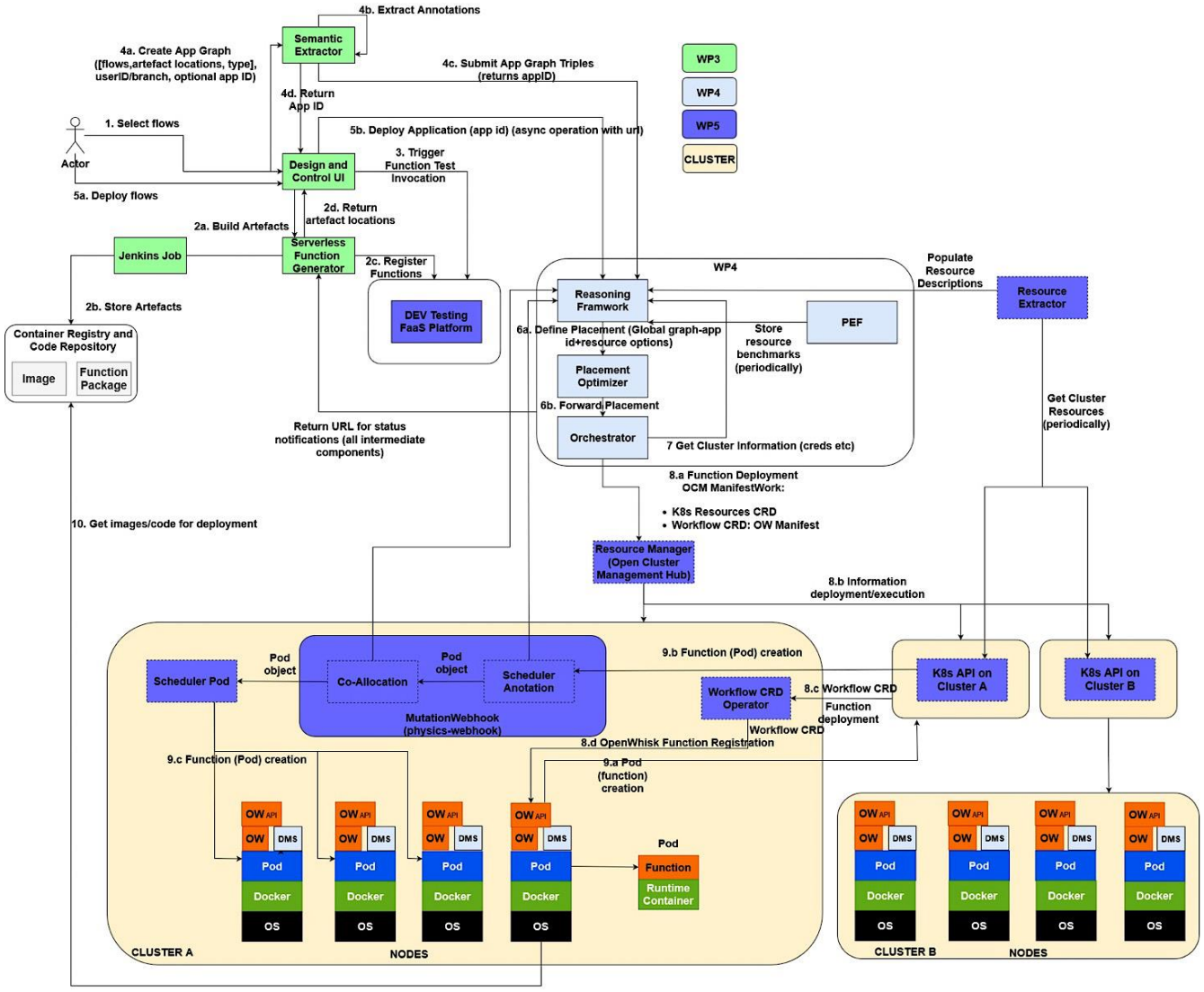


Figure 3 - Design, deployment and execution of a function

Process A assumes that the developer has used the Design and Control UI in order to create a set of flows, by using the PHYSICS provided flow templates, patterns and annotation nodes, as well as including their own application code. Local testing of these inside Node-RED can filter out common minor errors that could take up much time if on each occasion the function needs to be built and deployed in the formal testing or production environment. Furthermore, the developer has added annotations at the function or flow level for desired aspects (e.g. resource selection aspects, deployment options, QoS features etc.). The design process can be summarised as:

1. The developer uses the Design and Control User Interface (UI) to build one or more flows.
2. Function generation:
    a. The selected flow ids are sent to the Serverless Function Generator (SFG) component which orchestrates the process of building the related code or image artefacts with the help of a Jenkins job.
    b. Upon finalisation of the process, the artefacts are stored in the image registry (if they are images) or the code repository as a needed code package.

   c. After the deployable artefacts are ready, the related functions (i.e. OpenWhisk actions) are also registered in a test OpenWhisk (OW) environment.
   d. Upon finalisation the deployable artefact locations are returned to the Control UI in order to be used later on.
3. Registered function testing can be performed in the DEV testing environment. This aids in eliminating errors and bugs that occur in the OW function execution, without the time needed for going over the entire formal deployment process (including optimization and cluster selection, deployment etc. processes).

Process B: Deployment of an application on the production cluster

4. The developer can now initiate the formal deployment process.
   a. The set of the flows, along with the deployable artefact location per flow, are forwarded from the Design and Control UI to the Semantic Extractor (SE).
   b. The latter processes the flows and extracts their structure from the Node-RED JSON specification, as well as any other annotation used by the developer while creating the flow.
   c. The relevant information is mapped to the ontological triples based on the PHYSICS ontology and is forwarded to the WP4 Reasoning Framework for storage.
   d. The reasoning framework also assigns a unique application ID to the flow set, which is returned to the SE and from there to the Design and Control UI. This is the main identifier through which follow-up queries can be performed towards the Reasoning Framework (RF). If an update of the application is needed at a future point in time, the call to the SE should include that application ID to be used in the calls.
5. Once this process is finalised, the developer can initialise the actual deployment for that application ID.
   a. The Design and Control UI receives the request.
   b. The request is forwarded to the RF for initializing the process in WP4. A relevant URL is also given, in order for the various components in WP4 to inform the developer on the progress of the deployment.
6. The Reasoning Framework receives the request and retrieves the descriptions of the related application graph.
   a. It enriches the application graph with candidate resources, after applying the related inference based on user and resource annotations, and forwards the relevant description to the Placement Optimizer for placement. These descriptions include also performance metrics from the Performance Evaluator (PEF), acquired for a given cluster in an offline manner.
   b. The Placement Optimizer selects the most suitable resources for the deployment. This information is sent to the Orchestrator
7. The Orchestrator, which may query the RF for details of the defined cluster (e.g. endpoint, credentials etc.)
8. Application deployment
   a. The orchestrator generates the OCM ManifestWork CRD YAML with K8S resources CRDs and workflow CRD. The Orchestrator sends this information to the Resource Manager (Open Cluster Management Hub).

Each cluster receives this information through the K8s API. In this case Cluster A will process the request

b. The workflow operator is in charge of orchestrating the deployment and registration of the application workflow (set of functions and flows).
c. It processes the workflow CRD (one of the native Kubernetes mechanisms for extending its functionality. It keeps all information about the flows, both data (instance data) and metadata (requirements of the functions…)
d. Registers the function though the OpenWhisk API (OW function registration)

9. Function invocation
    a. When a function is invoked, if there are no Pods warm or pre-warm to execute that function, OpenWhisk triggers the creation of a Pod to execute the function invoking K8s API on the cluster where OpenWhisk is running (Cluster A).
    b. The creation of a Pod in the cluster is intercepted by the MutationWebhook which adds:
        i. The scheduler information to the pod (e.g., energy efficient scheduler)
        ii. Co-allocation strategies to the pod (e.g. co-allocate the pod with pods that do not consume network bandwidth)
    c. The Scheduler pod creates the pod in one node in the cluster and OpenWhisk can execute the function

2.1.2 Sample sequence flow

The following diagram includes more detailed steps, as well as interfaces and sequences of operations in one diagram. This will aid in the creation of an integrated flow in order to complete the needed transfer of functionalities from WP3 to WP4, starting from the description of the application to be deployed as well as including the other necessary options and annotations of the developer, while alerting the latter about the status in each step. The diagram is presented in Figure 4.

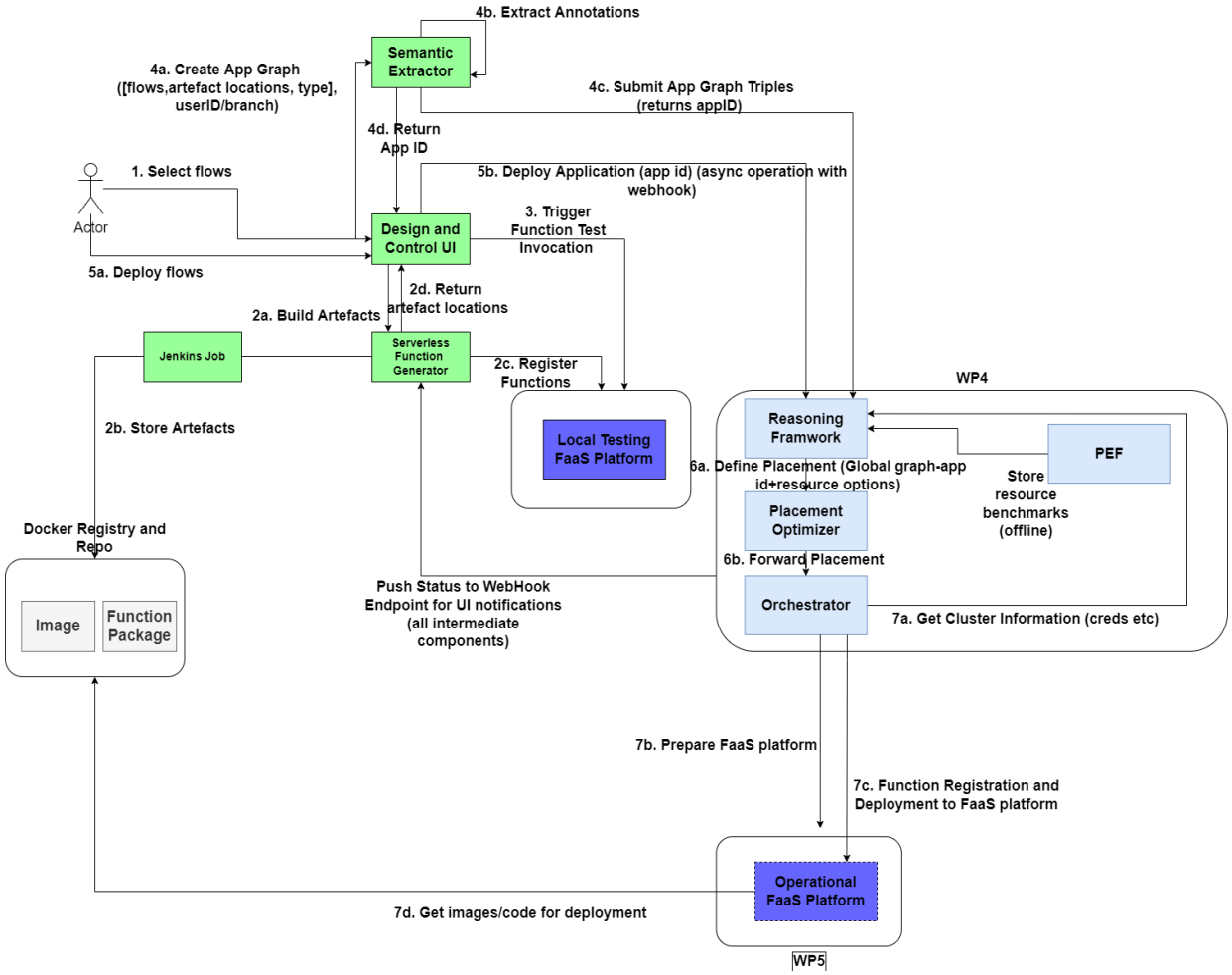


Figure 4 - WP3/4 Integration Diagram for Application Deployment

The developer uses the Design Environment Control UI in order to select a set of flows to build. The selected flow ids are sent to the Serverless Function Generator (SFG) component, which orchestrates the process of building the related code or image artefacts with the help of a Jenkins job. Upon finalisation of the process, the artefacts are stored in the image registry (if they are images) or the code repository as a needed code package. As part of the process, the related functions (or Openwhisk actions) are also registered in a test Openwhisk (OW) environment. This is necessary for testing the actual implementation as a function execution before submitting to the production cluster. Upon finalisation the deployable artefact locations are returned to the Control UI in order to be used later on. Through the Control UI, the developer may trigger the execution of the respective deployed version of the function in the test OW platform.

Once all tests have finished, the developer can initiate the formal deployment process. This starts with the triggering of an operation to create the respective application graph from the selected flows. The set of the flows, along with the deployable artefact location per flow, are forwarded from the Control UI to the Semantic Extractor (SE). The latter processes the flows and extracts their structure from the Node-RED JSON specification, as well as any other annotation used by the developer while creating the flow (as detailed in D3.1). The relevant information is mapped to the ontological triples based on the PHYSICS ontology and is forwarded to the WP4 Reasoning Framework for storage. The reasoning framework also assigns a unique app ID to the flow set, which is returned to the SE and from there to the Control UI. This is the main identifier through which follow-up queries can be performed towards the Reasoning Framework (RF). Once this process is completed, the developer can initialise the actual deployment for that app ID. If an update of the application is needed, the call to the SE should include that app ID to be used in the calls.

During the deployment process, the respective operation is initiated by the Control UI, giving at the same time a return URL in which it should be asynchronously notified for the status of the deployment. Each involved component in that process should use that URL in order to indicate success or failure of the intermediate steps. The first receiving component from WP4 is the Reasoning Engine, which retrieves all triples for the specified app ID, applies the related inference and forwards the relevant description to the Placement Optimizer for placement. The placement is finalised and forwarded to the Orchestrator, which may query the RF for details of the defined cluster (e.g. endpoint, credentials etc.). In the final description sent to the Orchestrator, the location of the deployable artefacts for each function or flow needs to be maintained, since it is needed for the registration process. Once the creation of the relevant FaaS platform is finalised, the Orchestrator can extract the function information from the provided JSON description of the app and perform the relevant registration calls, including the artefact location, towards the Openwhisk interface. In this step, the application functions and flows are deployed on the target platform and are ready to be used.

2.2 Sample application of the integrated PHYSICS solution framework

In order to proceed with the integration at the early stages of the project, a sample application (also referred as "Hello World" in some figures) that uses the capabilities of the integrated PHYSICS solution framework was created. The purpose of this application is primarily to test the design import and build processes of WP3, the WP3/4 interface (app graph creation and storing in the Reasoning Framework), the execution modes supported by the project (Native Openwhisk Sequence, Node-RED runtime function and Service execution), including the WP4/5 interface, as well as annotations passing from the developer in the Design Environment to WP4 and 5. These annotations may help decide on various aspects such as placement, scheduling, sizing etc., so the main purpose of integration in this case relates to how these annotations are propagated from the beginning to the respective component that needs to act upon them. The sample application consists of 3 main flows, each of which targets at a different execution mode.

2.2.1 Sample sequence flow

The sample sequence flow (Figure 5)aims to test the way a simple function chain is declared in Node-RED Things to test in this case primarily refer to how:

- The way function code is extracted from the Node-RED environment and gets wrapped around one of the typical runtimes of Openwhisk (node.js)
- How arguments are passed from one function to the next
- How the Node-RED JSON specification gets translated to the Openwhisk sequence definition
- Furthermore, annotations have been inserted in the following manner
- Execution mode semantic node
- sizingGB=512 in-code annotation for the sample function. This aims to reach the level of registration to Openwhisk with the target memory size for the function container.
- affinity=hello in-code annotation for world function to indicate colocation with the sample function, that aims to reach the node level placement in WP5

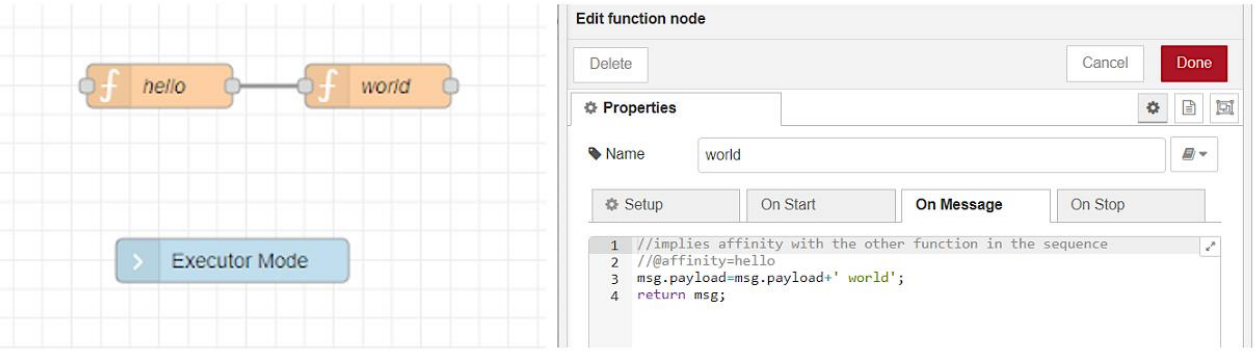


Figure 5 - Sample test sequence

2.2.2 Node-RED flow as function flow

The second flow (Figure 6) refers to sample function wrapped around an OW skeleton pattern. The main goal of this flow is to test the deployment and execution of the Node-RED flow as a function. Things to test in this case include:

- The build process for the Node-RED runtime image creation
- The creation of the app graph (including annotations), the registration to the Reasoning Framework, registration and execution to Openwhisk of a custom image
- The OW interface and argument passing
- The included annotations are:
- importance=high in-code level value and OptimizationGoal=performance, to be taken under consideration by the scheduling layers of WP5
- Sizing annotator node for setting Memory: 512 MB and Timeout: 120000, to be applied in the Node-RED function registration process
- Executor mode for indicating Node-RED flow as function execution

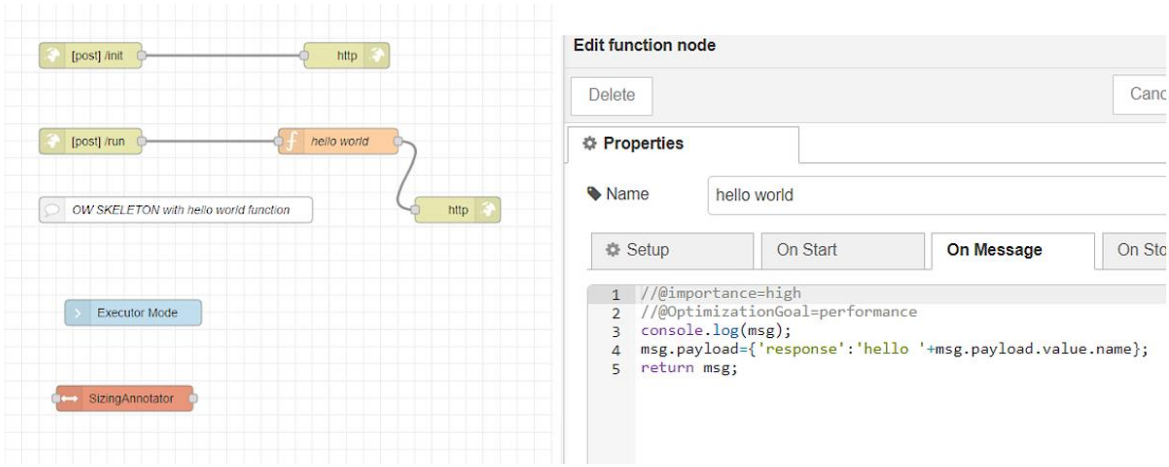


Figure 6 - Sample Node-RED flow as Function

2.2.3 Node-RED flow as Service flow

The third flow refers to a very similar flow as in the previous case of Figure 6, with the main difference that now the deployment needs to be performed as a service. Things to test in this case include:

- Build process for the service image

- Generation of the template for the service manifest
- A new in-code annotation (locality=edge) in order to dictate the deployment on an edge location.

2.3 Reusable Artefacts MarketPlace (RAMP) overview

RAMP will provide access to the project solutions and assets that address shortcomings in FaaS applications development and deployment, while it will enable relevant initiatives to contribute to the PHYSICS vision by hosting their solutions. A high-level overview of its structure is illustrated in Figure 7.

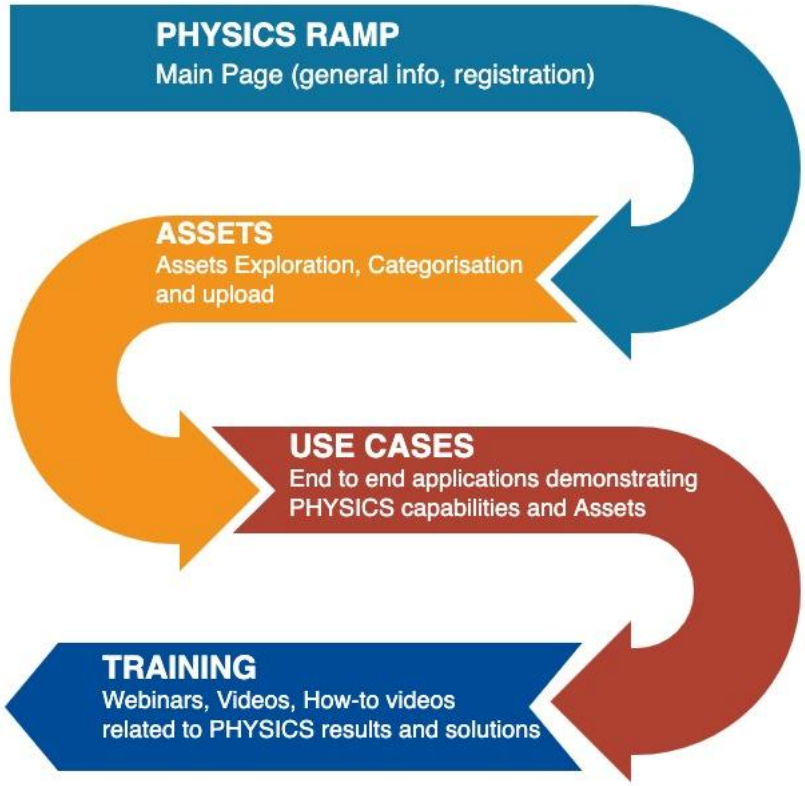


Figure 7 - RAMP High Level Overview

3. INTEGRATED PHYSICS SOLUTION FRAMEWORK IMPLEMENTATION

This chapter covers the design and implementation of the components and tools that together form the first version of the prototype of the integrated PHYSICS solution framework. Typically, most of the components/tools expose their own interfaces (e.g. REST API) to other (client) components/tools directly for a proper invocation and integration.
The following sections describe each component/tool through an overview, information about its technology architecture (design and implementation), a summary of the exposed interfaces (e.g. REST API endpoints), and information about its distribution and configuration for deployment.
The components/tools will be described (with reference to Figure 2) in a sequential and logical order (from top to bottom) aligned to the logical layers of the PHYSICS RA.

3.1 Visual Workflow

3.1.1 Overview

Visual workflow is a web application, which embeds Node-RED environment, communicate with the API of Node-RED and provide features to build and deploy flows to the other PHYSICS Components

3.1.2 Technology architecture

Visual workflow consists of eight components (It's possible that in the future the amount of microservices will increase):

1. Control UI (Frontend application)
2. SFG(Serverless Function Generator) (Backend for Control UI)
3. Artifact Query Service (Microservice)
4. Graph Draft Service (Microservice)
5. Function Service (Microservice)
6. Build Result Processor (Microservice)
7. Artifact Processor (Microservice)
8. Graph Processor (Microservice)

The entry point is the Control UI, which communicate through the REST API with SFG. SFG uses REST API to communicate with Artefact Query Service, Graph Draft Service, Function Service, Semantic Extractor, Inference Engine and Jenkins. Build Result Processor reacts on queue messages, which are emitted after successful Jenkins builds and pushes it to the queues, which are listened by the Artefact Processor and Graph Processor.

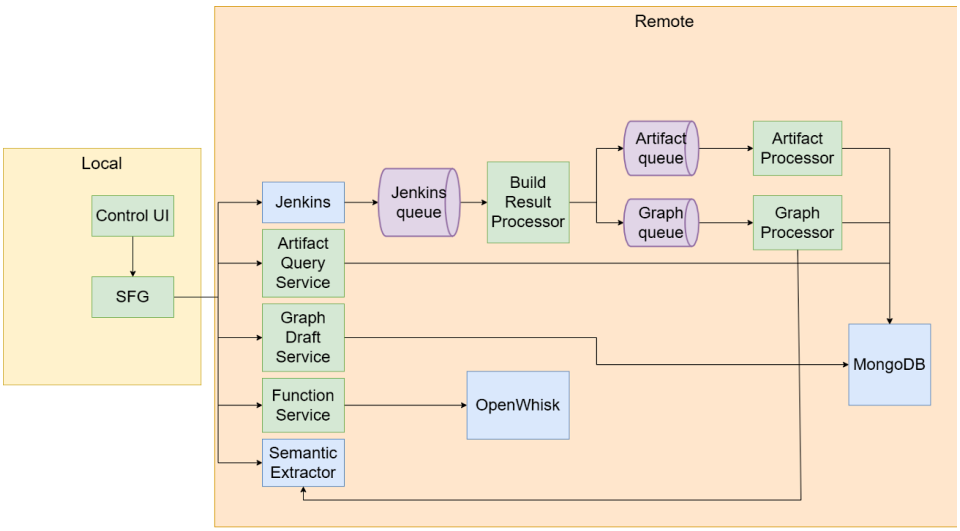


Figure 8 - Components Integration schema

3.1.1 Interfaces/API

Control UI API

Control UI is a frontend application, which provides a user interface to interact with Node-RED and all activities regarding triggering building, testing and deploying flows

SFG API

- Get Flows

Retrieves flows from Node-RED environment and returns it with build information from database.

Table 1 - VW/API- get flow

Path	**/flow**
Method	GET
Request body	-

Success response	HTTP 200 with response body:
	[
	{
	"flow": Flow[],
	"isBuilt": false
	}
	]

Error response	HTTP 404 when Node-RED or database is unavailable

- Build Flows

Receives flow id as input, triggers Jenkins build for given flow and returns URL to Jenkins job.

Table 2 - VW/API- build flow

Path	**/logout**
Method	POST

Request body	{
	"flowId": string
	}

Success response	HTTP 200 with response body: {

	"results": "url_to_jenkins_job", }
Error response	HTTP 404 if triggering job failed, because it's the only status we get from Jenkins as a response, so we can't figure out the reason.

- Get Build Status

Emits status of current Jenkins jobs

Table 3 - VW/API- build status

Path	**/build-status**
Method	Server Side Event
Request body	-

Success response	Message Event with data:
	{
	status: BuildStatus,
	"flowId": string
	}

Error response	Message Event with error message.

- Deploy App Graph

Receives graph id and triggers deployment process of WP4 for given graph.

Table 4 - VW/API- deploy app graph

Path	**/deployment**
Method	POST

Request body	{
	"graphId": string
	}

Success response	HTTP 200 with text response: "Success message"
Error response	Forwarded from WP4

- Create Graph

Receives flows as input and if they already built create graphs in Semantic Extractor, otherwise triggers build for all unbuilt flows and create graph draft in database.

Table 5 - VW/API-create graph

Path	**/graph**
Method	POST

Request body	{
	"flows": Flow[],
	}

Success response	HTTP 200 with text response: "Graph draft created and builds triggered for unbuilt flows" or "Graph created with id: …"
Error response	HTTP 404 in case of any connection problems

- Get All Created Graphs

Returns all created graphs.

Table 6 - VW/API- get all created graphs

Path	**/graph**
Method	GET
Request body	-

Success response	HTTP 200 with response body:
	[
	{
	"id": string,
	"flows": {
	“flow”: Flow,
	“url”: string
	}
	}
	]

Error response	HTTP 404 in case of any connection problems

- Get All Graph Drafts

Returns all graphs, which are still waiting for some of its flows to be built.

Table 7 - VW/API- get all graph drafts

Path	**/graph/draft**
Method	GET
Request body	-

Success response	HTTP 200 with response body:
	{
	"builtFlows": {
	“flow”: Flow,
	“url”: string
	}[],
	"unbuiltFlows": Flow[]
	}

Error response	HTTP 404 in case of any connection problems

- Get Functions

Returns all available functions’ names.

Table 8 - VW/API get functions

Path	**/function**
Method	GET
Request body	-
Success response	HTTP 200 with response body: string[] as function names
Error response	HTTP 404 in case of any connection problems

- Invoke Functions

Receives function name and parameters, invoke the function and returns activation id of this call

Table 9 - VW/API- invoke functions

Path	**/function/invoke**
Method	POST
Request body	{

	functionName: string, params: JSON object }
Success response	HTTP 200 with response body: String as activation id
Error response	HTTP 404 in case of any connection problems

- Get function activation result

Receives activation id as query parameter and return result of given function activation

Table 10 - VW/API- get function activation

Path	**/function/:activationId**
Method	GET
Request body	-
Success response	HTTP 200 with response body containing function result
Error response	HTTP 404 in case of any connection problems

Artefact Query Service API

- Get Artefacts

Receives optional flow array and returns all artifacts from database for given flows

Table 11 - VW/API- get artefact

Path	**/artefact**
Method	GET
Request body	flows: string[]
Success response	HTTP 200 with response body: { “flow”: Flow, “url”: string }[],
Error response	HTTP 404 in case of any connection problems

Graph Draft Service API

- Get Drafts

Returns all graph drafts from database

Table 12 - VW/API- get draft service

Path	**/draft**
Method	GET
Request body	-

Success response	HTTP 200 with response body:
	{
	"builtFlows": {
	“flow”: Flow,
	“url”: string
	}[],
	"unbuiltFlows": Flow[]
	}

Error response	HTTP 404 in case of any connection problems

Function Service API

- Get Functions

Returns all available functions' names.

Table 13 - VW/API- get function service

Path	**/function**
Method	GET
Request body	-
Success response	HTTP 200 with response body: string[] as function names
Error response	HTTP 404 in case of any connection problems

- Invoke Functions

Receives function name and parameters, invoke the function and returns activation id of this call

Table 14 - VW/API- invoke function

Path	**/function/invoke**
Method	POST

Request body	{
	functionName: string,
	params: JSON object
	}

Success response	HTTP 200 with response body: String as activation id
Error response	HTTP 404 in case of any connection problems

- Get function activation result

Receives activation id as query parameter and return result of given function activation

Table 15 - VW/API- get activationID

Path	**/function/:activationId**
Method	GET
Request body	-
Success response	HTTP 200 with response body containing function result
Error response	HTTP 404 in case of any connection problems

3.1.1 Build Result Processor API

The Build Result Processor reacts to the messages on queue emitted after successful Jenkins Build. Format of the message is described here:
https://plugins.jenkins.io/mq-notifier/
We are interested only in the message with state: 'COMPLETED" and push mapped message to queues for artifact processor and graph processor

3.1.2 Artefact Processor

Artifact Processor reacts on the message on queue emitted from Build Result Processor in format:

```
{
            "flow": Flow,
            "url": string
```

```
}
```

3.1.1 Graph Processor

Graph Processor reacts on the message on queue emitted from Build Result Processor in format:

```
{
                “flow”: Flow,
                “url”: string
}
```

3.1.1 Control UI description

Control UI consists of two main tabs: one with Node-RED environment and second with Admin Panel, which contains all logic for communication within the PHYSICS platform.
In the following the two main tabs are described in detail:

- Node-Red

Embedded Node-RED environment, which allows developers to use it as a standalone application.

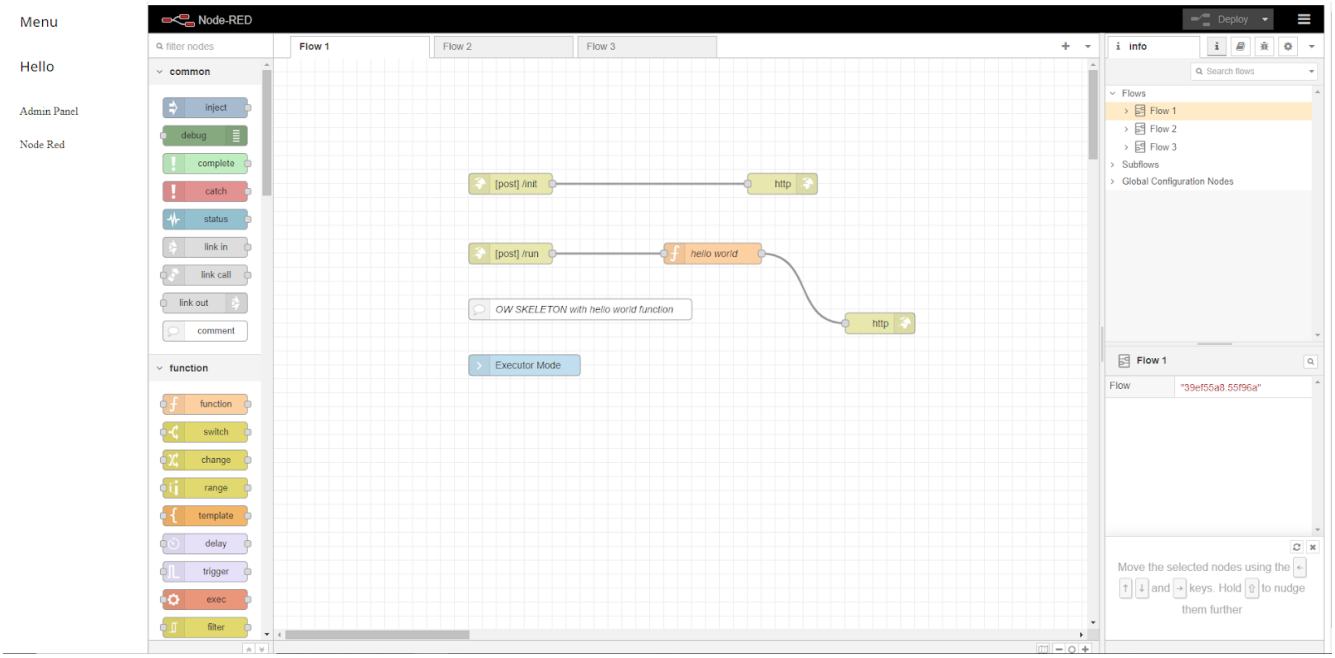


Figure 9 - Node-Red environment

- Build tab in Admin Panel

Dialog where developers can choose flow to build artefacts for.

Figure 10 - Build flow

- Test tab in Admin Panel

Dialog where developers can test flows deployed to the test OpenWhisk environment

Figure 11 - Test flow

- Graph tab in Admin Panel

Dialog where developers can see all the draft and created graphs.

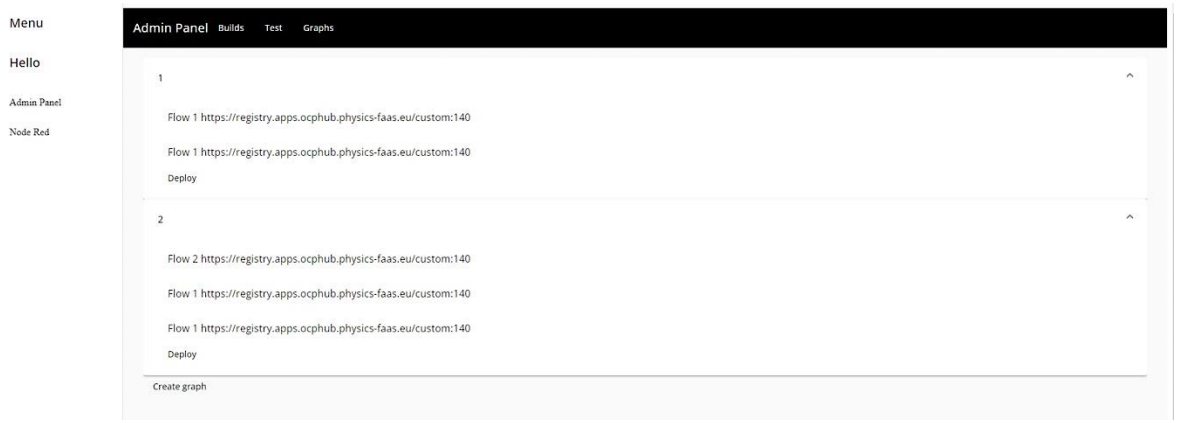


Figure 12 - See created and draft graphs

Developers can also create new graphs here.

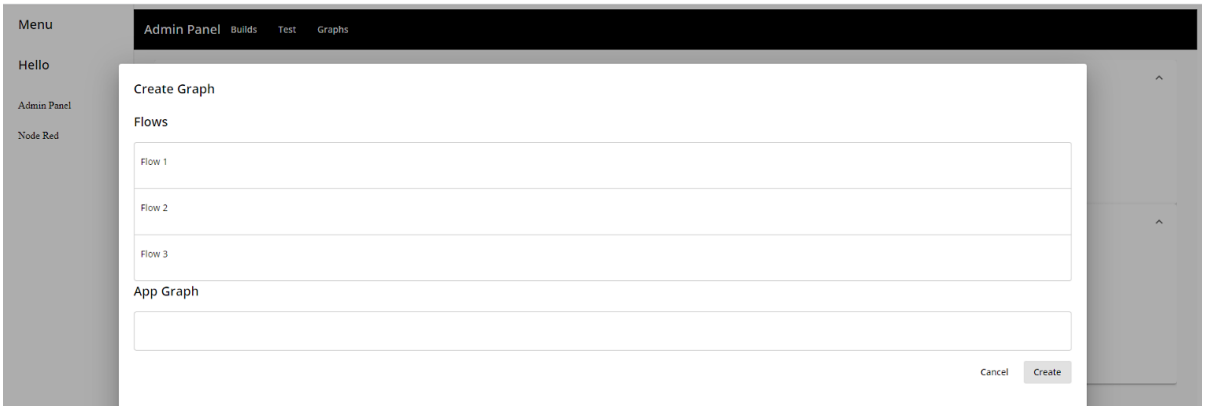


Figure 13 - Create a new graph

3.1.2 Distribution, deployment and configuration

All subcomponents are prepared as separated docker images. Artefact Query Service, Graph Draft Service, Build Result Processor, Artefact Processor and Graph Processor are hosted on the PHYSICS Kubernetes Cluster. SFG and Control UI are meant to be hosted locally on the user machine together

with the Node-RED environment. The whole local environment can be started utilising docker-compose files.

3.2 Semantic Extractor

3.2.1 Overview

The Semantic Extractor (SE) is an intermediate component that aims at transforming the Node-RED defined sequences and annotations into the semantic structure needed by the Reasoning Framework (RF) of T4.1. As such it implements the first stage of the specification transformation described in Section 2.1. The SE receives the flows that need transformation from the Design Environment, processes the respective JSON structure and maps the declared annotations to the necessary JSON-LD format that describes the semantic triples. The SE also processes the structure of the graph, based on the Node-RED specification for function wiring, and stores this structure in the RF.

3.2.2 Technology architecture

The SE is implemented as a Node-RED flow, since Node-RED is by default very good for processing structures and annotations. This way, the implemented flow can be included inside the Design Environment, if needed for minimising the number of services running, or it can be deployed as a separate microservice for better modularity and independence of deployment, update etc.

The extractor is based mainly on the jsonata[3] and jsonld.js[4] libraries for transformation, semantics extraction, and validation of the resulting json-ld application model, along with a custom parsing logic for annotation extraction from code included in function nodes. Other helper libraries included are stdlib-js[5], validate.io[6], clean-deep[7] and json-schema-library[8], which help implement several filtering, validation and transformation functions. Several pieces of custom logic for semantic extraction are implemented, based on the input structures from Node-RED, to produce an output that abides by the PHYSICS ontology.

3.2.3 Interfaces/API

The SE includes one method that appears in the following table.

Table 16 - SE-API-semantic retrival

Path	**/extract**
Method	GET
Request body	-{“flow”: JSON output of Node-RED flow, “artefactLocation”: URL, “type”: either code or image, “userID/branchID”: id of a user or branch, “appID”: optional in case we need to update an app id] }
Success response	HTTP 200 with response body the app ID as generated from the Reasoning Framework:

[3] http://docs.jsonata.org/overview.html
[4] https://github.com/digitalbazaar/jsonld.js
[5] https://github.com/stdlib-js
[6] https://www.npmjs.com/package/validate.io
[7] https://www.npmjs.com/package/clean-deep
[8] https://www.npmjs.com/package/json-schema-library

	{
	“appID”: “dsdsuJJjaj….”
	}

Error response	HTTP 40X

3.2.4 Distribution, deployment and configuration

The SE is currently packaged as a Node-RED flow. Initialization is partially done inside the flow, where any needed internal aspects, such as supported nodes list, JSON-LD context, and reusable utilities, are injected in the flow context at deployment. Each function node imports node modules through its setup pane, and reused logic through destructuring of the flow context.

The only required configuration for the semantic extractor is the endpoints of the Reasoning Engine and the Node-RED environments which define annotators, subflows and patterns. For cases where the flow to be submitted is not provided, but only a reference/ID to it is, the target Node-RED environment from where the flow is to be retrieved should also be provided in the configuration. The configuration can either be done within the flow, or through an environment file.

The SE can be distributed not only as a Node-RED flow, but also as a container that includes the Node-RED environment, the npm package dependencies and the SE flow itself. The npm packages mentioned in section 3.3.2 are installed in the Node-RED environment the SE is based on. The resulting container image can be deployed along with the main graphical composer and reasoning engine and is included in the relevant compose file of the overall Design Environment.

3.2.5 Sample Application Transformation

Given each flow ID input, the extractor provides the corresponding JSON-LD output, which is JSON-LD that follows the current ontology. Three example transformations are shown below, based on the test Hello World application that was presented in Section 2.1. On the left of each figure, the example flow is given, as it appears on Node-Red. The right has the JSON-LD output that is sent to the reasoning engine for each case (with has JSONDescription property retracted). This JSON-LD is a serialization format of semantic triples that are inputs for the reasoning engine. A native sequence transformation appears in Figure 14, a transformation of a Node-RED based OpenWhisk function in Figure 15 and while the transformation of a Node-RED based flow to a Node-RED service in Figure 16. In all cases, the different respective annotations used in each flow are mapped to the respective triples.

Figure 14 - SE transformation of native OW sequence

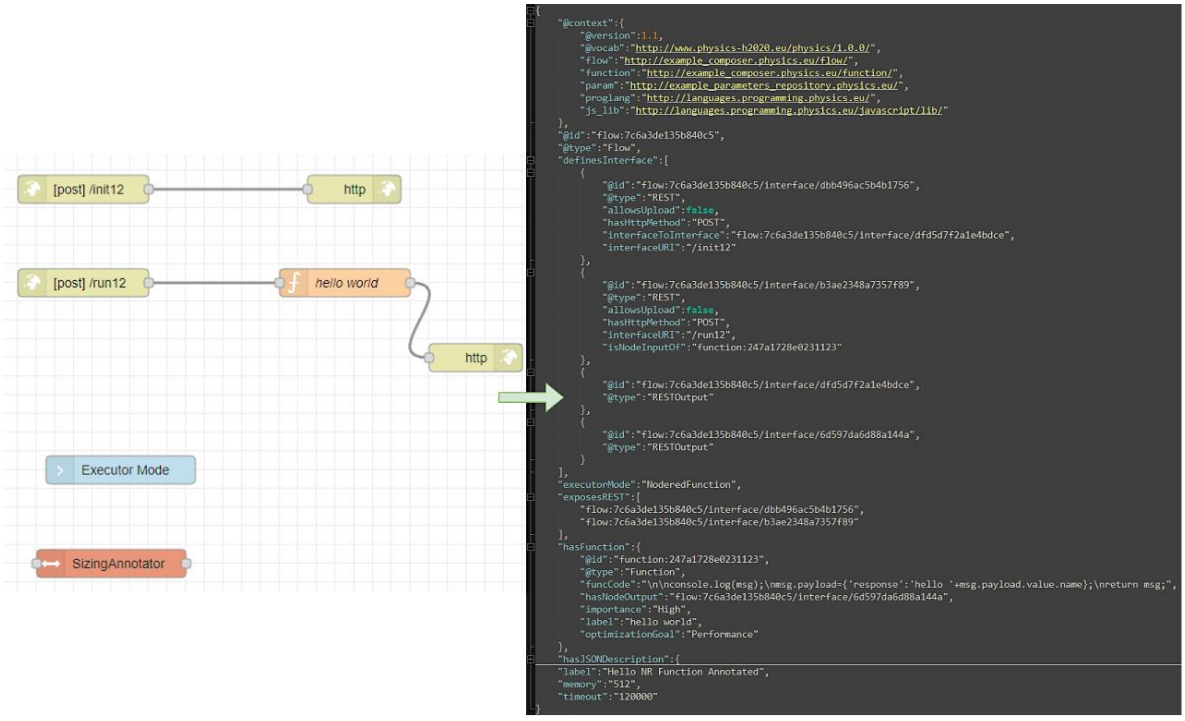


Figure 15 - SE transformation of Hello World Openwhisk Node-RED function

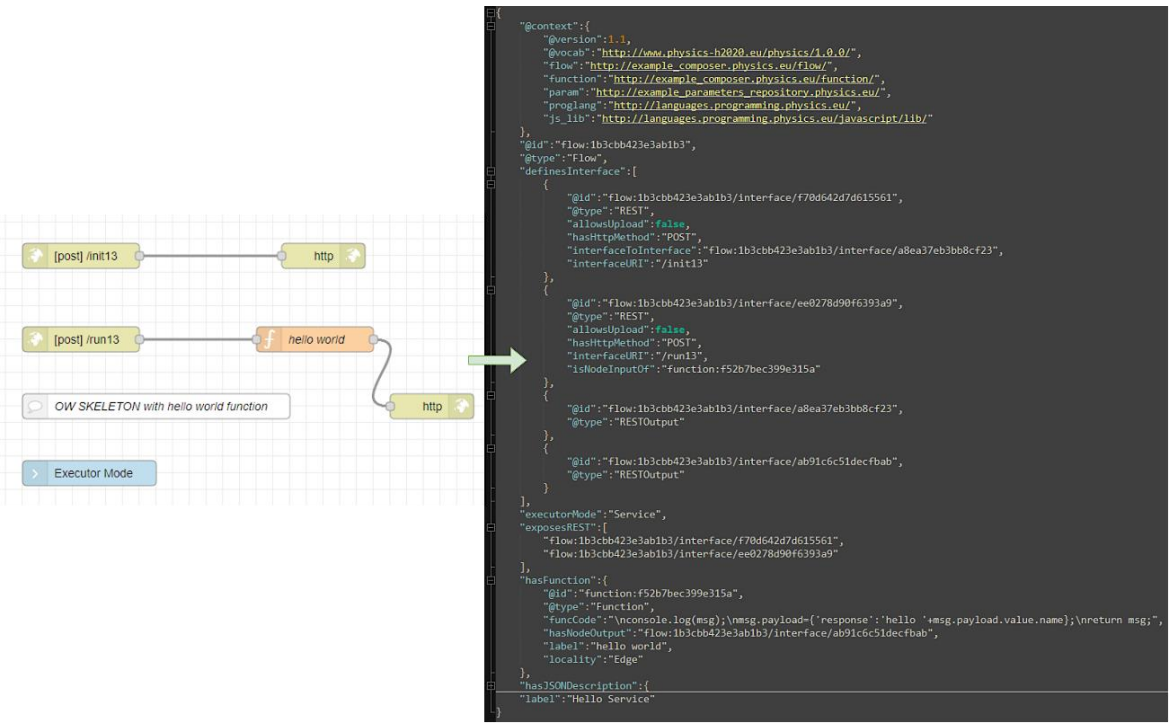


Figure 16 - SE transformation of Hello World Openwhisk Node-RED Service

The SE parses the structure of the original flow description, and makes a transformation based on the type of Node. Some nodes are interpreted as interfaces, while others are parsed for a specific purpose, such as the annotator nodes, and others may be ignored completely, such as the comments. The wires connecting the nodes are also parsed as interfaces, depending on the types of nodes they connect. Each function node has its code parsed for extractable function-level annotations. These annotations are then interpreted, and added as the appropriate attribute to the function. Similarly, the annotator nodes are parsed for their contained environment information, and the contained properties are interpreted and added to the Flow.

3.3 Patterns

3.3.1 Overview

The Patterns aim at offering reusable and parametric operational capabilities to the developers in order to enable an easier and more abstracted flow creation process. Patterns have been created for workflow enhancement, load distribution, message manipulation etc.

3.3.2 Technology architecture

The patterns are primarily Node-RED flows and subflows to integrate them directly integrated into the PHYSICS Design Environment. They can be executed as services or as functions, where applicable, based on the defined annotations. The patterns are available in the PHYSICS Gogs repository, so that they are included directly in the Node-RED base image available to the developer using the Design Environment.

3.3.3 Interfaces/API

Each pattern or subflow/node comes with its own specification in relation to its usage. The interfaces are through the fields of the incoming message to the subflow. The information is included in each

pattern documentation to be directly accessible in the Node-RED environment by the developer. Specific information and examples of usage for each pattern are included in D3.1.

3.3.4 Distribution, deployment and configuration

The Patterns are packaged and distributed as subflows that are available in the PHYSICS repository (Node-RED JSON descriptions including the code) and are embedded in the spawned Node-RED image coming with the Design Environment described in the previous sections, along with any dependencies they may have in terms of extra needed Node-RED nodes. Once the developer spawns the PHYSICS provided Node-RED image, they will have the Patterns available in a relevant node menu of the palette (Figure 17).

They have also been made available in online repositories (e.g. Node-RED flows repository[9]). They can be copied directly in any Node-RED environment, given that they are represented by a JSON string including the specification and extra code of the nodes used and their interconnections. Especially from the online Node-RED repo mode, the dependencies of each subflow are identified (in terms of other Node-RED nodes needed by the flow), so that the developer can pre-install them.

After inclusion in their design environment, the developers can drag and drop the respective subflow nodes. By double clicking on them, a relevant UI is available with the input and configuration parameters of each flow. The inputs can also be passed as message fields in most cases. Where configuration is needed (e.g. credentials for accessing an external service), this is highlighted in the README file as well as by comment nodes inside the flows. Each pattern also comes with a relevant readme file (Figure 18), available in the Node-RED environment, in which details of the operation are given as well as the needed specification of the input message if not configured by the UI.

[9] PHYSICS Patterns Collection, available at: https://flows.nodered.org/collection/G33TNU6z-Qia

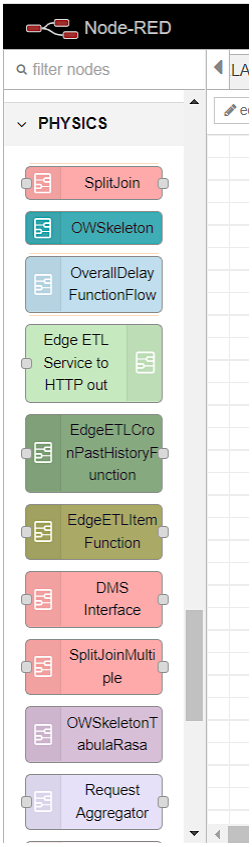


Figure 17 - PHYSICS Patterns Palette in Node-RED

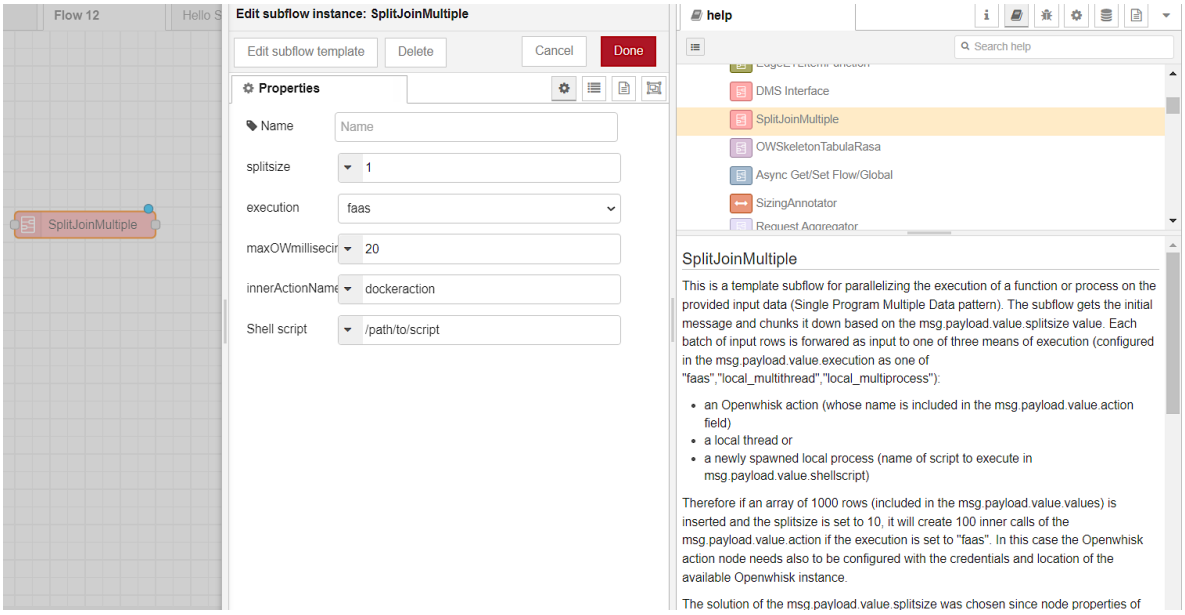


Figure 18 - Example UI configuration and README file in Pattern Node

In the future, in order to enable automatic updates of patterns to reach the developers, we want to package them into Node-RED modules[10]. We will have three modules containing the patterns for three categories of patterns.

[10] Subflow modules packaging: https://nodered.org/docs/creating-nodes/subflow-modules

3.3.5 Individual integration points of Patterns with the Data Management Service of T4.4

The Data Management Service (DMS) service in PHYSICS offers an interface through 3 main OpenWhisk(OW) actions that are foreseen for read/write/lookup operations, as indicated in the respective section. In order to make that more abstract, a relevant DMS interface (Figure 19) node has been created in the Node-RED environment, which appears in the following figure. The node has been created with the relevant fields needed in the DMS interface. The data to be written needs to be included in the msg.payload.data field of the incoming message. The operation can be selected from the UI (Read/Write/Lookup) or can be set via the msg.operation field. The job_id can be set from the UI or can be set via the msg.payload.jobid field. The pocket destination file can be set from the UI or can be set via the msg.payload.pocket_dst_file field. The pocket server location can be set from the UI or via the msg.payload.location field. The implementation of the node internally uses the typical OW action client node of Node-RED, which needs to be configured with the respective OW credentials. The namespace can also be configured through the UI or through the msg.payload.namespace field.

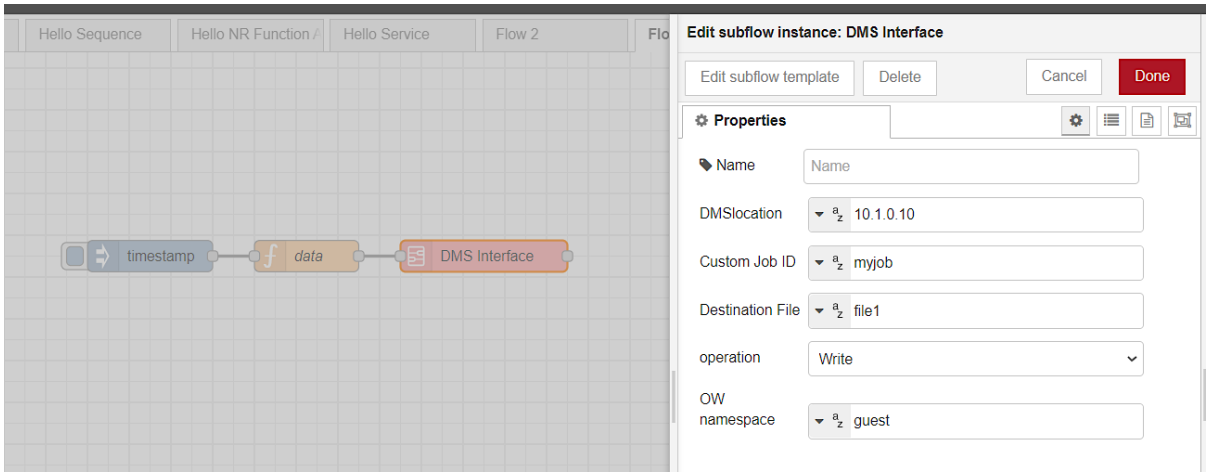


Figure 19 - DMS Interface Node-RED node

3.4 Elasticity controllers

3.4.1 Overview

The elasticity controllers are components which will be integrated into the infrastructure layer (either through the webhook provided by the resource management controllers component, or by extending Kubernetes horizontal pod autoscaling[11]). These components extend the existing Kubernetes scalability controllers to adjust to additional requirements such as supporting green computing (using less energy) and saving deployment cost (using cheaper instances). The newly developed elasticity controllers are expected to use a broader view of the cluster and look at a wider set of metrics than the existing controllers. Additionally, they implement interfaces to other PHYSICS components for further configuration. While existing controllers look at the existing known per-pod metrics and can take decisions based on the load on the specific pod instances, we want to extend this view in 3 ways:

[11] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

- Per pod custom metrics to monitor application specific metrics which can affect the elasticity policy. This is based on the K8s Scaling on custom metrics feature.
- Scaling which is based on multiple metrics, not only one. This way we can evaluate multiple configurations and take the maximal one. This is based on the K8s Scaling on multiple metrics feature.

Note: *This policy may not suit our goal directly since it takes the "maximal" configuration, so we may need to extend it or use one of the alternatives.*

- A global view which can affect the elasticity policy based on all the information in the monitoring system, and on cluster deployment metadata (such as energy consumption, price, topology and network usage). This additional metadata is modelled in the semantic model so it can be gathered and accessed by any PHYSICS component (such as the elasticity controllers). This is based on the K8s Support for metrics APIs feature.

Note: *This component is dependent on the work of other work packages and is planned for the second half of the PHYSICS project - accordingly the work on this component has not started yet*

3.5 Semantics Block

3.5.1 Overview

The Semantics Block component consists of two tasks related to semantics and the PHYSICS ontology: Reasoning Framework for Semantic Matching and Runtime adaptation (T4.1) and Semantic Models for Service Characteristics (T5.1), abbreviated as Reasoning Framework and Service Semantics, respectively. The goal of this component is twofold; first, the component will store application and resource metadata in graphs by first merging them and performing the necessary transformations into a dedicated semantic graph database and afterwards, it reasons those graphs to extract insights and deduce new information. Matching between the two graphs, deduction of specific relationships to connect resource and application individuals, and systematic observation of these metadata enables other components functionalities by providing the necessary information for them to act.
This design choice to combine the functionalities of the two tasks comes at hand as it eliminates the need for extra communication between them during certain operations, such as data injection from the service semantics to the central graph database. It also provides a degree of "future-proofing" as certain information needs to be propagated to WP5 components. By integrating the components as one service, the Service Semantics will be triggered faster and act upon the information to propagate necessary results.

3.5.2 Technology architecture

The first version of the Semantics Block consists of 3 main components, illustrated in Figure 20. Reasoning Framework Inference Engine and Resource Semantics components are implemented in Python language. These components are REST microservices based on the Flask[12] framework, and the development of the provided API interfaces is compliant with the OpenAPI[13] standard. The third component is a Semantic Graph Database facilitating the storage of the input RDF triples (i.e., Application and Resource triples), SPARQL querying and reasoning over the stored data. The community version of AllegroGraph[14] was opted for the data storage needs of the Semantics Block. AllegroGraph provides an architecture through the REST protocol and is characterised by the efficient

[12] https://flask.palletsprojects.com/en/2.0.x/#api-reference
[13] https://spec.openapis.org/oas/latest.html
[14] https://allegrograph.com/

use of memory by combining disk storage, making it possible to scale up to one billion nodes, always maintaining top performance. Basically, it provides services including vision building, rapid prototyping, and proof-of-concept development, complete enterprise technology solution stack, and best practices to maximise value from semantic technologies [2].

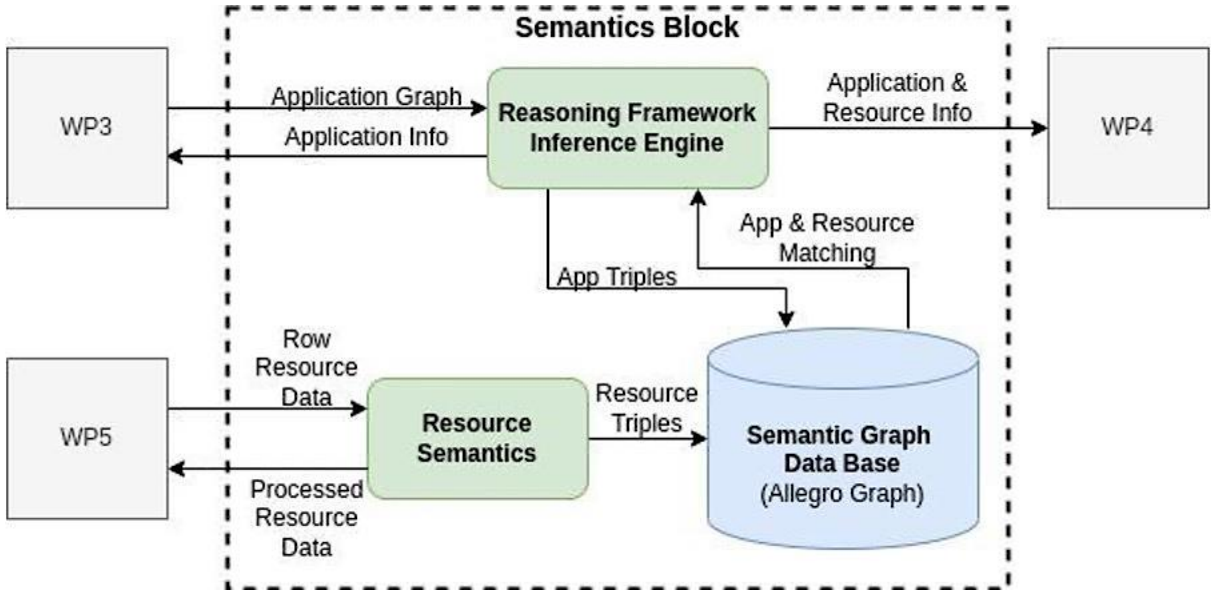


Figure 20 - Semantics Block internal components

3.5.3 Interfaces/API

This section lists all the currently available endpoints of the Semantics Block service. Although most are reached programmatically facilitating back-end platform operations, some are intended to be used visually, such as the ontology visualization endpoint.

Table 17 - Semantics Block Visualization endpoints

Method	Path/URI	Description	Request/ Parameters	Response
GET	/api/v1/ontology	Retrieve a visual representation of the PHYSICS ontology.	-	Web Interface
GET	/api/v1/doc	Get the OpenAPI documentation of the Semantics Block	-	Web Interface

Table 18 - Semantics Block endpoints for Resources

Method	Path/URI	Description	Request/ Parameters	Response
POST	/api/v1/cluster/{cluster_json}{nodes_json}{type}	Register a cluster along with the respective resource description. This includes the cluster, node and resource unit (Physical or VM) specifics.	JSON[15]	200, {cluster_id :"stored"
GET	/api/v1/cluster	List all the available clusters and their specs.	-	200, JSON

15 A json with 3 nested json objects and pass their identity as parameters in the post. For example, if the user wants to register a Kubernetes cluster of m nodes hosted on raspberry pi the parameters should be like appended to the URL as follows: ?1=cluster&2=nodes&3=raspberry

GET	/api/v1/cluster/{cluster_id}	Get a cluster and its complete specs.	string:cluster_id	200, JSON
DELETE	/api/v1/cluster/{cluster_id}	Delete a cluster and its triples from the DB.	string:cluster_id	200, {cluster_id:"deleted"}
PUT	/api/v1/cluster/{cluster_id}	Update a cluster and its triples from the DB.	string:cluster_id	200, {cluster_id:"updated"}

Table 19 - Semantics Block endpoints for Applications and Semantic Matching

Method	Path/URI	Description	Expects/ Parameters	Response
POST	/api/v1/application	Store a new application graph in the DB.	JSON-LD	200, {app_id:"stored"
PUT	/api/v1/application	Updates a stored application graph in the DB.	JSON-LD	200, {app_id:"updated"}
GET	/api/v1/application	Returns the ids and the names of the stored applications.	-	200, JSON
GET	/api/v1/application/{app_id}	Get an application graph from the DB.	string:app_id	200, JSON
DELETE	/api/v1/application/{app_id}	Delete an application and its triples from the DB.	string:app_id	200, {app_id:"deleted"}
GET	/api/v1/application/run/{app_id}	Get the required information for the deployment of the given application (to be used by the Optimizer and the Orchestrator).	string:app_id	200, JSON

3.5.4 Distribution, deployment and configuration

The three integrated components in the Semantics Block (i.e., Resource Semantics, Reasoning Framework, and Semantic Graph DB) are containerized to a single service as Docker Image[16] with their source code being maintained in the official repository of the PHYSICS project (Gogs repository: https://gogs.apps.ocphub.physics-faas.eu/WP4/semantics-block). Their latest built image is stored on the PHYSICS Harbor repository, leveraging the Jenkins pipeline for continuous integration (CI). The deployment of the Semantics Block will follow the same process as the other platform components in the PHYSICS AWS testbed in a dedicated namespace. The continuous delivery (CD) part will also be automated through the Jenkins pipeline.

[16] https://www.docker.com/

The specific instructions required to parametrize, build and deploy the Semantics Block (i.e., Docker-Compose, Jenkins, and K8s YAML config files) will also be available in its Gogs repository.

3.6 Performance Evaluation Framework

3.6.1 Overview

The Performance Evaluation Framework (PEF) aims at enabling REST based launching of relevant performance driven tests against a target service platform. To do so, it incorporates adapted Jmeter clients that are parametric and can be used in order to implement the necessary load generation process. The results are acquired and stored by PEF and available through a range of queries, either directly through relevant REST APIs, potentially with tailored metrics, or through the inclusion in the semantic model of the resource. In the latter case, the results are acquired through the relevant interfaces of the Reasoning Framework

3.6.2 Technology architecture

PEF is based on Docker containers in order to provide load injection and model creation processes in a scalable and abstract manner. In order to guarantee the validity of the measurement process and acquired performance information, PEF tightly regulates the way these containers are launched and synchronised during the preparation and execution of a test, as well as the result gathering in the end. This is performed through direct interaction with the Docker API.
To this end, it leverages the abilities of the Jmeter distributed mode of execution, for which it prepares the environment and regulates the creation of the relevant individual slave Jmeter clients. To ensure validity of the results, the load generation clients need to reside externally to the target service containers, in order not to add overhead to its execution from the increased load.
For each submitted test, the relevant needed Jmeter containers are spawned and dynamically configured in order to detect one another and proceed with their internal coordination plan. All the coordination is performed through relevant flows implemented in Node-RED, which also implements the REST API layer to expose the results, the control of the tests as well as the creation and usage of the models.

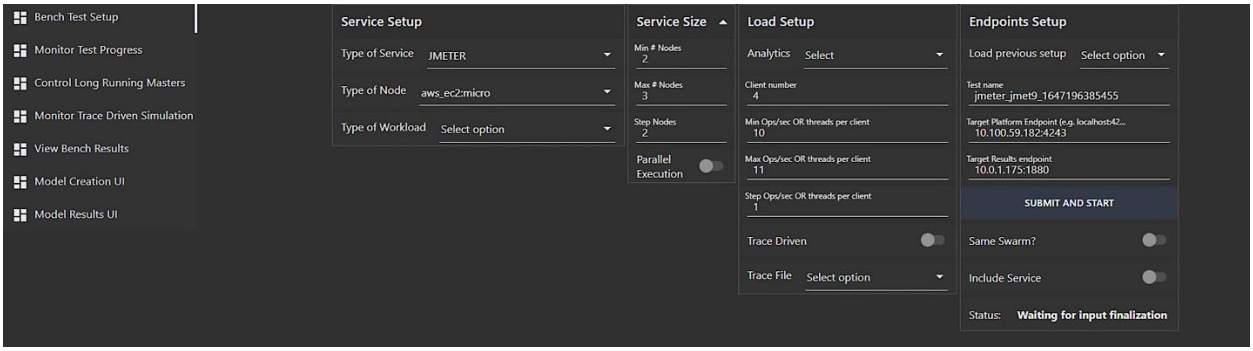


Figure 21 - Setup of a Benchmark Test

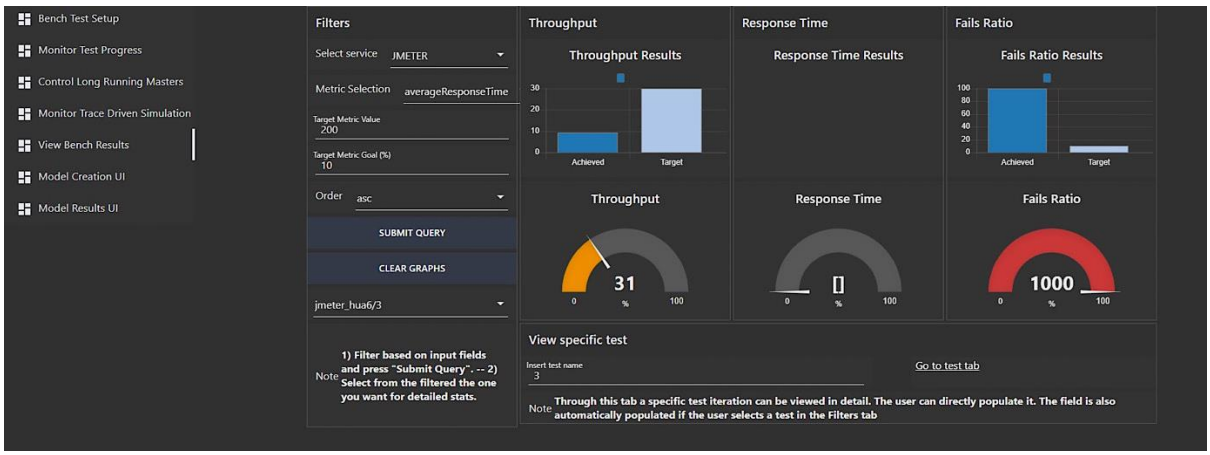


Figure 22 - View of the Bench Results

3.6.3 Interfaces/API

The API of the tool has been included in D4.1 in detail, while the relevant documentation is also available in the repository.
Furthermore, the tool comes with a set of useful UIs, which can aid in its usage. Examples include the creation of a simple test (Figure 21) and the viewing of the test results (Figure 22), although in the context of an automated usage the REST API manner is anticipated to be used.

3.6.4 Distribution, deployment and configuration

The main PEF tool is available as a docker image. Once deployed, the user needs to login to the main environment (typical Node-RED UI interface) and configure the required information. This includes the main location of the repository from which the various jmx files will be retrieved for execution, as well as the endpoint for the database in which the results are stored.
PEF also utilises a specially packaged version of underlying images with relation to various internal functionalities of the tool[17]:
Image with relation to the packaged jmeter version for load generation
Image with relation to the packaged GNU Octave environment needed for model creation as well as model inference
Images with relation to other helper function packaging (e.g. functions packaged as containers for the Node-RED orchestration execution, artificial delays etc.
The user does not need to perform any operation with relation to the usage of these images as they are available publicly and are pulled during in the preparation and usage scripts of the main tool.

3.6.5 Individual integration points of PEF with other components

The integration points of PEF with other components include:

- Metrics exposed to the semantic resource model of T5.1, which is the template of information that needs to be populated and is available for querying through the Reasoning Framework's API.
- Integration with FunctionBench abilities offered by T5.2

With relation to the semantic model information for describing a specific test, the following information has been provided that needs description in the according model:

[17] https://hub.docker.com/u/gkousiou

- Benchmark setup, i.e. a collection of tests that have been submitted for execution in PEF. The setup includes several tests that shares part of the configuration but have differences e.g. in the number of clients used in each iteration. A benchmark configuration will include information on the
- Benchmark iteration, i.e. a given test that has a specific result
- A workload description, that includes details on the function type being called, the load generator and the test type (e.g. hot/cold/warm).
- Function type, being one of the available FunctionBench tests.

For the integration with FunctionBench functions, this needs to be made available to the respective generic Jmeter clients as a parametric endpoint. To this end, the jmx files have been altered in order to receive the name of the function upon invocation, through a Jmeter property or variable. Furthermore the respective user-define variables in the jmx configuration have been changed in order to map to these dynamically set parameters during initialization.

3.7 Global Continuum Placement

3.7.1 Overview

The Global Continuum Placement component performs the higher level selection of the computing continuum clusters to perform the application execution. Since each application is expressed as a workflow composed of tasks; the component enables the deployment of the workflow by performing the placement of each task on one of the available clusters.
The selection is done by considering the applications' needs in resources, as described during the application design, in conjunction with the individual resources availability and possible optimization insights.
Once the placement of all the tasks has been fulfilled, the actual selection of individual resources per cluster and the execution of each task is done by the local cluster scheduler which in PHYSICS architecture will be performed by the combination of Openwhisk and Kubernetes.

3.7.2 Technology architecture

The Global Continuum Placement Component architecture has been provided in Deliverable D4.1 but it is provided also here as well to facilitate comprehension.

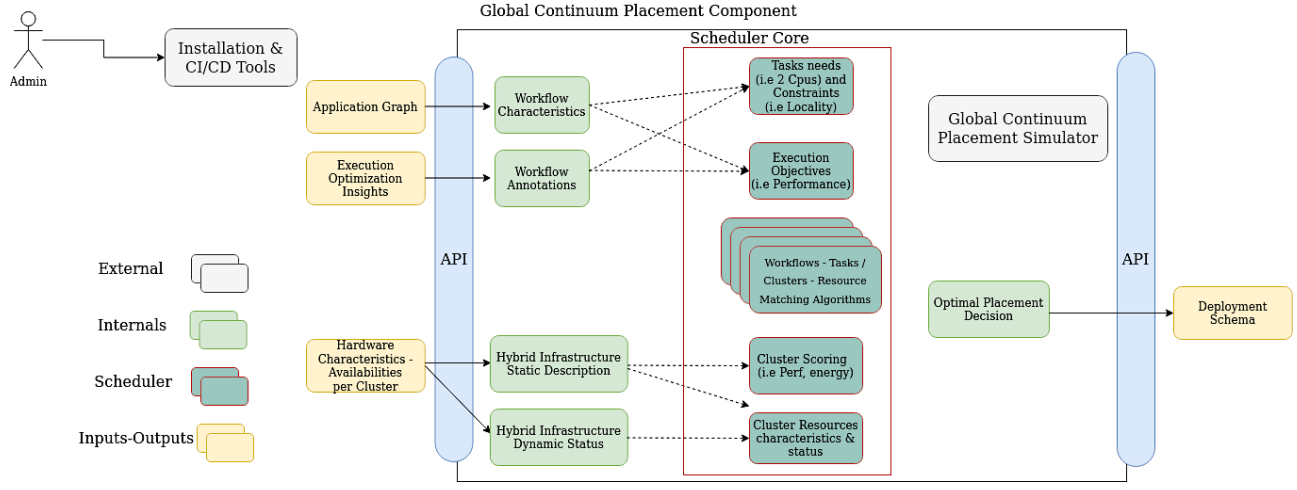


Figure 23 - Global Continuum placement component

As shown in the Figure 23 the Global Continuum Placement is composed from the following parts:

- A subcomponent that will consume inputs such as the Workflows' Characteristics and Annotations along with the Clusters' Static and Dynamic Status
- The scheduler core which performs the matching of tasks with resources based on different available algorithms
- The output subcomponent which will pushes the scheduling decision.

In parallel, two additional external subcomponents are provided:

- The simulator which will allow us to perform fast experimentation and evaluation of the different scheduling algorithms and
- The tools for installation and Continuous Integration / Continuous Deployment

Based on the current architecture of PHYSICS the Global Continuum Placement component does not have to take the final decision of the exact resources where the execution of a task happens. This is done by the local cluster Scheduling Algorithms which lies within the combination of Openwhisk-Kubernetes schedulers. Furthermore, it does not communicate directly with the local schedulers to propose the deployment schema. It forwards the deployment schema to the Orchestrator which will then communicate with each local-cluster scheduler.

3.7.3 Interfaces/API

The communication with other software components is performed through a REST-API. The current version of the Global Continuum Placement provides the implementation of the following API calls. The interfaces will evolve as the component matures.
Table 20 shows the API operations for scheduler:

Table 20 - Global Continuum API for the scheduler

Method	Path/URI	Description
POST	/api/v1/init	Initialise scheduler. Provide platform and workload to the scheduler
POST	/api/v1/schedule	Schedule the workload. Run the scheduler on the given workload. Returns placement mapping for each allocatable task.

The Table 21 shows the API operations for monitoring:

Table 21 - Global Continuum API for monitoring

Method	Path/URI	Description
GET	/api/v1/healthz	Information about monitoring. Check service status.

3.7.4 Distribution, deployment and configuration

The implementation of the Global Continuum Placement component is done using Python 3.7 programming language. The code can be found in GitLab[18]. We make use of NIX functional package manager to prepare the packaged containerized environment to perform the necessary CI/CD pipelines which are currently configured using Gitlab pipelines. Various tests are also implemented

[18] https://gitlab.com/ryax-tech/research/global-continuum-placement/-/tree/main

and incorporated in the CI/CD pipelines with a current coverage of 67%. The whole software can be packaged and installed as a container.

For the initial setup users can use the following commands:

```
git clone https://gitlab.com/ryax-tech/research/global-continuum-placement
poetry install
poetry shell
./main.py
```

Or make use of the container image directly:

```
docker run -ti -p 8080:8080 ryaxtech/global-continuum-placement:c85fa0ac
```

A complete example of usage has been added in D4.1 and some examples can be also found in the repository where the source code is available.

3.8 Distributed In-Memory Service

3.8.1 Overview

The Distributed In-Memory Service (DMS) component has been developed under task T4.4 – Distributed In-Memory State Services for Data Interplay. This component allows sharing large objects between functions. The DMS is based on Pocket [3] [4], an open source prototype data sharing developed at Stanford University. Pocket was developed for sharing data across Lambda functions [19] Pocket provides a Python interface for writing (put) and reading (get) data to/from the Pocket Storage Servers. The DMS allows sharing large objects among functions of a sequence deployed in OpenWhisk[20], within the PHYSICS FaaS platform.

3.8.2 Technology architecture

The architecture description of the DMS can be found in deliverable D4.1 Cloud Platform Services for a Global Space-Time Continuum Interplay Scientific Report and Prototype Description V1. The architecture consists of three main components (Figure 24): the *Controller*, one or several *Metadata Servers* and one or more *Storage Servers*. The *Storage Servers* components are the ones keeping the state. The *Controller* component is in charge of monitoring the status of the system and assigning *Metadata Servers* to allocate the required memory resources. The *Metadata Server* and the *Storage Servers* send status metrics to the C*ontroller* which decides to deploy or remove instances of these components to bear the load of the system. The *Metadata Server* redirects client requests to the corresponding *Storage Server*, checks the storage capacity of all *Storage Servers* and send this information to the C*ontroller.*

[19] https://aws.amazon.com/lambda/?nc1=h_ls
[20] https://openwhisk.apache.org/

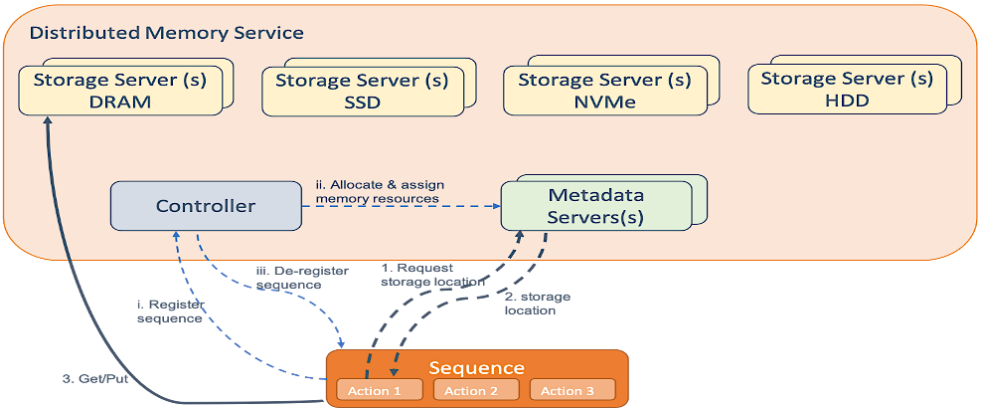


Figure 24 - DMS component

Data shared between functions can be persistent, which means that the data remains in the DMS even if the job has been deregistered, or non-persistent, the data is removed from the DMS as soon as the job is deregistered.

3.8.3 Interfaces/API

The available methods of the DMS are:

- Job register

Receives a job name and optional hints (e.g., requirements regarding the type of storage) as input and returns a job identifier string as output.

Table 22 - DMS-API-job register

Method	**register_job(jobname, hints=none)**
Input	jobname hints
Output	jobid
Error response	"Error registering job"

- Job Deregister

Receives a job identifier (jobid) and notifies the controller that the job has finished. The non-persistent data of the job is deleted.

Table 23 - DMS-API-job Deregister

Method	**deregister_job(jobid)**
Input	jobid
Output	"Successfully deregistered jobid"
Error response	"Error deregistering job"

- Connect

Receives the metadata server address (IP:PORT), opens a connection with the metadata server and returns a pocketHandle object to be used in the rest of the methods.

Table 24 - DMS-API-Connect

Method	**connect(metadata_server_address)**
Input	metadata_server_address

Output	pocketHandle
Error response	“Connecing to metadata server failed!”

- Close

Closes a connection with the metadata server.

Table 25 - DMS-API-close

Method	**close(pocketHandle)**
Input	pocketHandle
Output	
Error response	

- Count files

Receives the pocketHandle, directory name (dir) and job identifier (jobid) as input, shows the number of files in the directory and returns 0 if the process has finished successfully, returns -1 otherwise.

Table 26 - DMS-API-count files

Method	**count_files(pocketHandle, dir, jobid)**
Input	pocketHandle dir jobid
Output	<files number>
Error response	-1

- Lookup

Receives a pocketHandle, job identifier (jobid) and object_name (directory name or file name) as input and returns 0 if the object_name exists, returns -1 otherwise.

Table 27 - DMS-API-lookup

Method	**lookup(pocketHandle , jobid, object_name)**
Input	pocketHandle jobname object_name
Output	0
Error response	-1

- Create directory

Receives a pocketHandle, directory name (dir) and job identifier (jobid) as input, creates a directory under jobid directory and returns 0 if the creation was successful, returns -1 otherwise.

Table 28 - DMS-API-create directory

Method	**create_dir(pocketHandle , dir, jobid)**
Input	pocketHandle dir jobid
Output	0
Error response	-1

- Delete directory

Receives a pocketHandle, directory name (dir) and job identifier (jobid) as input, deletes the directory under jobid directory and returns 0 if the directory was deleted, -1 otherwise.

Table 29 - DMS-API-delete directory

Method	**delete_directory(pocketHandle , dir, jobid)**
Input	pocketHandle dir jobid
Output	0
Error response	-1

- Put

Receives a pocketHandle, a path to the file with the data to be stored (src_filename), a path to the file where the data is going to be stored in the Storage Server layer (file_name), job identifier (jobid) and optionally a PERSIST flag as input. This PERSIST flag default value is set to false, this means that when the job is deregistered the file will remain in the system. It returns 0 if the data has been stored, returns -1 otherwise.

Table 30 - DMS-API-put

Method	**put(pocketHandle, src_filename, file_name, jobid, PERSIST=false)**
Input	pocketHandle src_filename object_name jobid PERSIST
Output	0
Error response	-1

- Get

Receives a pocketHandle, a path to the file that contains the data to be read inside the Storage Service layer (src_filename), a path to the file where the data is going to be stored once it is been read (file_name), the job identifier (jobid) and optionally a DELETE flag as input. The DELETE flag default value is false, which means that when the data is read the file will remain in the system. It returns 0 if the data has been read successfully, returns -1 otherwise.

Table 31 - DMS-API-get

Method	**get(pocketHandle, src_filename, file_name, jobid, DELETE=false)**
Input	pocketHandle src_filename object_name jobid DELETE
Output	0
Error response	-1

- Put buffer

Receives the pocketHandle, the path to the variable with the data to be stored (src), the size of the data to be stored (len), the path to the file where the data is going to be stored in the Storage Server layer (file_name), the job identifier (jobid) and optionally a PERSIST flag as input. This PERSIST flag default values is false, which means that when the job is deregistered the file will remain in the system. It returns 0 if the data was stored, returns-1 otherwise.

Table 32 - DMS-API-put buffer

Method	**put_buffer(pocketHandle, src, len, object_name, jobid, PERSIST=false)**
Input	pocketHandle file_filename len object_name jobid PERSIST
Output	0
Error response	-1

- Get buffer

Receives a pocketHandle, a path to the file that contains the data to be read inside the Storage Service layer (src), the variable where the data is going to be stored once it is read (object_name), the size of the data to be read (len), the job identifier (jobid) and optionally a DELETE flag as input. The DELETE flag is set to false by default; this means that when the data is read the file will remain in the system. It returns 0 if the data was read, -1 otherwise.

Table 33 - DMS-API-get buffer

Method	**get_buffer(pocketHandle, src, object_name, len, jobid, DELETE=false)**
Input	pocketHandle src object_name len jobid DELETE
Output	0
Error response	-1

3.8.4 Distribution, deployment and configuration

The DMS component distribution is available in the PHYSICS git repository21. Each sub-component is contenarized in images that are created by the DMS-BUILD Jenkins pipeline. Those images are stored in the Harbor image registry of the PHYSIC platform. Once the images have been created, the DMS-DEPLOY Jenkins pipeline is executed. The DMS-DEPLOY Jenkins pipeline deploys the DMS components in the PHYSICS AWS infrastructure. This CI/CD pipeline is executed each time a new version of the DMS code is uploaded into the PHYSICS git repository.
The parametrization of the resources of the pods of the different components will be added as parameters in the DMS-DEPLOY Jenkins pipeline. The parameters allow modification of the memory and hugepages resources limits and requests.

3.9 Adaptive Platform Deployment, Operation & Orchestration

3.9.1 Overview

The Orchestrator component will manage the deployment of the application components in the infrastructure offering available to the PHYSICS platform chosen by the Inference Engine (Reasoning Framework) and the Global Continuum Placement (Optimizer) components. This component will also execute the runtime adaptations needed to enforce the QoS associated with the application components by the user owner of the application.

3.9.2 Technology architecture

To facilitate the integration with the components of WP5 the Orchestrator needs to implement the Open Cluster Management (OCM) interface. To allow this the Orchestrator will parses a JSON file with the application graph (created by the Inference Engine component) and translate it to a YAML file with the schema defined by the ManifestWork CRD (Custom Resource Definition) component of the OCM. This ManifestWork has to be deployed in the Hub or master Kubernetes cluster (part of OCM configuration) to instruct to the target cluster infrastructure which kind of resources need to be created in the target Spoke or managed Kubernetes cluster.

This process is only half of required deployment of the application components in the target cluster chosen by the Optimizer from the candidate list generated by the Inference Engine. In the target managed cluster or Spoke we need to implement a Kubernetes Operator should be implemented to interpret the specific CRD created in PHYSICS and describes the workflow of functions that make up the application (also called Workflow CRD). This new Workflow CRD will be embedded in the ManifestWork CRD inside the OCM interface. The operator implements the specific interface to the OpenWhisk (OW) FaaS platform, pre-installed in the managed cluster, and register the functions and sequences/flows in the local OW.

Figure 25 shows the flow of actions to be executed when deploying an application by the Orchestrator:

[21] https://gogs.apps.ocphub.physics-faas.eu/WP4/DMS.

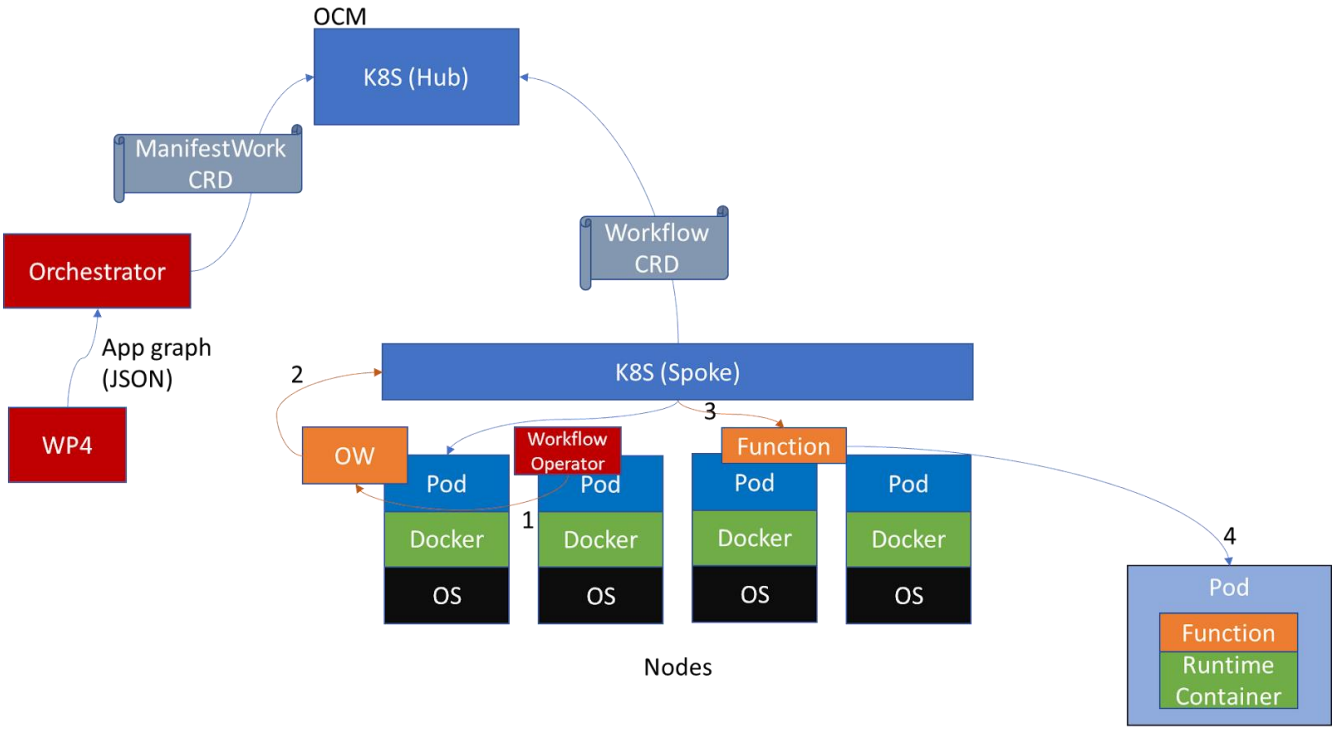


Figure 25 - Orchestrator flow

By now, the implementation only involves the OW interface, however, further integrations with other FaaS platforms like KNative are foreseen in this custom-made Operator.

3.9.3 Interfaces/API

The available endpoints were described in the deliverable D4.1 (section 6.2.3 Interfaces and integrations) and it is repeated here for convenience to facilitate the comprehension of the component functionalities. Many of these endpoints will be included in the Kubernetes Operator for the Workflow CRD.
The following table shows the API operations for functions:

Table 34 - Orchestrator component functions API

Method	Path/URI	Description
GET	/api/v1/function/{id}	Get information details of a function that exists in the function catalogue.
POST	/api/v1/function	Create or register a function in the functions catalogue.
PUT/PATCH	/api/v1/function/{id}	Update a function in the functions catalogue.
DELETE	/api/v1/function/{id}	Delete a function from the functions catalogue.
GET	/api/v1/functions	List all the functions from the catalogue.

API operations for flows/sequences of functions (applications):

Table 35 - Orchestrator component flows (application) API

Method	Path/URI	Description
GET	/api/v1/flow/{id}	Get information details of a flow or sequence of functions from the catalogue.
POST	/api/v1/flow	Create or register a flow or sequence in the catalogue.
PUT/PATCH	/api/v1/flow/{id}	Update a flow or sequence in the catalogue.

DELETE	/api/v1/flow/{id}	Delete a flow or sequence in the catalogue.
GET	/api/v1/flows	List all the flows or sequences defined in the catalogue.

API operations for invocation of runtime actions (RPC API call style):

Table 36 - Orchestrator component runtime RPC API

Method	Path/URI	Description
POST	/api/v1/flows/deploy	Deploy/install a flow or sequence in a target FaaS engine/platform. This operation will also install the functions of the flow.
POST	/api/v1/flows/undeploy	Uninstall a flow or sequence from the target FaaS engine/platform. This operation will also uninstall the functions of the flow.
POST	/api/v1/flows/run/{id}	Execute a flow or sequence in the target FaaS engine/platform.
POST	/api/v1/functions/deploy	Install a standalone function in the target FaaS engine/platform.
POST	/api/v1/functions/undeploy	Uninstall a function from the target FaaS engine/platform.
POST	/api/v1/functions/run	Execute a standalone function in the target FaaS engine/platform.

3.9.4 Distribution, deployment and configuration

All the modules of this component are implemented in Go language and packetized in container images as part of the CI/CD pipeline implemented with Jenkins. The source code will be in the official repository of the PHYSICS project[22] under an open-source licence (Apache 2.0). The container images will be stored in the image repository of PHYSICS in Harbor.

For the parametrization of the container images installation, we will use kustomizer and specific YAML config files available in the source code repository.

For the development of the API interfaces, we will use OpenAPI Generator tools[23]. For the development of the Kubernetes Operator, we will use Operator SDK tools[24].

For the deployment destination (testbed) we will use Openshift as CaaS (Container as a Service) installed in the PHYSICS testbed hosted by Amazon AWS. The CD pipeline will manage the deployment of the components in the assigned namespaces of the cluster.

3.10 Scheduling Algorithms

3.10.1 Overview

This component is responsible for 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 the tasks of a broader FaaS application workflow upon the underlying compute infrastructure of a single cluster. This scheduling phase takes place after the global continuum placement has selected the most adapted cluster to execute each task.

[22] https://gogs.apps.ocphub.physics-faas.eu/WP4/orchestrator
[23] (https://openapi-generator.tech/).
[24] (https://sdk.operatorframework.io/).

In PHYSICS architecture, the scheduling algorithms are implemented as Openwhisk-Kubernetes scheduling mechanisms and are responsible for actual execution of the tasks on the selected underlying computing resources of the clusters.

3.10.2 Technology architecture

Based on the analysis provided in D5.1 and our initial focus to provide a scheduling algorithm that will minimise the time a task is executed, by addressing the delays related to the downloading of containers and their layers; We proposed our first algorithm to be a "container layer aware" scheduler for Kubernetes. There is already an image locality plugin in the Kubernetes scheduler but it does not take into account layers in the image.
This plugin is located in pkg/scheduler/framework/plugins/imagelocality in the current Kubernetes codebase (02/2022).
The idea here is to modify the *imagelocality* plugin to:

- Get the layers available with their size (if possible) on each of the nodes at startup
- For each new pod compute a score regarding the cumulative size of already present layers.

The CRI(Container Runtime Interface) manifest contains the layers with their sizes and it is available at pull time.
See the OCI(Open Container Initiative) reference about manifest at Git server[25].
The idea is to get the layers' info from the manifest CRI and put it in the *ImageSpec.annotations* field in the CRI API see Git repository[26] .

To do so, we have to implement a change of the internal Kubernetes interfaces to add the Layers information into Kubernetes core.v1 protocol. This allows the layer information from the CRI interface to be propagated to the internal *NodeStatus.Images* interface that is already accessible by the scheduler.
To do so, we have to implement tour solution in this branch of our Kubernetes fork[27]:
Besides the changes taking place within Kubernetes we have to modify the actual CRI. For that purpose we choose to adapt the CRI-O(Open Container Initiative) as the CRI of reference because it is the one used in our PHYSICS testbed and it is used as the default CRI by OpenShift.
The layer information must come from the image pulled to the CRI interface. However this is not possible using the default version of CRI-O. For that, we need to find a way to get information from the manifest which contains layers digest and size to be added in CRI v1 protocol ImageSpec Annotation map with a prefix (See kubernetes implementation).
To do that we have prototyped our solution in this branch of our CRI-O fork: based on this fork CRI-O is now giving layers size in the CRI protocol Image annotations.

3.10.3 Interfaces/API

The API used for the new scheduling algorithm is practically the API of Kubernetes scheduler since we comply with the standards provided by its API. This API can be found online here[28]
Nevertheless, to support the changes mentioned above certain modifications had to be made in order to enhance the default API with the container layer details. For that we had to modify the

[25] https://github.com/opencontainers/image-spec/blob/main/manifest.md
[26] https://github.com/kubernetes/cri-api/blob/master/pkg/apis/runtime/v1/api.proto#L673
[27] https://github.com/RyaxTech/kubernetes/tree/image-layer-locallity-scheduler
[28] https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.23/

ContainerImage API from the default version here[29] to also contain a map with the layers ID and layer size.
This will allow us to GET these new details related to the container layers and perform the right scheduling decision taking into account these details as well.

3.10.4 Distribution, deployment and configuration

As mentioned previously the code of the new "Container layers aware" Scheduling algorithm component is actually implemented within different open-source projects.
Changes are needed within the CRI and in particular we have focused on CRI-O as it is the default CRI of Openshift. The code can be found in this CRI-O fork[30]:
To configure and deploy this particular version of CRI users can follow the next steps:

1. Pull the cri-o fork with:

```
git pull https://github.com/RyaxTech/cri-o
git checkout image-layer-locality-scheduler
cd cri-o
```

2. Build cri-o with Nix:

```
nix build -f nix
```

3. Start Cri-O:

```
sudo --preserve-env=PATH ./result/bin/crio --log-dir /tmp/cri-o/logs --root /tmp/cri-o/root --
log-level debug --signature-policy test/policy.json
```

4. Pull an image and query CRI-O through the CRI API:

```
sudo crictl --runtime-endpoint unix:///var/run/crio/crio.sock pull
docker.io/library/debian:latest
sudo crictl --runtime-endpoint unix:///var/run/crio/crio.sock pull
docker.io/library/python:latest
sudo crictl --runtime-endpoint unix:///var/run/crio/crio.sock images -o json
```

Following that we can see that the annotation map contains one common key because the python image is based on debian. The common layer is:

```
"imageLayer.sha256:0c6b8ff8c37e92eb1ca65ed8917e818927d5bf318b6f18896049b5d9afc28343": "54917164"
```

On the other side changes have been made within Kubernetes to expose layer information to the scheduler.
We are currently finalizing the implementation of the "container layer aware" scheduler to use layers information of already scheduled images. This represents an extension of the existing Kubernetes ImageLocality plugin in order to compute the new score based on layers. The new plugin will be part of the above Kubernetes fork.

3.11 Resource Management Controllers

3.11.1 Overview

This component 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 other PHYSICS components, so that they can better control both the infrastructure itself and

[29] https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.23/#containerimage-v1-core
[30] https://github.com/RyaxTech/cri-o/tree/image-layer-locality-scheduler

the applications running on top. This means offering the functionality and APIs to be able to perform wiser scheduling and co-allocation decisions, as well as enabling applications deployments across different clusters.
As a result, this component also focuses on adding the missing functionality at the infrastructure layer to be able to support the PHYSICS architecture. This component is implemented as extensions to the existing upstream projects, when feasible or as new components that plug into the kubernetes ecosystem.

3.11.2 Technology architecture

At the current state of the project, we developed 2 sub components to support the resource management controllers:

- Scheduler selection webhook[31] - This sub component uses the kubernetes webhook extension mechanism to control the scheduler that is used for scheduling each pod. Pods can be labelled to use specific PHYSICS schedulers that use PHYSICS specific scheduling logic (for example, additional affinity/anti affinity rules or green energy pods)
- The Workflow Custom Resource Definition (CRD) - CRD is one of the native kubernetes mechanisms for extending its functionality. Specifically, the WF CRD is used to keep all the required information about the Workflows, both data (instance data) and metadata (definitions) in one central place where it can be used by any of the PHYSICS components. This component is in a final design mode and is not fully implemented yet.

The relationship between the components in the context of the PHYSICS platform can be seen in the following diagram (Figure 26):

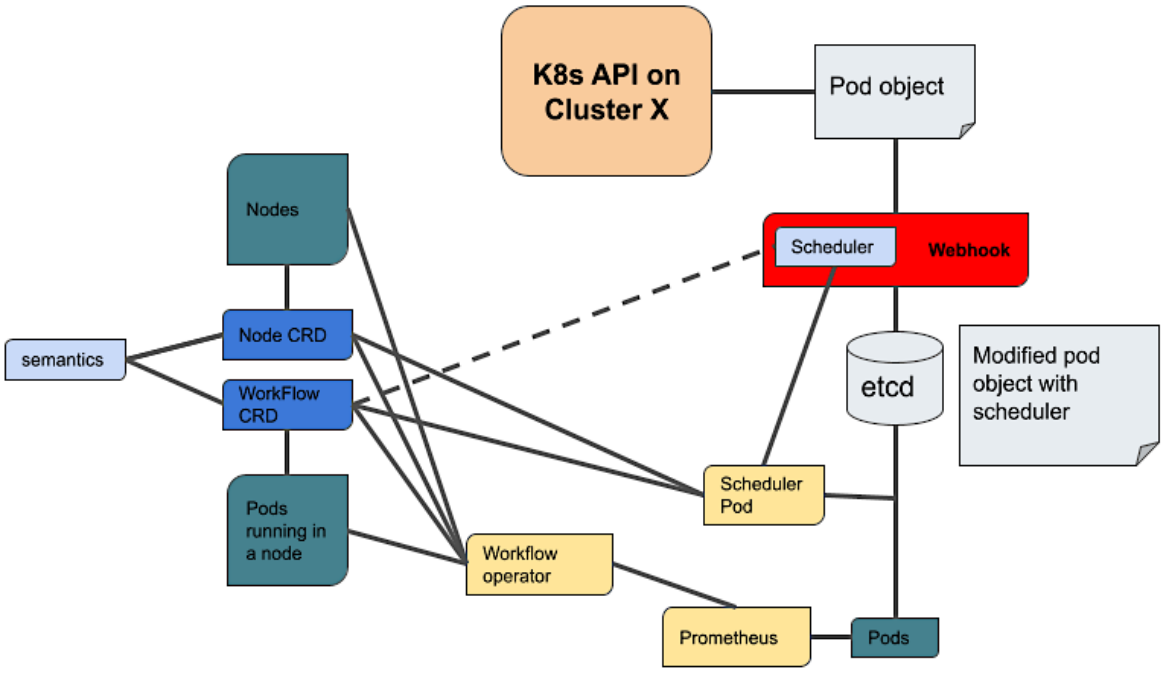


Figure 26 - PHYSICS context

In addition we contributed to several other upstream components in order to make them work better in the cloud compute continuum and especially in edge nodes. The components we contributed to are:

[31] https://github.com/luis5tb/physics-webhook

- Microshift[32] - A small form factor Openshift that is optimised for deployment on edge nodes.
- Submariner[33] - This is a tool for connecting kubernetes clusters. We pushed some bug reports in order to make it work better with Microshift and ovn-kubernetes CNI.

Figure 27 shows how Submariner and Microshift (uShift) are used in the context of the PHYSICS platform architecture:

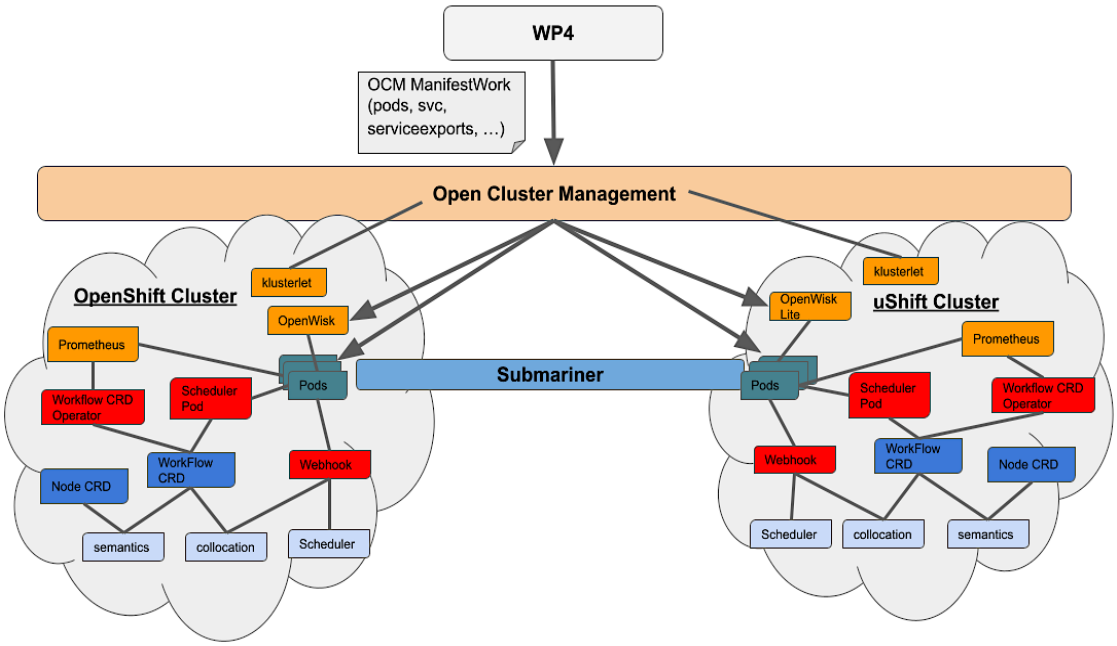


Figure 27 - Submarine and Microshift

3.11.3 Interfaces/API

Webhook

The Webhook component is a static component and the interface with it is declarative according to kubernetes guidelines. It is configured by yaml files which are deployed into the cluster, and pods are registered to a specific scheduler by adding annotations to them. This process has two steps which together implement the interface to the webhook (after it was deployed):

1. Add a label into the kubernetes namespaces where the webhook needs to be used. The value of this label indicates which scheduler to use. By default "physics-webhook: enabled"
2. In order to direct kubernetes to use any of the PHYSICS schedulers on a running pod, we need to add annotation with a specific label (by default "physics-scheduler") and the name of the scheduler to be used for that pod. Then the webhook will be invoked on pod creation and modify the pod so that the scheduler to be used is the one stated in the annotations (if the namespace where the pod was created has the label stated at step 1).

[32] https://github.com/redhat-et/microshift
[33] https://github.com/submariner-io/submariner

Workflow CRD

The Workflow CRD is managed by a kubernetes Operator[34]. This is in charge of managing the events once CRDs of type Workflow are created/deleted.

As stated before, the usage of CRDs allows extending the Kubernetes API with custom resources, and Workflow CRD is a resource of this type. Therefore the same API as for any Kubernetes object applies. The physics components will make use of the Kubernetes API on this CRD for:

1. Creating objects of type Workflow CRD: This is the same as creating any other Kubernetes resource (pod, services, configmaps, etc.) The different options available in that call depends on the fields defined for the workflow CRD, and how internally the operator manages it. Similarly to when a pod is created with one option or another.
2. Deleting objects of type Workflow CRD: Same as before, it is just a call to Kubernetes API to remove one object. The operator will be in charge of triggering all the needed actions in reaction to that.
3. Other operations on the Workflow CRD (such as read or update) will be performed by the operator and use the kubernetes API - these operations are internal and are not exposed to the users directly.

The kubernetes API is a declarative one, it is based on creating an object (data object) and sending it to the API. It is accessible in multiple forms. From the command line it can be invoked via the kubectl command, for example:

```
kubectl apply -f <yaml file>
kubectl delete -f <yamlfile>
kubectl get {resource type}
```

The same paradigm exists in many programming languages when you create a data structure and apply it via different APIs. The Workflow CRD operator is written in the go programming language and will use the kubernetes API implementation in go: https://github.com/kubernetes/client-go

3.11.4 Distribution, deployment and configuration

The Webhook mechanism is composed of several yaml files that should be deployed on the kubernetes cluster[35]. There are configurable options on the webhook_server to decide what annotation should be searched on the pods to decide on the scheduler. This should be configured in the yaml files before applying them. Another configurable option is to change the namespaceSelector (label) that you would like to use to tag the namespaces where the webhook will be enforced.

Along with this a python script parses the pod annotations and changes the pod scheduler accordingly.

The Workflow CRD will be implemented as a Kubernetes Operator, therefore it will also be distributed as a set of yamls to deploy on the kubernetes cluster. Another option may be to integrate it with OLM, and then being able to install it from the OKD UI. The configurable options will be implemented in the operator, and will be accessible by other components by generating a CRD with one or other options, i.e. filling in some properties or another - in a similar way as any other Kubernetes object.

[34] sdk.operatorframework.io

[35] Instructions on the README file at https://github.com/luis5tb/physics-webhook

3.12 Co-Allocation Strategies

3.12.1 Overview

The Co-allocation strategies component was designed and implemented under the scope of task T5.4 Optimised service co-allocation strategies. The description of this component was reported in deliverable D5.1 Extended Infrastructure Services with Adaptable Algorithms Scientific Report and Prototype Description V1. This component is triggered whenever a new pod is going to be deployed in the infrastructure. The co-allocation component analyses the cluster status (pods running at each node and node resources), the requirements of the function to be deployed (defined in the Workflow) and produces a set of affinity and anti-affinity rules to be used by the Kubernetes scheduler to find out the most suitable node for the function deployment. The Workflow is a Kubernetes Custom Resource Definition (CRD) object that contains information about the functions that belong to a given workflow (sequence), the needs for each specific function and the relation with the functions in the workflow, among other information.

3.12.2 Technology architecture

The Co-allocation strategies component is integrated with the mutating webhook component developed in task T5.3 Resource Management Controllers and Interfaces. The mutating webhook detects when a new pod object is going to be deployed in a Kubernetes cluster. It first invokes the Scheduler component, developed under task T5.2 Adaptable Provider Level Scheduling Algorithms, and then invokes the Co-Allocation strategies component (Figure 28).

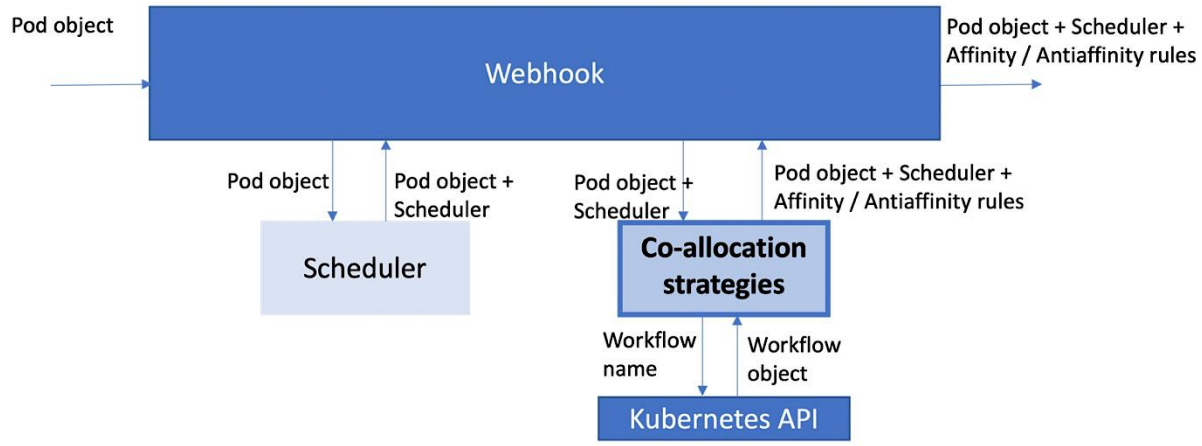


Figure 28 - Co-allocation invocation Technology architecture

The co-allocation component is made out of three sub-components: the Workflow information collector, the Co-allocation algorithm and the Affinity/Anti-affinity rules injector (Figure 29). The Workflow information collector receives the pod object with the resources needed and the annotations regarding the type of hardware needed with all the required information to be deployed, obtains the workflow object name from the pod object and sends a request through the Kubernetes API to retrieve the workflow object information. This workflow object information keeps the requirements in terms of resources of the function. The Co-allocation algorithm, based on those requirements, and taking into account the cluster status and the possible interferences that this function can cause, produces a set of affinity and anti-affinity rules to be used by the scheduler to place the function pod in a Kubernetes cluster node. Finally, the Affinity/Anti-affinity rules injector sub-component receives the rules produced by the co-allocation algorithm and enhances the pod object adding those rules in the affinity section of the pod object.

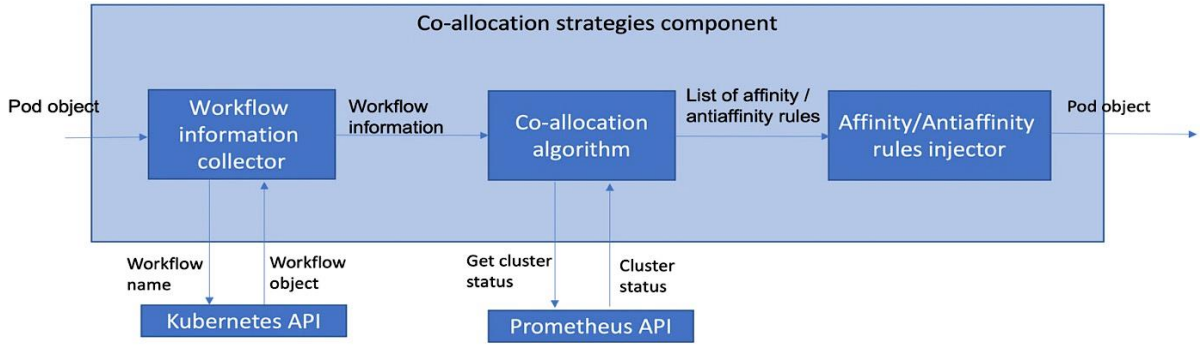


Figure 29 - Co-allocation strategies component internal architecture

3.12.3 Interfaces/API

The co-allocation strategies component has a Python interface with one method.

- Get_affinities()

It receives the pod YAML object to be deployed as input and returns an enhanced version of the pod YAML with affinity and anti-affinity rules.

Table 37 - Co-Allocation/API-get affinities

Method	**get_affinities(pod)**
Input	pod: pod object
Success response	pod: pod object enhanced with the affinity and anti-affinity rules
Error response	- Workflow name not defined in pod object - Missing workflow object - Not enough resources to allocate the pod

3.12.4 Distribution, deployment and configuration

The Co-allocation strategies component is implemented in Python that is part of the mutating webhook logic. The Co-allocation strategies component has been integrated in the PHYSIC CI/CD pipeline, using Jenkins build pipelines. The distribution of the component is available in the PHYSICS git repository at https://gogs.apps.ocphub.physics-faas.eu/WP5/Webhook-CoAllocationStrategies.

The CoAllocationStrategies-BUILD Jenkins pipeline creates the images and pushes the docker image to the PHYSICS Harbor images registry each time a new version of the Co-allocation component is uploaded to the PHYSICS git repository. Next the deployment of the webhook and the co-allocation strategies component will be done by the PHYSICS platform administrator. The CoAllocationStrategies-DEPLOY Jenkins pipeline deploys the webhook and co-allocation logic in the PHYSICS AWS platform.

4. REUSABLE ARTEFACTS MARKETPLACE IMPLEMENTATION

This chapter covers the design and implementation of the first version of the prototype of the Reusable Artefacts MarketPlace (RAMP) application, which was established to provide access to the project solutions emphasising in patterns that can be reused in FaaS applications. Besides that, RAMP will exploit the project results by offering relevant training materials (e.g., videos, webinars) and demonstrating the utilisation of its offerings in real-life use cases.
PHYSICS RAMP is available at URL: *https://marketplace.physics-faas.eu/*

4.1.1 Overview

RAMP implementation relies on WordPress[36], which tops the list of the three most often used site building packages in the world, followed by Joomla and Drupal. WordPress is free to download and use, comes with numerous add-ons for specialised functionality, and can be customised to suit the needs of individual users. Although WordPress was originally designed to support blogging and related online publishing, it also powers a wide range of sites with other purposes. The WordPress package plus a variety of basic and premium plugins can run complex sites for large multinational corporations, manage small businesses, and create personal blogs. WordPress sites can contain full-service eCommerce stores, showcase a portfolio, or host a social network, group, or podcast. Thanks to its many themes and easy access to its source files, WordPress can also facilitate the required adaptability of a project's changing needs.

4.1.2 Technology architecture

The RAMP service has been developing leveraging the WordPress interface which allows for the continuous updates of web site's structure and content. Given that a WordPress site is not a typical application that can follow a usual DevOps pipeline where the developers maintain the application's artefacts in a Git repository, a self-hosted server, operated by INNOV, was opted to accommodate the PHYSICS marketplace. This approach ensures that the RAMP will be operational after the end of the project without requiring any migration (i.e., from PHYSICS AWS to another server). In addition, GFT facilitates the DNS server of the RAMP which is provided by the Amazon Route 53 service (see also Figure 30). Currently, RAMP users are divided into three categories: i) the administrators, ii) the registered users and (iii) the visitors. The administrators (i.e., INNOV, GFT) can modify the structure or add content directly from a web browser by accessing its URL and logging in. On the other hand, visitors have limited access to the provided assets while full access is enabled only to registered users. The latter have full access to the offered resources and can also upload new assets. It's worth mentioning that new assets are added to the site after review from the administrators.

[36] https://wordpress.org

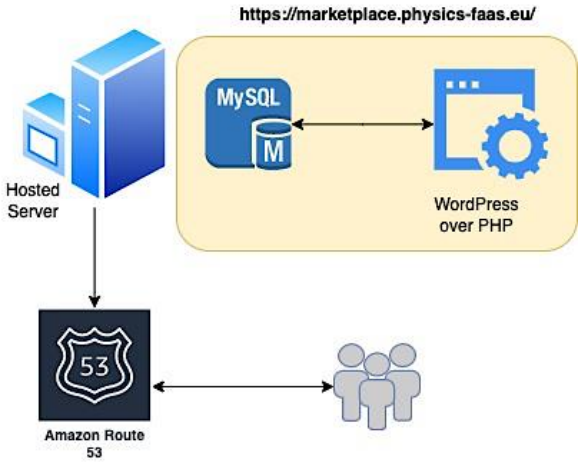


Figure 30 - RAMP Architecture

4.1.3 Artefacts

The initial content to be populated on the RAMP will be based on the outcomes of technical WPs and the project's dissemination activities. Thus, it will include patterns, flow, and functions related to the FaaS domain, services, semantics and interoperability tools, algorithms, artefacts for the optimization of applications' deployment at the edge, and more. In addition, educational resources in the context of functional programming such as tutorials, webinars, and workshops will be included in RAMP. Besides that, end-to-end use cases leveraging the project's offerings will be provided.
Towards monitoring the integration process of this content, there is an indicative document in the project's shared repository summarising the assets (i.e., name, type, description, owner, etc.) that will be provided by the consortium. This document is continuously updated as the project produces more results. However, RAMP enables the direct uploading of an asset through a dedicated form.

4.1.4 Distribution, deployment and configuration

RAMP is deployed in a dedicated cloud-based Hosting and Database Server hosted in Germany. The utilised server follows a shared resources plan which enables autoscaling based on site's requirements. Thus, as RAMP will be populated with more assets, this server may be upgraded to facilitate the respective resource needs. The management of this server is undertaken by INNOV which is also responsible for the front-end design of the RAMP.
As a DNS server responsible for translating the Hosting Server's IP in an interpretable name (i.e. marketplace.physics-faas.eu/), the Amazon's Route 53 web service was chosen. This offers a highly available and scalable cloud Domain Name System (DNS) which has been purchased by GFT.

4.1.5 User Story

The PHYSICS market platform consists of three pages (i.e. main/homepage, assets, and training), and an extra page will be added to show the use cases of the project solutions.
The home page, depicted in Figure 31, introduces the user to the concepts of the project and the RAMP, offering also the register/sign-in option. At the bottom of this page, the latest assets uploaded in the marketplace are illustrated and a direct link to upload new assets is offered to the already registered users.

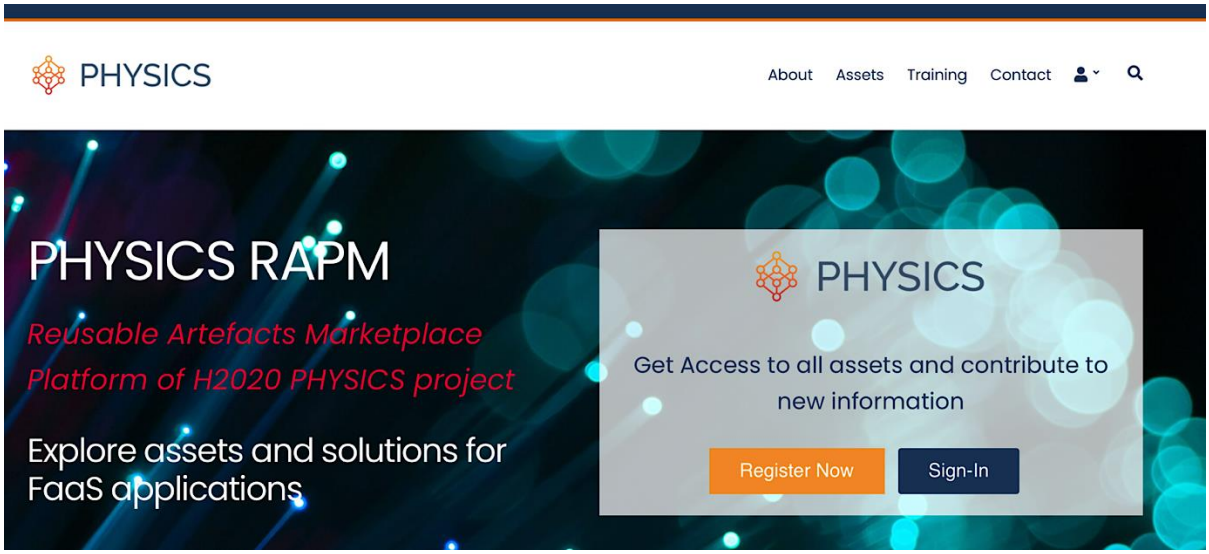


Figure 31 - Homepage

In the Assets page (see Figure 32), the user can explore the provided solutions, tools, artefacts, and services based on their category (i.e., pattern, service, semantics, etc.). Moreover, by clicking on a specific asset, they can access the complete description along with usage instructions of the given asset (see Figure 33). As depicted in Figure 34, the "Add New Asset" tab populates a form for uploading a new asset. In the case of a simple visitor, it redirects to the registration/login form

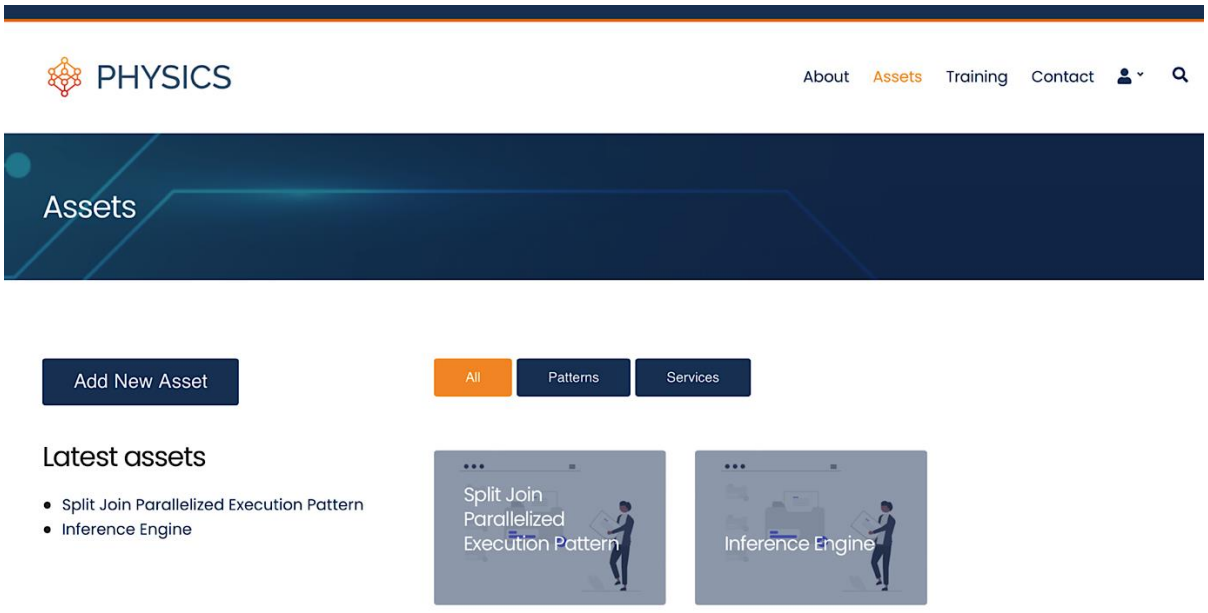


Figure 32 - Assets page

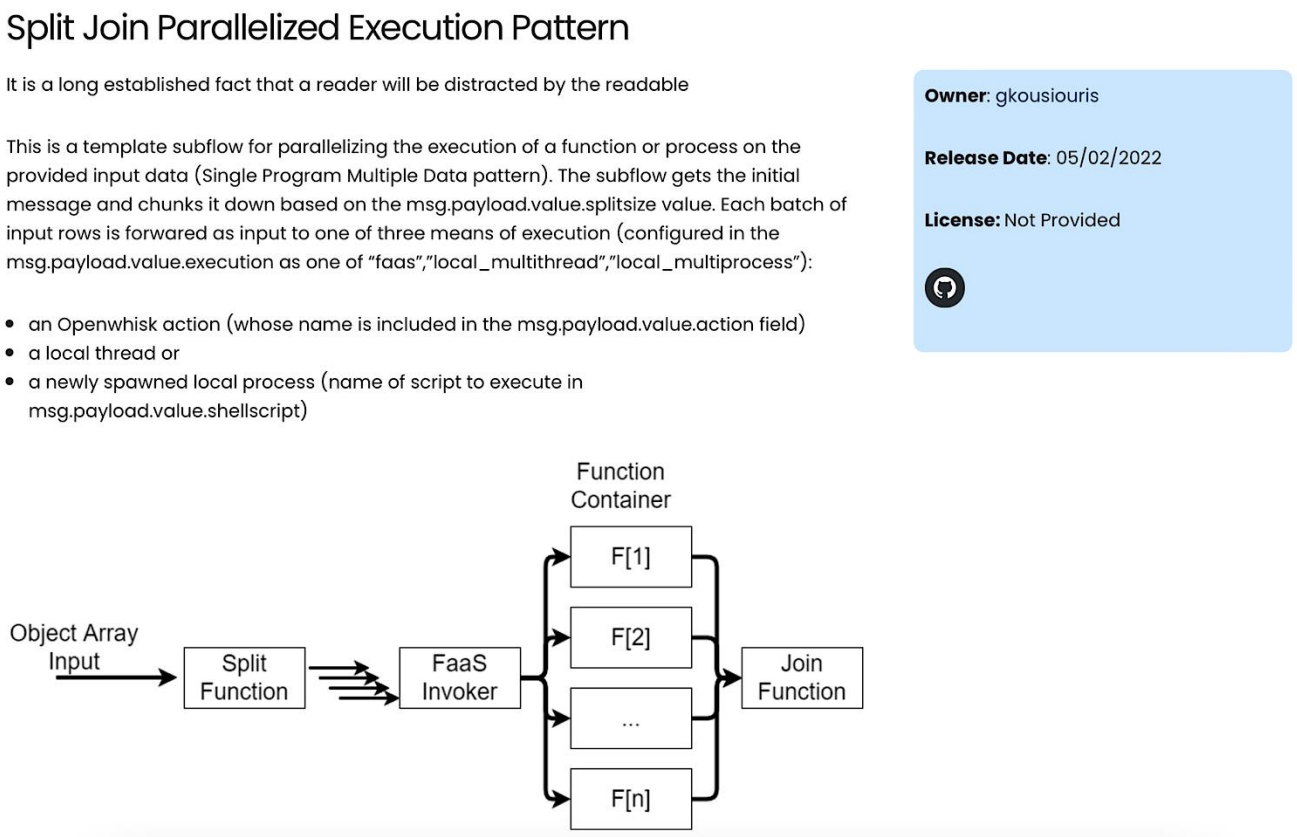


Figure 33 - An asset in RAMP

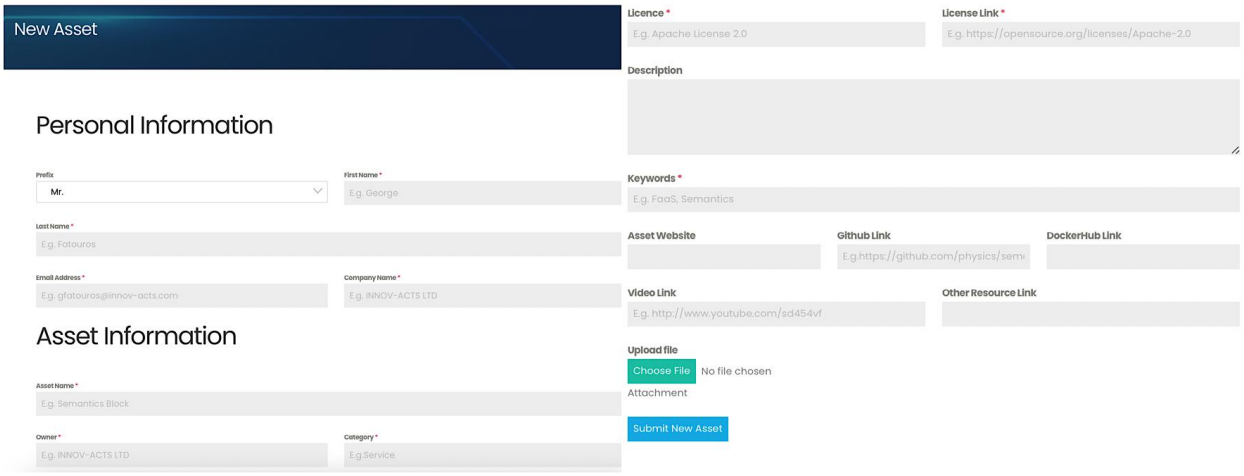


Figure 34 - Form to add a new Asset in RAMP

The Training page will be populated with videos, webinars, workshops, and other relevant resources related to the project activities. Furthermore, one more page demonstrating the use cases of PHYSICS will be added as soon as such results are made available.

5. PHYSICS SOLUTION FRAMEWORK INTEGRATION ENVIRONMENT

This chapter summarises the integrated development and testing environment upon which the PHYSICS solution framework is built, including the Continuous Integration/Continuous Delivery processes put in place to support all the development, testing and integration activities.

5.1 Integration Infrastructure

In order to build and deploy the PHYSICS platform as a whole, as described in D2.4 we envision two different strategies, one for each of the two phases, so respectively:

- Development strategy
- Deployment strategy

The Development strategy defines the collaborative work of the developers' partners to build up the framework, with the goal of creating a Minimum Viable Platform (MVP) of the PHYSICS framework.
The Deployment strategy defines a uniform approach to deploy all the PHYSICS components, in particular about how to deploy them inside a cloud provider or an edge location based on a Kubernetes cluster.
The PHYSICS RA design approach plans to consider a microservices architecture implementation, with services/functions interacting among them through REST APIs based on OpenAPI specification. In that respect, all microservices run in containers on the Kubernetes platform.

5.1.1 Development strategy

The Development strategy provides that developers writing the individual components of the PHYSICS platform need an integrated environment where they can test their components working together with the other services. To support this process, we implemented a continuous integration environment based on the Kubernetes[37] orchestrator provided by OKD[38] and deployed inside the AWS Cloud provider[39]. AWS provides a cost-effective, quickly expandable, ready-to-use environment without having to spend time on procuring resources, at the same time OKD provides many functionalities out-of-the-box not available in Kubernetes Vanilla such as Authorization (OpenID, LDAP), traceability, scaling and management, and much more. Kubernetes is an ideal choice for the development strategy environment since it allows easy updates of deployments when new application images are built, with manifests containing deployment configurations versioned in Git alongside the application source code. Furthermore, it is easy to spin up new test environments from scratch, which enables future scenarios including automated end-to-end integration testing. Build agents themselves are also created on demand and removed when done, providing efficient resource utilisation and clean environments to ensure build reproducibility.
The development strategy has been implemented using the DevOps[40] methodology through the tools (shown in Figure 35) and hosted in a specific namespace named "devops" inside the OKD cluster.

[37] https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/
[38] https://www.okd.io/#what-is-okd
[39] https://aws.amazon.com/what-is-aws/?nc1=h_ls
[40] https://www.gartner.com/smarterwithgartner/the-science-of-devops-decoded#:~:text=Gartner%20defines%20DevOps%20as%20a,between%20operations%20and%20development%20teams.

Figure 35 - DevOps Tools

The DevOps tools are:

- Gitlab[41] is a Git repository manager that lets developer teams collaborate on PHYSICS's source code.
- Jenkins[42] is the de-facto standard open-source automation server for orchestrating CI (Continuous Integration)/CD (Continuous Delivery) workflows. At the same we also planned to evaluate the possible usage of Tekton[43] tools, since it allows to implement pipeline using YAML file
- Harbor[44] is a popular CNCF compliant Docker registry.
- OpenLDAP[45] is used as the single user directory for all tools, centralising authentication and simplifying management of developer accounts.
- Helm[46] is a package manager that streamlines installing and managing Kubernetes applications.

The interaction between these tools and the use by a partners is shown in Figure 36. Starting from point 1, when a developer pushes new component code, Gogs invokes through a webhook a pipeline (also referred as job) configured inside Jenkins. The job builds the component, runs unit tests and, if everything has worked in a proper way, builds an updated Docker image that it pushed it to Harbor. The following step is deploying the updated component in the specific namespace; in fact, we will have as many namespaces as the WPs (Work Package) allowing us to have the correct isolation from an access perspective, so each person inside a specific WP is able to interact with WP namespace while all namespaces are opened to interact with each other. At the end of the process, Jenkins sends a notification to a dedicated CI/CD channel on the PHYSICS Slack[47] workspace, so that developers are informed that a new build occurred and whether it was successful or not. In case of errors, developers will have to inspect the build logs, find the problem and correct it. In case of success, developers will go ahead and test that the new version works correctly in the test environment.

[41] Gitlab (https://about.gitlab.com/solutions/agile-delivery/)
[42] Jenkins (https://www.jenkins.io/doc/)
[43] Tekton (https://tekton.dev/docs/)
[44] Harbor (https://goharbor.io/docs/2.3.0/install-config/)
[45] OpenLDAP (https://www.openldap.org/doc/admin25/)
[46] Helm (https://helm.sh/docs/intro/)
[47] Slack (https://slack.com/intl/en-pt/features)

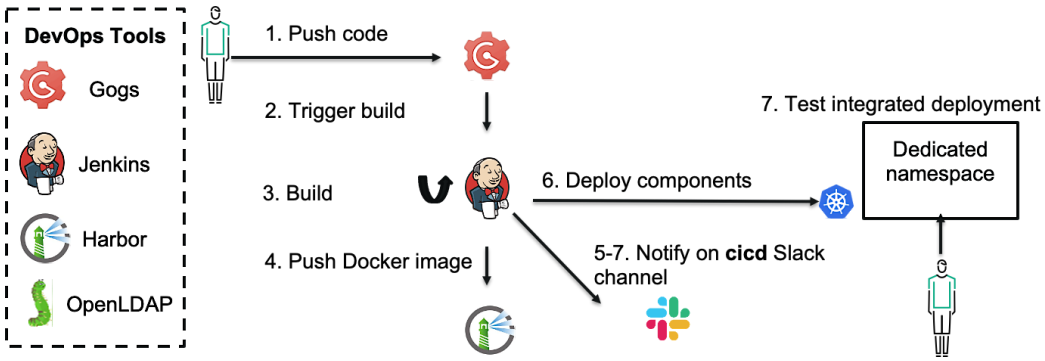
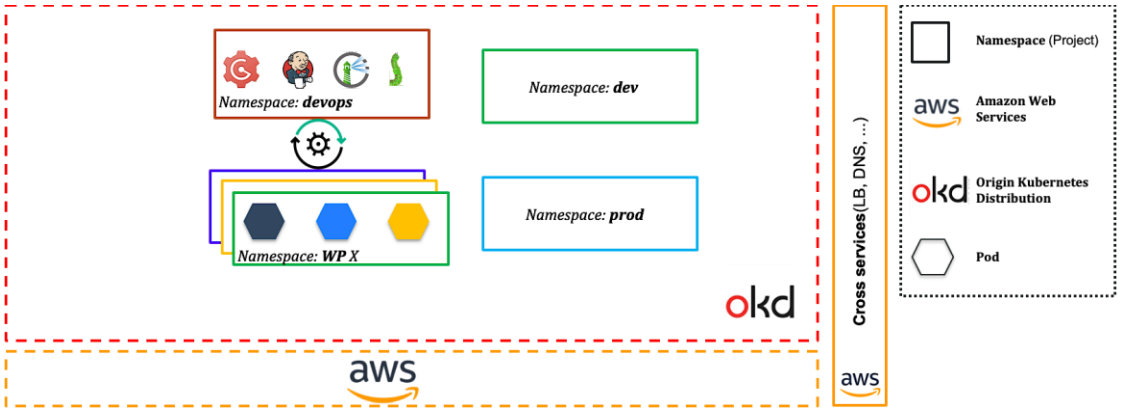


Figure 36 - CI/CD flow

In addition to the “devops” namespace and those dedicated to WPs, it was decided to create two further namespaces, one called "dev" and the second one called "prod".
The “dev” namespace will contain the stable (for example version 1.0) version of PHYSICS components, in this way the developers will be able to continue developing their own components, in their own namespace, without affecting the global functioning of a specific version of the platform versions. The “prod” namespace instead will contain the final version of the PHYSICS components that can be used for a demo even after the project has been completed. The integration environment described above, together with the cross-services provided by AWS for its proper functioning, is presented in Figure 37.

Figure 37 - Integration namespaces



The interaction of developers and partners with the integration environment can take place with two methodologies or through the OKD GUI (Figure 38) or through the use of the oc[48] client (Figure 39).

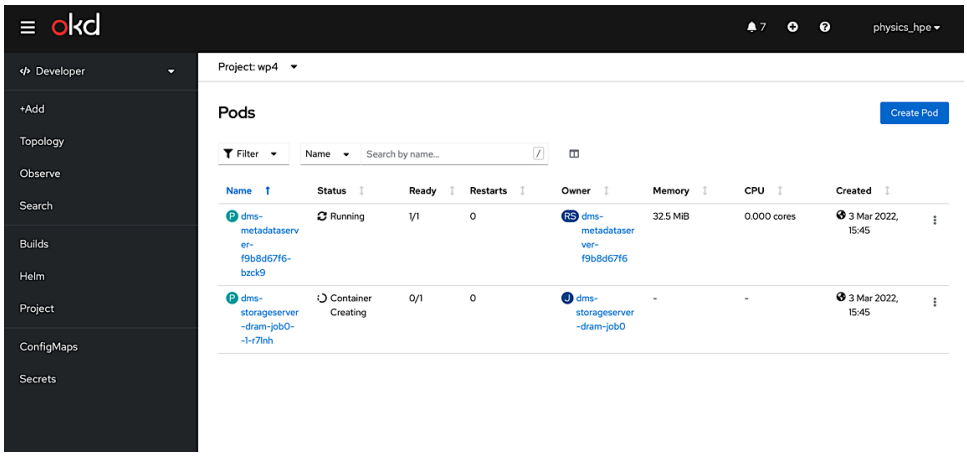


Figure 38 - OKD GUI

```
[root@master PHYSICS]# oc -n wp4 get pod
NAME                                  READY   STATUS              RESTARTS   AGE
dms-metadataserver-f9b8d67f6-bzck9    1/1     Running             0          11d
dms-storageserver-dram-job0--1-r7lnh  0/1     ContainerCreating   0          11d
[root@master PHYSICS]#
```

Figure 39 - OC client

The same segregation implemented into integration environment has been reported within the Gogs structure.
Specifically, 5 “Organisations” have been defined in Gogs as presented in Figure 40.

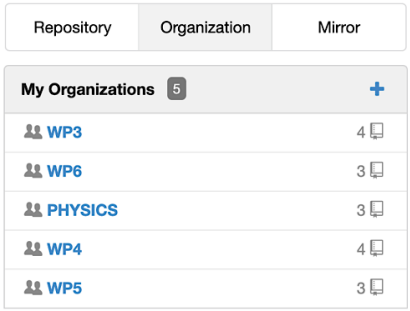


Figure 40 - Organisation inside Gogs

In this way, the developer accesses his organisation, and once inside he can create the repository or the repositories for storing the code of the component to integrate (an example is given in Figure 41)

[48] https://docs.openshift.com/container-platform/4.7/cli_reference/openshift_cli/getting-started-cli.html

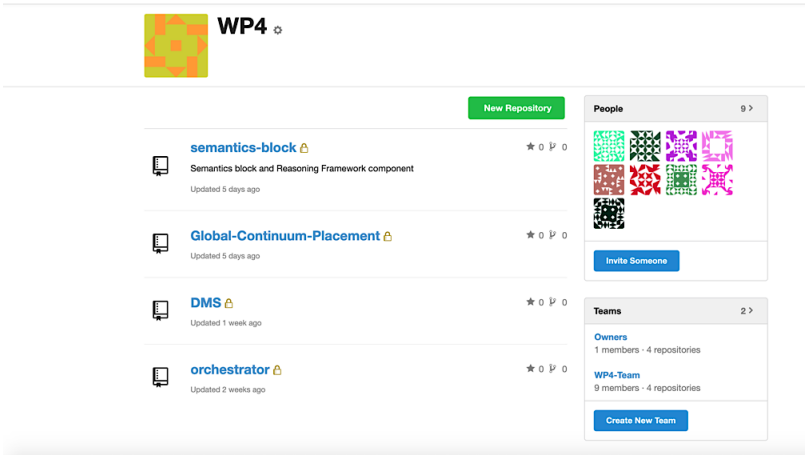


Figure 41 - Repositories inside one organisation

5.1.1 Deployment strategy

The PHYSICS platform is designed to be able to operate from on-premises, to the cloud and up to the edge.
The PHYSICS platform is designed to be able to operate from on-premises to the cloud and up to the edge. The variety of environments in which it can be performed as well as the diversity of the components that constitute, requires the use of deployment methodologies that are simple, general purpose and that minimise the possibility of errors.
With this in mind, it was considered to use IaC (Infrastructure as Code) tools like Terraform[50] . Such tools are preferred, as they can easily recreate "on demand" the blueprint environment. Terraform was selected because it is one of the best tools for IaC available on the market and it allows to recreate an infrastructure in a predictable and safe way. Moreover, Terraform is an open-source software with a very large community and it is infrastructure agnostic.
Figure 42 presents a possible flow that could be used to deploy PHYSICS components. This flow is composed of two macro phases. During the first phase, the Terraform scripts are retrieved from the PHYSICS general GIT repository and used to create the environment that will accommodate the PHYSICS components in any location both cloud and edge. In the second phase the HELM charts are used to install and configure the components into the environments created by Terraform. The only prerequisites that the customer deploying PHYSICS needs are the Terraform and HELM clients alongside the resources needed by PHYSICS to run.

[50] Terraform (https://www.terraform.io/intro/index.html)

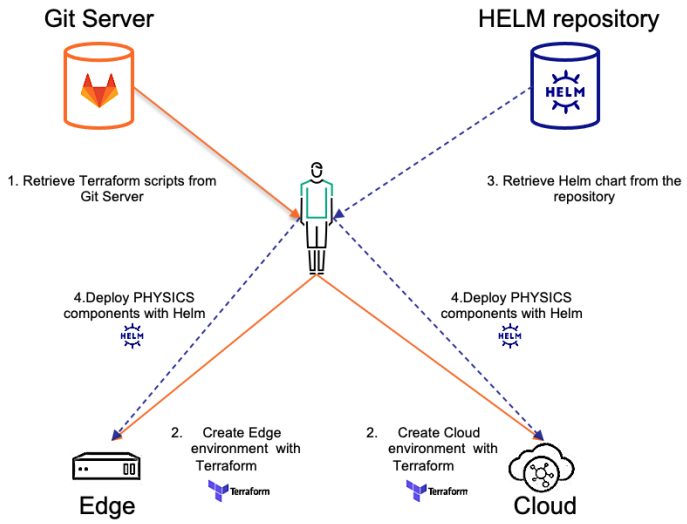


Figure 42 - Deployment flow

5.2 Visual Workflow component

In addition to the implementation of the integration environment, HPE worked with the WP3 team for the architectural definition of the Visual Workflow (Figure 8) in order to provide some essential components for its correct functioning.
In particular, a dedicated Git server based on Gogs was installed within the wp3 namespace responding to the URL https://repo.apps.ocphub.physics-faas.eu to manage the code associated with the development of the various functions that developers can implement through the Visual Workflow.
The functions created, in order to be used and consumed by PHYSICS components, must be transformed into docker images and injected into the OpenWhisk component.
To manage this transformation and interaction with Openwhisk, it was decided to use a dedicated Jenkins server responding to the address https://orchestrator.apps.ocphub.physics-faas.eu/
Two macro pipelines have been created on this server:

- One for managing the transformation of functions in Docker
- Second one for encoding of images in OpenWhisk

The use of Jenkins as an orchestration engine makes both the integration with the Visual Workflow immediate because it has REST APIs that can invoke the created pipelines and thanks to the various plugins it has, it is possible to address many types of interactions out of the box.
At the same time, to manage the multi-user offered by Visual Workflow, a queue system has been set up in which to log the build number and the artefacts produced by the execution of each specific pipeline. This queue system has been relieved by installing a RabbitMQ[51] server as a backend and integrating its use inside Jenkins through the MQ Notifier[52] plugin.
The last component that has been installed and configured in this architecture is MongoDB[53]. Its use was necessary to produce a history of the artefacts produced by the execution of pipelines in Jenkins. MongoDB being a no-SQL DB is particularly suitable for this type of use, because it allows you to store unstructured data and execute optimised queries on them for retrieving information in a very short time.

[51] https://www.rabbitmq.com/documentation.html
[52] https://plugins.jenkins.io/mq-notifier/
[53] https://www.mongodb.com/what-is-mongodb

6. CONCLUSIONS

This document has reported the results of PHYSICS WP6 Task T6.1 “Solution Services Integration and Reusable Artefacts Marketplace Platform (RAMP) Creation”, achieved during the first phase of the project,

The document is the accompanying textual specification of the major result of the deliverable and the task: the first version of the prototype of the integrated PHYSICS solution framework and RAMP which has been deployed into the PHYSICS blueprint reference target infrastructure. The document and the integrated PHYSICS solution framework and RAMP setup constitute the overall deliverable and task output.

The achieved results provided key contributions for the fulfilment of the 2nd major WP6 milestone (MS5 – PHYSICS 1st integrated platform release – foreseen for M15 of the project) and provide the first release of the proposed solution.

The work has been carried out in close cooperation and coordination with the other PHYSICS WP6 tasks and Work Packages 2-3-4-5 tasks and partners, taking into account and integrating the delivered results and concepts (e.g. the PHYSICS Reference Architecture proposed by WP2 and the solution framework major components and services artefacts proposed by WP3, WP4 and WP5) in a coherent and uniform manner.

The overall progress of T6.1 will be one of the major drivers of the remaining WP6 tasks, mainly T6.3 (Use Cases Adaptation & Experimentation) and T6.4 (Use Case Evaluation) for the 1st iteration of the PHYSICS Pilots and Use Cases Operations and Stakeholders’ Evaluation of the proposed solution framework.

Finally, the delivered integrated PHYSICS solution framework and RAMP marketplace will be fundamental inputs and drivers for WP7 (Exploitation, Dissemination and Impact Creation), with special emphasis on T7.2 (Business Innovation Development & Exploitation).

The WP6 work on Tasks 6.1 will continue without any interruption in the next period of the project, evolving and enriching the proposed solution framework and marketplace with additional capabilities and features. Such enhancements will take into account the latest evolutions of relevant technologies occurring during the related project timeframe, and also the expected feedbacks coming from the initial wave of the Pilot Operations and Stakeholders Evaluation tasks.

Such work will lead to the second and final releases of the deliverable, which will be fully documented in deliverable D6.2 (Prototype of the Integrated PHYSICS solution framework and RAMP V2), planned for M35.

7. REFERENCES

[1] “PHYSICS official Web site,” 31 Mar 2022. [Online]. Available: https://physics-faas.eu/wp-content/uploads/2021/10/PHYSICS-D2.4-Reference-Architecture-Specification-V1.pdf.

[2] F. D. &. B. J., “Graph Databases Comparison: AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, and OrientDB,” *In Data,* pp. 373-380, 2018 July.

[3] S. University, “Pocket,” [Online]. Available: https://github.com/stanford-mast/pocket.

[4] A. a. W. Y. a. S. P. a. T. A. a. P. J. a. K. C. Klimovic, “Pocket: Elastic Ephemeral Storage for Serverless Analytics,” in *OSDI'18*, CARLSBAD, CA, USA, 2018.

DISCLAIMER

COPYRIGHT MESSAGE