



# PHYSICS

OPTIMIZED HYBRID SPACE-TIME SERVICE CONTINUUM IN FAAS

## D6.2 – PROTOTYPE OF THE INTEGRATED PHYSICS SOLUTION FRAMEWORK AND RAMP V2

<b>Lead Beneficiary</b>	HPE
<b>Work Package Ref.</b>	WP6 – Use Cases Adaptation, Experimentation, Evaluation
<b>Task Ref.</b>	T6.1 – Solution Services Integration and Reusable Artefacts Marketplace Platform (RAMP) Creation
<b>Deliverable Title</b>	D6.2 – Prototype of the Integrated PHYSICS solution framework and RAMP V2
<b>Due Date</b>	2023-10-31
<b>Delivered Date</b>	2023-10-31
<b>Revision Number</b>	3.0
<b>Dissemination Level</b>	Public (PU)
<b>Type</b>	Demonstrator (DEM)
<b>Document Status</b>	Release
<b>Review Status</b>	Internally Reviewed and Quality Assurance Reviewed
<b>Document Acceptance</b>	WP Leader Accepted and Coordinator Accepted
<b>EC Project Officer</b>	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 <sup>1</sup>	Name Surname <sup>2</sup>
HPE	Lead Beneficiary	Alessandro Mamelli, Domenico Costantino, Roberto Musso
UPM	Contributor	
HUA	Contributor	
GFT	Contributor	
RHT	Contributor, Internal Reviewer	Luis Tomas Bolivar
INNOV	Contributor	
INQBIT	Contributor	
DFKI	Quality Assurance	Carsten Harms, Maciej Kolek
BYTE	Contributor	
RYAX	Contributor	
ATOS	Contributor, Internal Reviewer	Carlos Sánchez Fernández

## REVISION HISTORY

Version	Date	Partner(s)	Description
0.1	2023-06-21	HPE	ToC Version and preliminary contents
0.2	2023-07-10	HPE	Refined ToC Version and additional preliminary contents
1.0	2023-09-08	HPE, all Contributors	1 <sup>st</sup> integrated version
1.1	2023-10-04	HPE, all Contributors	2 <sup>nd</sup> integrated version
1.2	2023-10-11	HPE	Version for Peer Reviews
1.3	2023-10-18	ATOS-RHT	Peer Reviews completed
2.0	2023-10-25	HPE	Version for Quality Assurance
2.1	2023-10-27	DFKI	Quality Assurance completed
3.0	2023-10-30	HPE	Version for Submission

---

<sup>1</sup> Lead Beneficiary, Contributor, Internal Reviewer, Quality Assurance

<sup>2</sup> Can be left void

## LIST OF ABBREVIATIONS

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

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

## EXECUTIVE SUMMARY

Within the scope of PHYSICS Work Package 6 (Use Cases Adaptation, Experimentation, Evaluation), this document describes the final results of Task T6.1 (Solution Services Integration and Reusable Artefacts Marketplace Platform (RAMP) Creation) achieved during the second and last phase of the project, and provides the second version (D6.2) of this deliverable (out of the two iterations foreseen in the whole work plan for WP6, i.e. D6.1 and D6.2).

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 7th major WP6 milestone (MS12 – PHYSICS 2nd integrated platform release – foreseen for M34 of the project) and provide the second and final release of the proposed solution.

The document is the accompanying textual specification of the major result of the deliverable and the task: the second 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 final 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 2nd 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 are 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.

## SUMMARY OF CHANGES FROM D6.1

CHAPTER	UPDATE	SECTION(s)
1	Updated text	All
2	Alignment to the final version of the architecture, new sample application of the integrated PHYSICS solution framework and final RAMP overview	2.1 – 2.2 – 2.3
3	Update of all the components of the Integrated PHYSICS solution framework (with addition of new components: Gaming Platform, Runtime Adaptation, Cluster Availability Monitor)	3.x
4	Update of the design and implementation of the final version of the Reusable Artefacts MarketPlace (RAMP) application	All
5	Update of the final integrated development and testing environment upon which the PHYSICS solution framework is built (with addition of “Cross infrastructure components” and “Visual Workflow on cloud” sections)	5.x
6	Updated text	6

# CONTENTS

1.	Introduction.....	15
1.1	Objectives of the Deliverable.....	15
1.2	Insights from other Tasks and Deliverables.....	16
1.3	Structure .....	17
2.	Integrated PHYSICS solution framework and RAMP Overview.....	18
2.1	PHYSICS solution framework architecture overview.....	18
2.1.1	Design, Deployment and Execution of Functions .....	19
2.1.2	Sample sequence flow.....	22
2.2	Sample application of the integrated PHYSICS solution framework.....	24
2.2.1	Sample sequence flow.....	24
2.2.2	Orchestrator Flow.....	25
2.3	Reusable Artefacts MarketPlace (RAMP) overview.....	26
3.	Integrated PHYSICS solution framework Implementation .....	28
3.1	Visual Workflow.....	28
3.1.1	Overview .....	28
3.1.2	Technology architecture .....	28
3.1.3	Interfaces/API.....	29
3.1.4	Distribution, deployment and configuration .....	46
3.1.5	Control UI.....	46
3.1.6	Cloud version .....	51
3.2	Semantic Extractor.....	55
3.2.1	Overview .....	55
3.2.2	Technology architecture .....	56
3.2.3	Interfaces/API.....	56
3.2.4	Distribution, deployment and configuration .....	57
3.2.5	Sample Application Transformation .....	58
3.3	Design Patterns.....	60
3.3.1	Overview .....	60
3.3.2	Technology architecture .....	60
3.3.3	Interfaces/API.....	61
3.3.4	Distribution, deployment and configuration .....	61
3.3.5	Flow update process .....	66
3.3.6	Individual integration points of Patterns with the Data Management Service of T4.4....	68
3.4	Elasticity Controllers .....	69
3.4.1	Overview .....	69
3.4.2	Technology architecture .....	69

3.4.3	Interfaces/API.....	70
3.4.4	Distribution, deployment and configuration .....	70
3.5	Gaming Platform .....	70
3.5.1	Overview .....	70
3.5.2	Technology architecture .....	71
3.5.3	Interfaces/API.....	72
3.5.4	Distribution, deployment and configuration .....	73
3.5.5	Game Portal Tutorials.....	73
3.6	Reasoning Framework .....	76
3.6.1	Overview .....	76
3.6.2	Technology architecture .....	77
3.6.3	Interfaces/API.....	78
3.6.4	Distribution, deployment and configuration .....	78
3.7	Runtime Adaptation.....	79
3.7.1	Overview .....	79
3.7.2	Technology architecture .....	79
3.7.3	Interfaces/API.....	81
3.7.4	Distribution, deployment and configuration .....	82
3.8	Cluster Availability Monitor .....	82
3.8.1	Overview .....	82
3.8.2	Technology architecture .....	83
3.8.3	Interfaces/API.....	83
3.8.4	Distribution, deployment and configuration .....	83
3.9	Resource Semantics .....	84
3.9.1	Overview .....	84
3.9.2	Technology architecture .....	85
3.9.3	Interfaces/API.....	85
3.9.4	Distribution, deployment and configuration .....	86
3.10	Performance Evaluation Framework .....	86
3.10.1	Overview .....	86
3.10.2	Technology architecture .....	87
3.10.3	Interfaces/API.....	87
3.10.4	Distribution, deployment and configuration .....	91
3.10.5	Individual integration points of PEF with other components .....	91
3.11	Global Continuum Placement .....	92
3.11.1	Overview .....	92
3.11.2	Technology architecture .....	93
3.11.3	Interfaces/API.....	93



3.11.4	Distribution, deployment and configuration .....	94
3.12	Distributed In-Memory Service .....	95
3.12.1	Overview .....	95
3.12.2	Technology architecture .....	95
3.12.3	Interfaces/API.....	96
3.12.4	Distribution, deployment and configuration .....	96
3.13	Adaptive Platform Deployment, Operation & Orchestration .....	96
3.13.1	Overview .....	96
3.13.2	Technology architecture .....	97
3.13.3	Interfaces/API.....	98
3.13.4	Distribution, deployment and configuration .....	98
3.14	Scheduling Algorithms (Local Adaptive Scheduler) .....	99
3.14.1	Overview .....	99
3.14.2	Technology architecture .....	99
3.14.3	Interfaces/API.....	100
3.14.4	Distribution, deployment and configuration .....	100
3.15	Resource Management Controllers .....	102
3.15.1	Overview .....	102
3.15.2	Technology architecture .....	102
3.15.3	Interfaces/API.....	104
3.15.4	Distribution, deployment and configuration .....	106
3.16	Co-Allocation Strategies.....	106
3.16.1	Overview .....	106
3.16.2	Technology architecture .....	107
3.16.3	Interfaces/API.....	109
3.16.4	Distribution, deployment and configuration .....	109
4.	Reusable Artefacts MarketPlace Implementation.....	110
4.1	Overview .....	110
4.2	Technology architecture .....	110
4.3	Artefacts.....	111
4.4	Distribution, deployment and configuration .....	112
4.5	User Story .....	112
5.	PHYSICS solution framework Integration environment .....	116
5.1	Integration Infrastructure.....	116
5.1.1	Development strategy.....	117
5.1.2	Deployment strategy .....	120
5.2	Cross infrastructure components.....	121
5.3	Visual Workflow component .....	122

5.4	Visual Workflow on cloud .....	122
6.	Conclusions .....	127
7.	References .....	128

## FIGURES

Figure 1 - High level relations between WP6 and T6.1 and the other technical WPs .....	16
Figure 2 - prototype architecture from RA .....	18
Figure 3 - Design, deployment and execution of a function .....	20
Figure 4 - WP3/4 Integration Diagram for Application Deployment.....	23
Figure 5 - Sample Node-RED function flow.....	25
Figure 6 - Sample Node-RED Orchestrator flow as Function .....	26
Figure 7 - RAMP High Level Overview .....	27
Figure 8 - Visual Workflow components integration schema .....	29
Figure 9 - Node-RED environment.....	47
Figure 10 - Build flow .....	47
Figure 11 - Test flow .....	48
Figure 12 - See created and draft graphs .....	48
Figure 13 - Create a new graph .....	49
Figure 14 - Import image .....	49
Figure 15 - Export subflow.....	50
Figure 16 - Dashboard .....	50
Figure 17 - Control UI cloud.....	51
Figure 18 - Build and Execution Process for the Semantic Extractor .....	58
Figure 19 - Example DE output towards SE for the sample App .....	59
Figure 20 - SE Annotated Output Graph towards RF.....	60
Figure 21 - PHYSICS Patterns Palette in Node-RED.....	62
Figure 22 - Example UI configuration and README file in Pattern Node .....	62
Figure 23 - Subflow Description Information for npm node conversion of a subflow .....	63
Figure 24 - Example Subflow Node published on npm .....	64
Figure 25 - a) Node Addition Process in Node-RED b) Available Example Node on Node-RED repo.....	65
Figure 26 - Direct Installation of Node through the built-in Palette Management of Node-RED .....	65
Figure 27 - Retrieval of Pattern Subflow from PHYSICS Node-RED repo collection.....	66
Figure 28 - Import of Updated Pattern Subflow in the DE Node-RED Editor .....	67
Figure 29 - Warning Message for Duplicate Nodes.....	67
Figure 30 - Selection of Node and Subflow Replacement Option .....	68
Figure 31 - Different Appearance of Versions a) without name differentiation b) with name differentiation.....	68
Figure 32 - Elasticity Controllers Flow .....	69
Figure 33 - Gaming Platform Architecture .....	71
Figure 34 - Settings Screen.....	73
Figure 35 - Game Starting Screen .....	74
Figure 36 - Local Menu Screen .....	74
Figure 37 - Node-RED Cookbook Screen .....	75
Figure 38 - Messages Tutorial Screen .....	75
Figure 39 - Messages #1 Tutorial Screen .....	76
Figure 40 - Node-RED Portal Screen.....	76
Figure 41 - Reasoning Framework interactions with other components .....	77
Figure 42 - Reasoning's Framework internal components.....	78
Figure 43 - Screenshot of Reasoning framework logs.....	79
Figure 44 - Runtime Adaptation Architecture.....	80
Figure 45 - Screenshot of Cluster Availability Monitor logs.....	84
Figure 46 - Request Aggregator Instantiation in PEF for the eHealth Use Case.....	92
Figure 47 - Global Continuum placement component.....	93
Figure 48 - DMS component.....	95
Figure 49 - Orchestrator flow.....	97

Figure 50 - PHYSICS cluster onboarding .....	103
Figure 51 - Resource Management Components and interactions overview .....	104
Figure 52 - Co-allocation invocation Technology architecture.....	107
Figure 53 - Co-allocation strategies component internal architecture .....	108
Figure 54 - RAMP Architecture .....	111
Figure 55 - Homepage.....	113
Figure 56 - Assets page .....	114
Figure 57 - An asset in RAMP .....	115
Figure 58 - Form to add/request Asset in/from RAMP .....	115
Figure 59 - eHealth Use Case Post.....	116
Figure 60 - DevOps Tools.....	117
Figure 61 - CI/CD flow.....	118
Figure 62 - Integration namespaces .....	119
Figure 63 - OKD GUI .....	119
Figure 64 - OC client.....	119
Figure 65 - Organizations inside Gogs.....	120
Figure 66 - Repositories inside one organization .....	120
Figure 67 - Deployment flow .....	121
Figure 68 - Visual Workflow CLOUD .....	123
Figure 69 - Jenkins Pipeline for Visual Workflow .....	125

## TABLES

Table 1 - VW/API- get flow .....	29
Table 2 - VW/API get subflow .....	30
Table 3 - VW/API- build flow.....	30
Table 4 - VW/API- build status .....	30
Table 5 - VW/API- delete build .....	31
Table 6 - VW/API- deploy app graph.....	31
Table 7 - VW/API- deploy status .....	31
Table 8 - VW/API- last state of deploy status .....	32
Table 9 - VW/API-create graph.....	32
Table 10 - VW/API- get all created graphs.....	32
Table 11 - VW/API- get all graph drafts .....	33
Table 12 - VW/API get functions .....	33
Table 13 - VW/API- invoke functions.....	33
Table 14 - VW/API- get function activation.....	34
Table 15 - VW/API- get clusters .....	34
Table 16 - VW/API- get flow file .....	34
Table 17 - VW/API- start performance pipeline.....	35
Table 18 - VW/API- get performance result.....	35
Table 19 - VW/API- performance pipeline status.....	35
Table 20 - VW/API- get all performance pipeline status.....	36
Table 21 - VW/API- get imported image .....	36
Table 22 - VW/API- Import image.....	37
Table 23 - VW/API- Check if import image already exists .....	37
Table 24 - VW/API - Get all imported images .....	38
Table 25 - VW/API - Get credentials id .....	38
Table 26 - VW/API - Create credentials id .....	39
Table 27 - VW/API- import image status .....	39
Table 28 - VW/API- export subflow.....	39
Table 29 - VW/API- create artifact.....	40
Table 30 - VW/API- get artifact.....	40
Table 31 - VW/API- delete artifact.....	41
Table 32 - VW/API- get draft service.....	41
Table 33 - VW/API- create graph draft .....	41
Table 34 - VW/API- get function service .....	42
Table 35 - VW/API- invoke function.....	42
Table 36 - VW/API- get activationID .....	42
Table 37 - VW/API- get all clusters service .....	42
Table 38 - VW/API- get imported image .....	43
Table 39 - VW/API- Insert import image.....	43
Table 40 - VW/API- Update import image .....	44
Table 41 - VW/API- Check if import image already exists .....	44
Table 42 - VW/API - Get all imported images .....	45
Table 43 - VW/API - Get credentials id .....	45
Table 44 - SE-API-semantic retrieval .....	56
Table 45 - Local Mode APIs .....	72
Table 46 - Online Mode APIs.....	72
Table 47 - Marketplace APIs .....	73
Table 48 - Interfaces and Endpoints for Runtime Adaptation .....	81
Table 49 - Cluster Availability Monitor Interfaces.....	83
Table 50: PEF Raw Profiling Data API .....	87

Table 51: PEF Benchmarking Data API .....	87
Table 52: PEF Clustering Push Data API .....	88
Table 53: PEF Clustering Retrieval API .....	89
Table 54: PEF Function Profile API .....	90
Table 55 - Global Continuum API for the scheduler .....	93
Table 56 - Global Continuum API for monitoring .....	94
Table 57 - DMS-API .....	96
Table 58 - Orchestrator component runtime RPC API .....	98
Table 59 - Co-Allocation/API-get affinities .....	109

# 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 generalized 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;
- Optimized 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.) can 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 final results of PHYSICS WP6 Task T6.1 “Solution Services Integration and Reusable Artefacts Marketplace Platform (RAMP) Creation” and provides the second version (D6.2) of the deliverable (out of the two foreseen in the WP6 work plan for this task, i.e. D6.1 and D6.2). 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 7th major WP6 milestone (MS12 – PHYSICS 2nd integrated platform release – foreseen for M34 of the project) and provide the second and final release of the proposed solution.

The document is (on purpose) self-contained, i.e., there's no need to read the previous version (D6.1) to get a full understanding of the updated contents and achieved results. This release includes many additional and enhanced features with respect to the previous one, which are summarized in the table "SUMMARY OF CHANGES FROM D6.1" that can be found earlier, after the "EXECUTIVE SUMMARY" of the document.

The document is the accompanying textual specification of the major result of the deliverable and the task: the second 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:

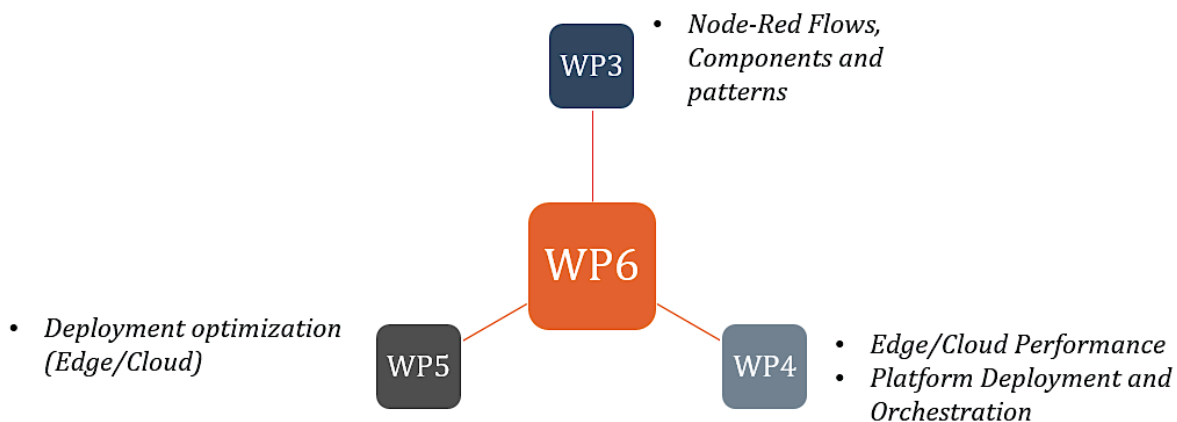


Figure 1 - High level relations between WP6 and T6.1 and the other technical WPs

Important and relevant inputs for WP6 Task 6.1 are the outcomes of:

- WP3 tasks and their final available deliverable "D3.2 – Functional and Semantic Continuum Services Design Framework, Scientific Report and Prototype Description V2";
- WP4 tasks and their final available deliverable "D4.2 – Cloud Platform Services for a Global Continuum Space-Time Continuum Interplay, Scientific Report and Prototype Description v2";
- WP5 tasks and their final available deliverable "D5.2 – Extended Infrastructure Services with Adaptable Algorithms Scientific Report and Prototype Description V2".

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 upcoming 2nd 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 final PHYSICS Reference Architecture).

Finally, the delivered integrated PHYSICS solution framework and RAMP marketplace are 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 final version of the prototype, its architecture and relationship to the final 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 final version of the prototype of the integrated PHYSICS solution framework;
- Chapter 4 “Reusable Artefacts MarketPlace Implementation” describes the design and implementation of the final 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 final version of the prototype of the integrated PHYSICS solution framework and RAMP, through its architecture and relationship to the final 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 summarizes the final architecture of the integrated PHYSICS solution framework. The PHYSICS architecture was described in deliverable D2.5 Reference Architecture SpecificationV2 (PHYSICS Consortium, 2022).

The final version of the prototype implementation of the integrated PHYSICS solution is fully aligned to the final PHYSICS architecture. The main components of the PHYSICS architecture, implemented in the PHYSICS prototype, are shown in Figure 2

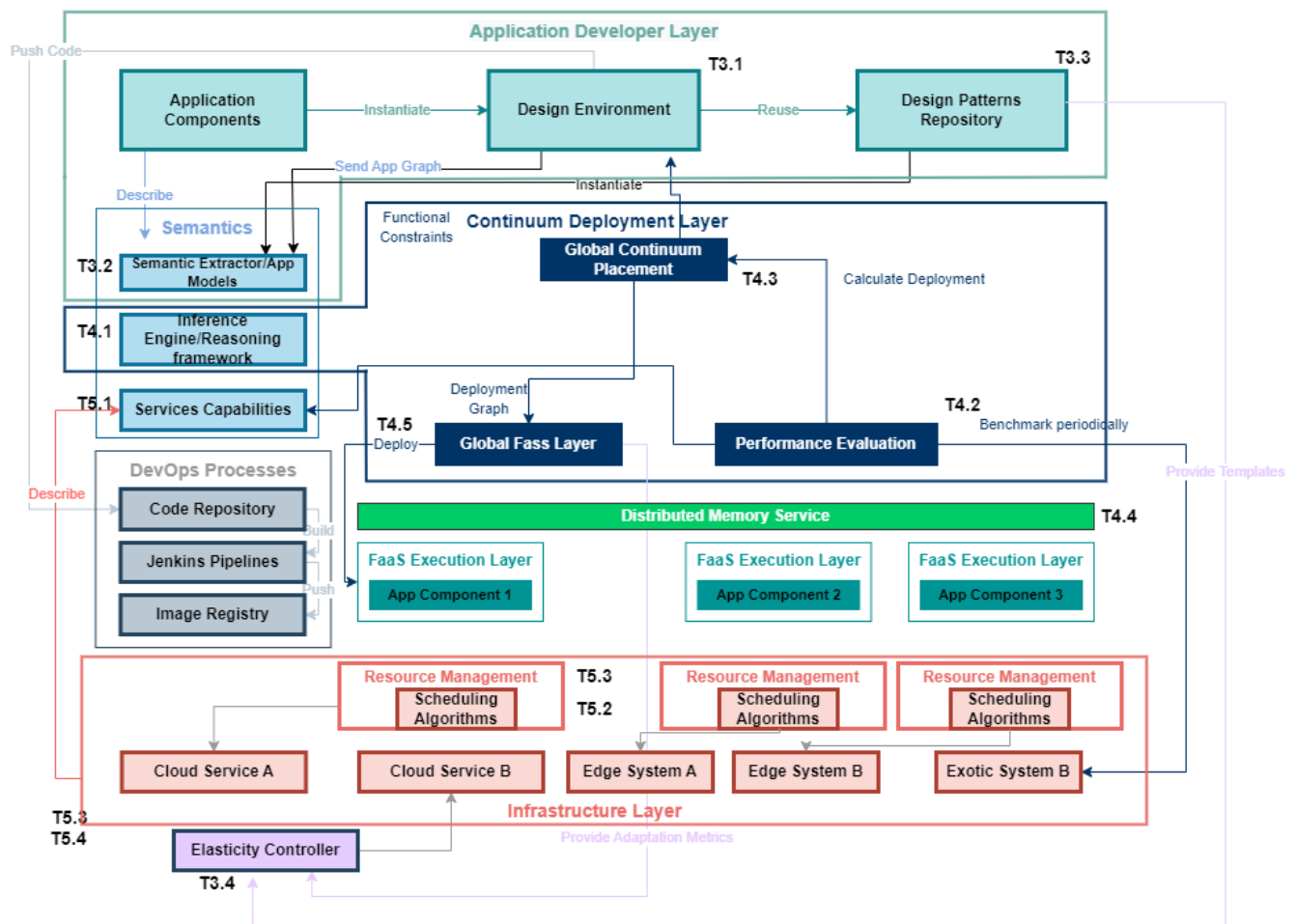


Figure 2 - prototype architecture from RA

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 parallelization, 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 Extractor/ Application Models). Application components may have elasticity controllers that regulate the algorithms and resources needed for scaling a component. Once the application is ready an application graph is built by the Semantic Extractor (SE) which processes the flows, extracts their structure, as well as any other semantic annotation. This information is mapped to ontological triples and is stored in the Reasoning Framework/Inference Engine.

The Continuum Deployment Layer oversees the deployment of applications and providing uniform access to the diverse cloud services provided by one or more cloud providers. Once an application is built in the previous step, it can be deployed. This step is initiated from the Design environment contacting the Reasoning Framework. It retrieves all triples for the application, filters the list of candidate services based on the application graph needs, and forwards the application graph to the Global Continuum Placement (Placement Optimizer). The Global Continuum Placement decides the most suitable deployment of applications considering the performance of the services, costs, and affinity constraints of components. For that purpose, it uses the performance of the services provided by the Performance Evaluation component. The placement creates a deployment graph which is forwarded to the Global FaaS Layer (Orchestrator), which may query the Reasoning Framework for details of the available clusters (e.g., endpoint, credentials etc.). The Orchestrator will use the function information and do the actions needed to register the function in a cluster using a Kubernetes operator. In this step, the application functions and flows are deployed and ready to be used. The Orchestrator (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 optimized operation of the edge and cloud services utilized for the realization of the application service graph. It deals with a single cluster. To this end the Service Capabilities component depicts and models the abilities of each service and resource type (used by the Reasoning Framework in the previous step) in a cluster (cloud provider). 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 scheduling algorithms take into the information provided Co-location strategies component to create instances of functions in the cluster nodes in order to maximize performance. The co-location component based on the interferences of functions running in the cluster nodes generates affinity and anti-affinity rules to be used by the scheduler to decide in which node a function will be finally executed in a given cluster.

### 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)

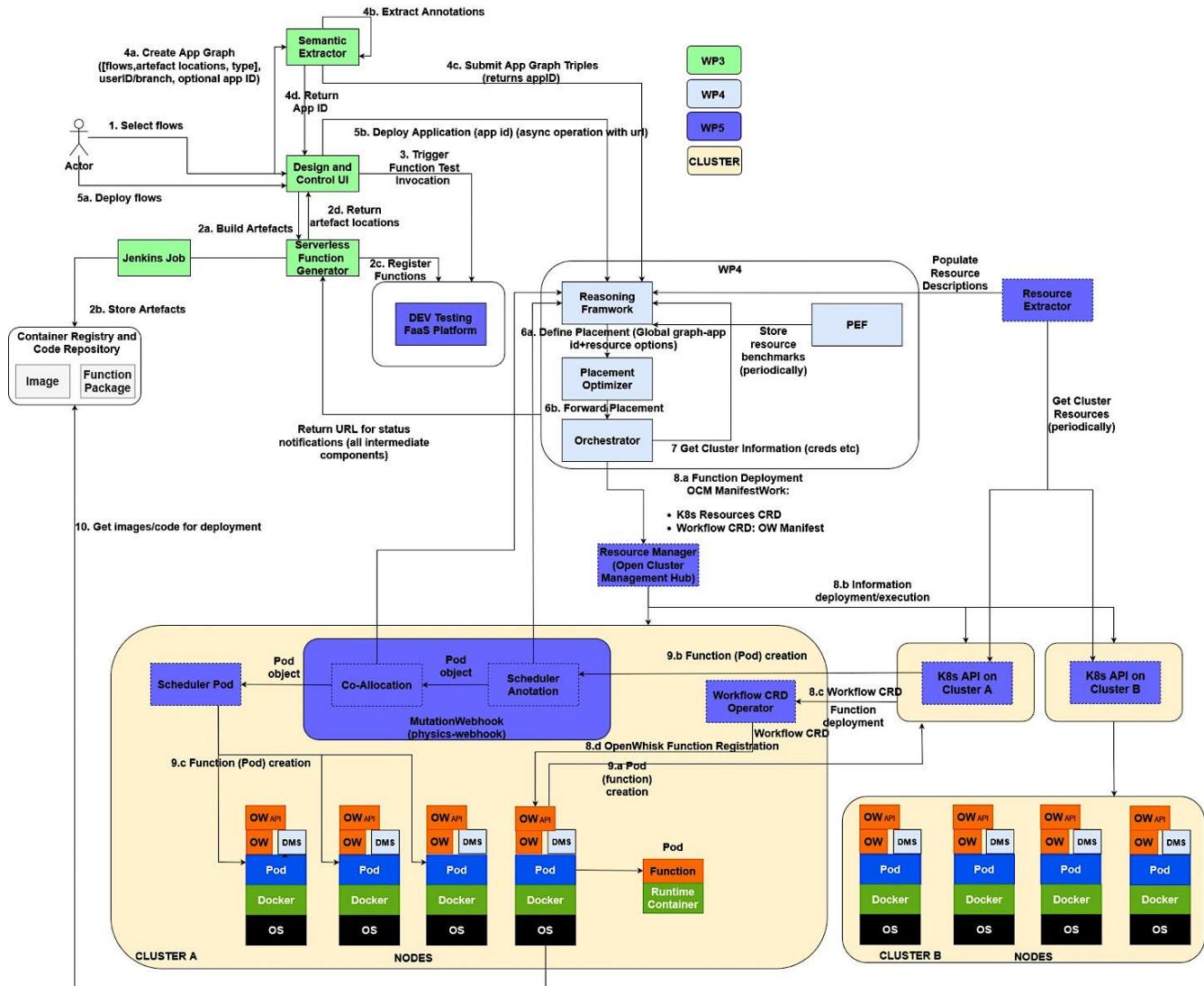


Figure 3 - Design, deployment and execution of a function

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 summarized 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 finalization of the process, the artefacts are stored in the image registry.
  - c. After the deployable artefacts are ready, the related functions (i.e., OpenWhisk (OW) actions) are also registered in a test OpenWhisk environment.
  - d. Upon finalization 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 (RF) 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. 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 finalized, the developer can initialize 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. 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 through the OpenWhisk API (OW function registration)

#### 8. 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 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.



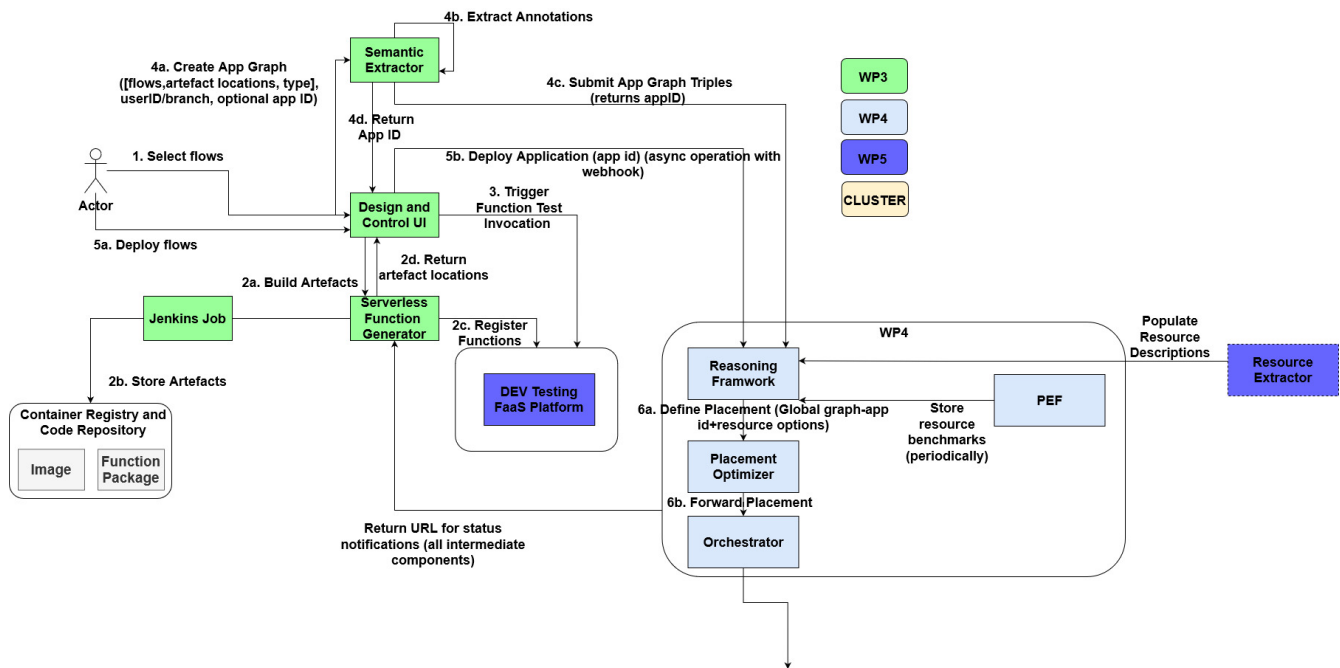


Figure 4 - WP3/4 Integration Diagram for Application Deployment

The developer uses the Design Environment Control UI 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 finalization 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 environment. This is necessary for testing the actual implementation as a function execution before submitting to the production cluster. Upon finalization the deployable artefact locations are returned to the Control UI 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 initialize 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 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 finalized 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 finalized, 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, a sample application that uses the capabilities of the integrated PHYSICS solution framework was created. The purpose of this application is primarily to test new features of the PHYSICS platform, such as the inclusion of benchmarking results and resource profiling, the ability to dynamically place functions and invoke them in the scope of one application etc., 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 2 main flows, aiming to test basic functionalities such as image building etc., as well as the creation of the app graph (including annotations), the registration to the Reasoning Framework, registration and execution to OpenWhisk of a custom image.

### 2.2.1 Sample sequence flow

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

- The build process for the Node-RED runtime image creation
- The OW interface and argument passing
- The included annotations are:
  - **Importance**=high flow level value to be taken under consideration by the scheduling layers
  - **OptimizationGoal** (performance), to be taken under consideration by the optimization placement layers of WP4
  - **Sizing annotator node** for setting Memory=512 MB and Timeout=220000, to be applied in the Node-RED function registration process
  - **Executor mode** for indicating Node-RED flow as function execution
  - **Locality**=aws, to be taken under consideration by the placement layers of WP4
- The specific function has also undergone the process of the Performance Pipeline, described in D3.2, to extract benchmarking results on both available clusters (AWS and Azure) as well as to extract the resource profile of the specific function. This is needed for the function to participate in the placement optimization and coallocation strategies.



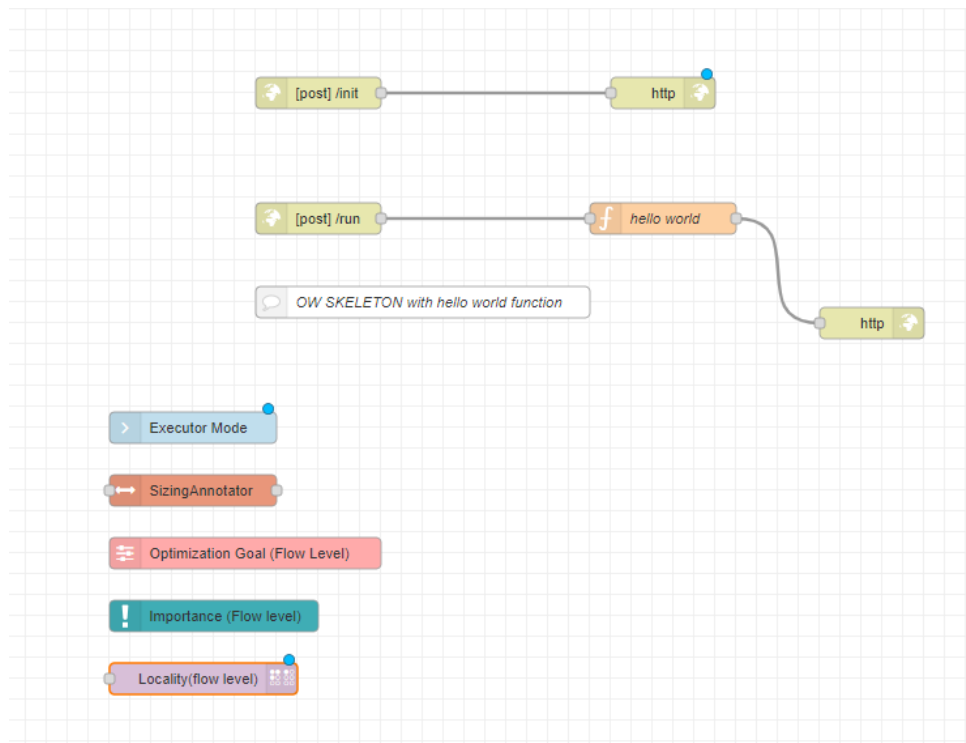


Figure 5 - Sample Node-RED function flow

### 2.2.2 Orchestrator Flow

The second flow (Figure 6) refers to a sample Node-RED flow function that needs to orchestrate and invoke actions in the same application graph, wrapped around an OW skeleton pattern. The main goal of this flow is to test the dynamic placement of the functions in the graph in a multi-cluster setting. Given that the developer may not know beforehand where each function will be placed on a multi-cluster setting (due to the PHYSICS global placement optimization process), they cannot dictate the location of the invoked function at design time. They could do that through the locality annotation however this would negate the benefits of the multi-cluster ability. Things to test in this case include:

- The Dynamic Orchestrator pattern, defined in D3.2, and how the information can be propagated through the respective layers, indicatively
  - Semantic annotations inside the flow
  - Application graph creation through the Semantic Extractor
  - Storage and manipulation by the Reasoning Framework to reach the Orchestrator of WP4
- Dynamic configuration of the created orchestrator function with information on the location of the invoked function by the Orchestrator and successful execution of the application in the two available clusters, with one function being able to invoke the other remotely placed function without using static configuration. This is needed for the Adaptive Platform Deployment process
- The flow includes also a QoSRequirements semantic node, which indicates the need for an average wait time less than 200 milliseconds, a setting that can be used in the Runtime Adaptation process.

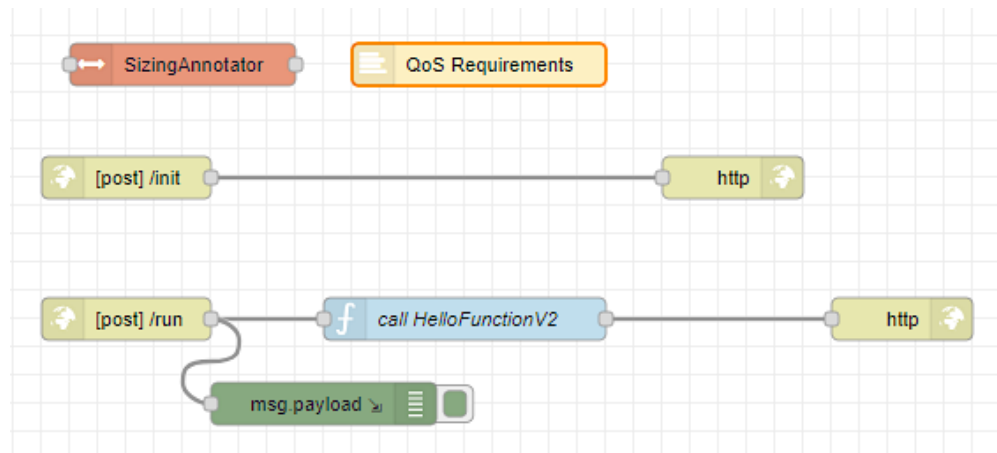


Figure 6 - Sample Node-RED Orchestrator flow as Function

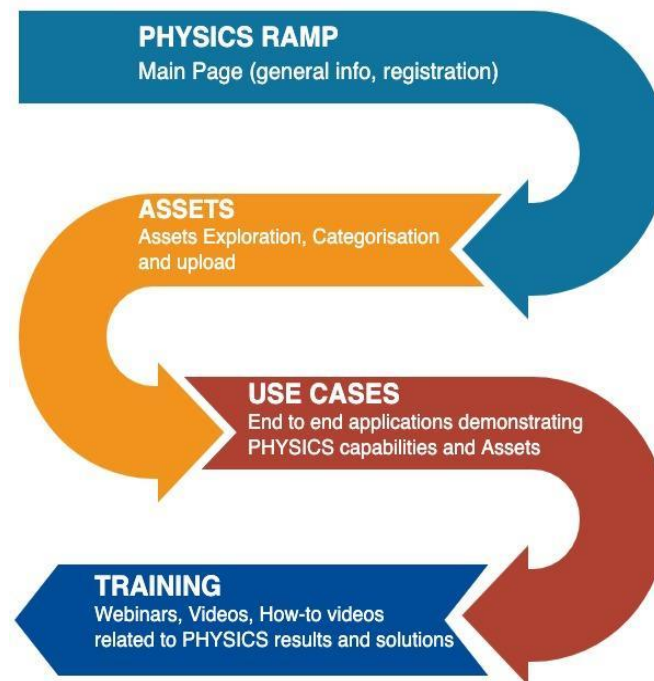
Through the availability of the resource profile of at least one function, we can also test the optimized cluster operation and the co-allocation scenarios, provided that the function for which the performance profile exists is scheduled on the respective cluster that includes this mechanism and is executed alongside other functions for which their profile is known.

Thus, the setup of this test application and the careful selection of the parameters included in its functions enables the testing of all the main parts of the PHYSICS platform and newly added Y3 features.

## 2.3 Reusable Artefacts MarketPlace (RAMP) overview

The Reusable Artefacts MarketPlace (RAMP) serves as a dynamic platform that bridges the gap between the project's solutions, results, and use cases, and the external stakeholders and initiatives that can leverage these outcomes. It is an ecosystem that fosters the growth and development of reusable low-code artefacts. These artefacts facilitate the creation of serverless applications or aid their transition to the serverless cloud paradigm.

RAMP's design encourages collaboration and innovation. It enables external project entities to host their solutions, thereby expanding the range of available artefacts. Moreover, it provides an interface for these entities to request extensions or technical assistance for an existing artefact, fostering a collaborative environment that promotes continuous improvement and adaptation. RAMP's progress and advancements made during the second period of the project are detailed in Section 4. For a concise understanding of its structure, refer to Figure 7 which provides a high-level overview of the platform's architecture.



*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 second and final 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 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 (also known as Design Environment) is a web application, which embeds Node-RED environment, communicates with the API of Node-RED and provides features to build and deploy flows to the other PHYSICS Components.

##### 3.1.2 Technology architecture

Visual workflow consists of the following major components:

1. Control UI (Frontend application)
2. Serverless Function Generator (SFG) (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)
9. Import Image (Microservice)
10. Cluster Service (Microservice)

The entry point is the Control UI, which communicate through the REST API with SFG. SFG uses REST API to communicate with Artifact Query Service, Graph Draft Service, Function Service, Semantic Extractor, Inference Engine, Import Image, Cluster service 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 to by the Artifact Processor and Graph Processor. Deployment state reacts on a queue message to update the state on the WebSocket of the deployment state of the graph. The following Figure 1Figure 8 shows a graphical representation of the components integration.

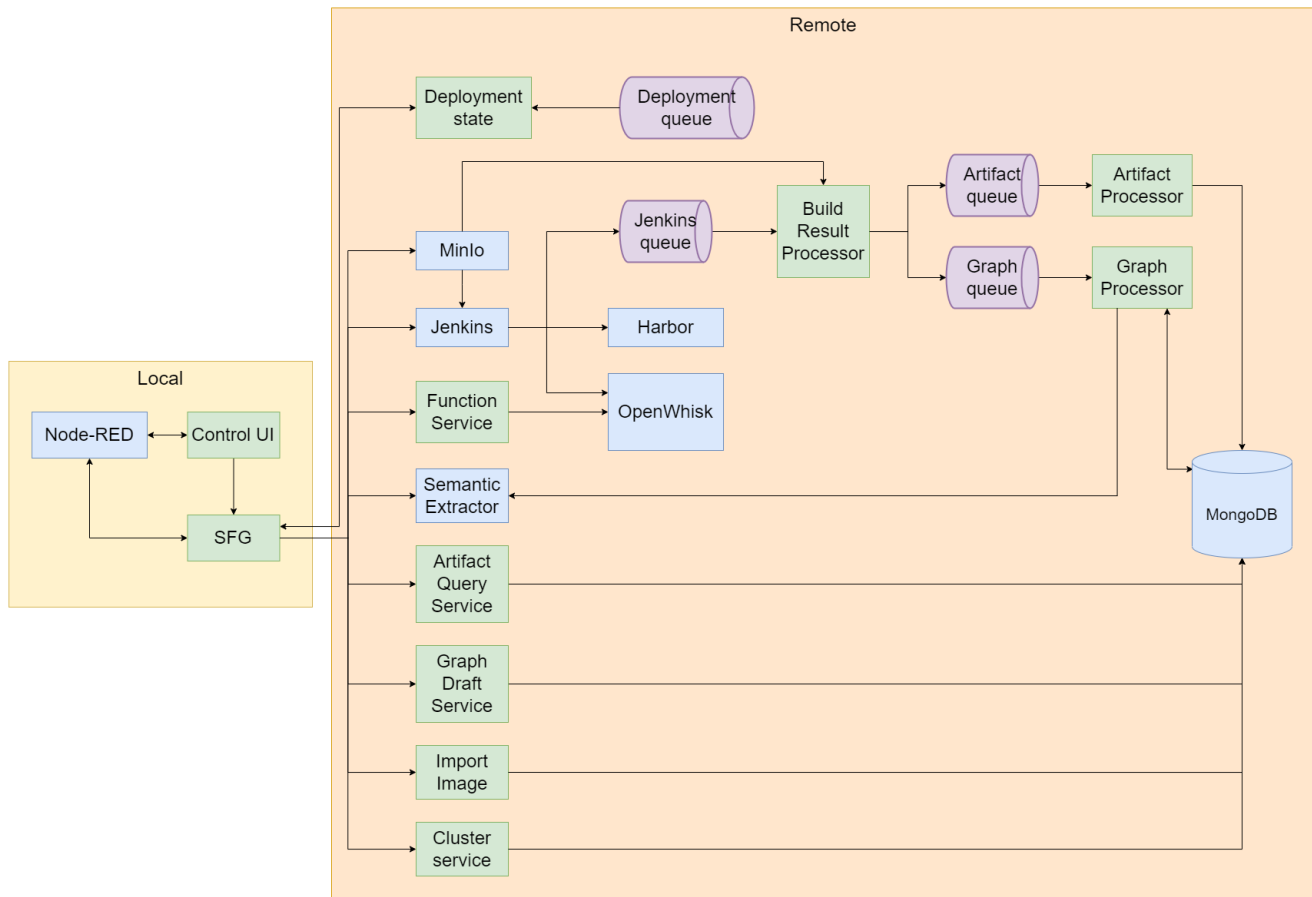


Figure 8 - Visual Workflow components integration schema

### 3.1.3 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
<b>Method</b>	GET
<b>Request body</b>	-
<b>Success response</b>	HTTP 200 with response body: <pre>{   "success": boolean,   "value": Flow[] }</pre>
<b>Error response</b>	HTTP 404 when Node-RED or database is unavailable

➤ Get Subflow

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

*Table 2 - VW/API get subflow*

Path	/subflow
<b>Method</b>	GET
<b>Request body</b>	-
<b>Success response</b>	HTTP 200 with response body: <pre>{   "success": boolean,   "value": [     {       "id": string,       "name": string,       "info": string,       "category": string,     }   ] }</pre>
<b>Error response</b>	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 3 - VW/API- build flow*

Path	/build
<b>Method</b>	POST
<b>Request body</b>	<pre>{   "flowId": string   "branchName": string }</pre>
<b>Success response</b>	HTTP 200 with response body: <pre>{   "results": "url_to_jenkins_job", }</pre>
<b>Error response</b>	HTTP 500 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 4 - VW/API- build status*

Path	/build-status
<b>Method</b>	POST
<b>Request body</b>	-
<b>Success response</b>	Message Event with data: <pre>{   status: BuildStatus,   "flowId": string }</pre>

	}
<b>Error response</b>	Message Event with error message.

- Delete build

Emits status of current Jenkins jobs

*Table 5 - VW/API- delete build*

<b>Path</b>	<b>/build/:id</b>
<b>Method</b>	DELETE
<b>Request body</b>	-
<b>Success response</b>	HTTP 200 with response body: "id_artifact_removed"
<b>Error response</b>	HTTP 404 if no artifact present

- Deploy App Graph

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

*Table 6 - VW/API- deploy app graph*

<b>Path</b>	<b>/deployment</b>
<b>Method</b>	POST
<b>Request body</b>	{ "graphId": string }
<b>Success response</b>	HTTP 200 with text response: "Success message"
<b>Error response</b>	Forwarded from WP4

- Get Deploy app graph state

Emits status of current status of the deploy

*Table 7 - VW/API- deploy status*

<b>Path</b>	<b>/deployment-status</b>
<b>Method</b>	Server Side Event
<b>Request body</b>	-
<b>Success response</b>	Message Event with data: { id: string, data: { stepName: string, success: boolean, message: string, isLastStep: boolean, applicationId: string, branch: string } }
<b>Error response</b>	Message Event with error message.

- Get last status of Deploy app graph state

Get last status of status of the deploy graph

Table 8 - VW/API- last state of deploy status

Path	/deployment-status/last-status/:branchName
Method	GET
Request body	-
Success response	HTTP 200 with text response: [ { id: string, data: { stepName: string, success: boolean, message: string, isLastStep: boolean, applicationId: string, branch: string } } ]
Error response	Message Event with error message.

➤ 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 9 - 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 10 - VW/API- get all created graphs

Path	/graph
Method	GET
Request body	-
Success response	HTTP 200 with response body: [ { "id": string, "flows": {



	<pre>         "flow": Flow,         "url": string       }     }   ] </pre>
<b>Error response</b>	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 11 - VW/API- get all graph drafts*

<b>Path</b>	<b>/graph/draft</b>
<b>Method</b>	GET
<b>Request body</b>	-
<b>Success response</b>	HTTP 200 with response body: <pre> {   "builtFlows": {     "flow": Flow,     "url": string   }[],   "unbuiltFlows": Flow[] } </pre>
<b>Error response</b>	HTTP 404 in case of any connection problems

➤ Get Functions

Returns all available functions' names.

*Table 12 - VW/API get functions*

<b>Path</b>	<b>/function</b>
<b>Method</b>	GET
<b>Request body</b>	-
<b>Success response</b>	HTTP 200 with response body: string[] as function names
<b>Error response</b>	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 13 - VW/API- invoke functions*

<b>Path</b>	<b>/function/invoke</b>
<b>Method</b>	POST
<b>Request body</b>	<pre> {   functionName: string,   params: JSON object } </pre>
<b>Success response</b>	HTTP 200 with response body: String as activation id
<b>Error response</b>	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 14 - VW/API- get function activation*

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

➤ Get cluster list

Retrieve all defined external cluster.

*Table 15 - VW/API- get clusters*

Path	/function/clusters
<b>Method</b>	GET
<b>Request body</b>	-
<b>Success response</b>	HTTP 200 with response body: [ { cluster: string, owurl: string, credid: string, remote: boolean } ]
<b>Error response</b>	HTTP 404 in case of any connection problems

➤ Get flow file

Retrieve the requested flowfile

*Table 16 - VW/API- get flow file*

Path	/function/flowfile/:flowFile
<b>Method</b>	GET
<b>Request body</b>	-
<b>Success response</b>	HTTP 200 with response body: { flowfile: string }
<b>Error response</b>	HTTP 404 in case of any connection problems

➤ Start performance pipeline

Request the start of the performance pipeline.

*Table 17 - VW/API- start performance pipeline*

Path	/function/start-performance
<b>Method</b>	POST
<b>Request body</b>	<pre>{   functionName: string,   params: Record&lt;string, string&gt;,   remote: boolean,   owurl: string,   credid: string,,   dockerimagename: string,   cluster: string,   openwiskTest: boolean,   functionMemory: number,   testDuration: number,   testFunctionPayload: string }</pre>
<b>Success response</b>	HTTP 200 with response body: <pre>{   "results": "url_to_jenkins_job" }</pre>
<b>Error response</b>	HTTP 404 in case of any connection problems

- Get performance result

Request the result of the performance pipeline.

*Table 18 - VW/API- get performance result*

Path	/function/performance-result
<b>Method</b>	POST
<b>Request body</b>	<pre>{   jobUrl: string }</pre>
<b>Success response</b>	HTTP 200 with response body: <pre>{   activation: string,   result: string }</pre>
<b>Error response</b>	HTTP 404 in case of any connection problems

- Get performance pipeline status

Emits status of current Jenkins jobs

*Table 19 - VW/API- performance pipeline status*

Path	/performance-pipeline-status
<b>Method</b>	Server Side Event
<b>Request body</b>	-
<b>Success response</b>	Message Event with data: <pre>{   id: string,   status: {</pre>

	<pre> state: string, url: string, progress: number } } </pre>
<b>Error response</b>	Message Event with error message.

- Get all performance pipeline status

Retrieve all running performance pipeline status for the local user.

*Table 20 - VW/API- get all performance pipeline status*

Path	<b>/performance-pipeline-status/getAll</b>
<b>Method</b>	GET
<b>Request body</b>	-
<b>Success response</b>	Message Event with data: <pre> [   {     id: string,     status: {       state: string,       url: string,       progress: number     }   } ] </pre>
<b>Error response</b>	Message Event with error message.

- Get list of imported image by user

Get the list of the user imported image with the Keycloak Bearer token, in the “Authorization” header.

*Table 21 - VW/API- get imported image*

Path	<b>/import-image</b>
<b>Method</b>	GET
<b>Request body</b>	
<b>Success response</b>	HTTP 200 with response body: <pre> {   "success": boolean,   "value": [     {       _id: string,       user: string,       buildInfo: BuildInfoDto,       buildDate: Date,       imageStatus?: ImageStatus,       jobUrl?: string,       oldaction?: string     }   ] } </pre>

<b>Error response</b>	HTTP 404 in case of any connection problems
-----------------------	---

- Start request for import image

Start the imported image pipeline with the Keycloak Bearer token, in the “Authorization” header.

*Table 22 - VW/API- Import image*

<b>Path</b>	<b>/import-image/:branchName</b>
<b>Method</b>	POST
<b>Request body</b>	<pre>{   user: string,   buildInfo: BuildInfoDto,   buildDate: Date,   imageStatus?: ImageStatus,   jobUrl?: string,   oldaction?: string }</pre>
<b>Success response</b>	HTTP 200 with response body: <pre>{   "success": boolean,   "value": [     {       _id: string,       user: string,       buildInfo: BuildInfoDto,       buildDate: Date,       imageStatus?: ImageStatus,       jobUrl?: string,       oldaction?: string     }   ] }</pre>
<b>Error response</b>	HTTP 404 in case of any connection problems

- Get docker image

Check if the image requested by the user is already present with the Keycloak Bearer token, in the “Authorization” header.

*Table 23 - VW/API- Check if import image already exists*

<b>Path</b>	<b>/import-image/docker-image?dockerImage=string&amp;version=string</b>
<b>Method</b>	GET
<b>Request body</b>	-
<b>Success response</b>	HTTP 200 with response body: <pre>{   registry: string,   public: string,   repo: string,   credid: string,   credDescription: string, }</pre>

	<pre> dockerimage: string, version: string, actionname: string, externalregistry: string } </pre>
<b>Error response</b>	HTTP 404 in case of any connection problems

- Get all imported image

Get all imported image.

*Table 24 - VW/API - Get all imported images*

<b>Path</b>	<b>/import-image/all</b>
<b>Method</b>	GET
<b>Request body</b>	-
<b>Success response</b>	HTTP 200 with response body: <pre> [   {     _id: string,     user: string,     buildInfo: BuildInfoDto,     buildDate: Date,     imageStatus?: ImageStatus,     jobUrl?: string,     oldaction?: string   } ] </pre>
<b>Error response</b>	HTTP 404 in case of any connection problems

- Get jenkins credentials id created by the user

Get all created Jenkins credentials id by the user with the Keycloak Bearer token, in the “Authorization” header.

*Table 25 - VW/API - Get credentials id*

<b>Path</b>	<b>/import-image/cred-id</b>
<b>Method</b>	GET
<b>Request body</b>	-
<b>Success response</b>	HTTP 200 with response body: <pre> [   {     id: string,     description: string   } ] </pre>
<b>Error response</b>	HTTP 404 in case of any connection problems

- Create jenkins credentials id

Create Jenkins credentials id with the Keycloak Bearer token, in the “Authorization” header.

*Table 26 - VW/API - Create credentials id*

Path	/import-image/cred-id/:userName
<b>Method</b>	POST
<b>Request body</b>	{ description: string, username: string, password: string }
<b>Success response</b>	HTTP 200 with response body: { id: string }
<b>Error response</b>	HTTP 404 in case of any connection problems

- Get import image status

Emits status of current Jenkins jobs.

*Table 27 - VW/API- import image status*

Path	/import-image-status
<b>Method</b>	Server Side Event
<b>Request body</b>	-
<b>Success response</b>	Message Event with data: { imgId: string, status: { state: string, url: string, progress: number } }
<b>Error response</b>	Message Event with error message.

- Export Node-RED subflow

Generate a downloadable package of the subflow.

*Table 28 - VW/API- export subflow*

Path	/npm-packages/:flowId
<b>Method</b>	Server Side Event
<b>Request body</b>	{ name: string, version: string, color: string, category: string }
<b>Success response</b>	HTTP 200 with response body: { success: boolean, value: { msg: string }

	<pre> }, file: {   name: string,   data: Buffer } } </pre>
<b>Error response</b>	HTTP 404 in case of any connection problems

#### Artifact Query Service API

##### ➤ Create Artifact

Create new artifact based on incoming flow.

*Table 29 - VW/API- create artifact*

<b>Path</b>	<b>/artifact</b>
<b>Method</b>	POST
<b>Request body</b>	<pre> {   flows: Flow[],   logClientId: string } </pre>
<b>Success response</b>	HTTP 200 with response body: <pre> [   {     flow: Flow,     label?: string,     url?: string   } ] </pre>
<b>Error response</b>	HTTP 404 in case of any connection problems

##### ➤ Get Artifact

Return artifact from database for given flowfile.

*Table 30 - VW/API- get artifact*

<b>Path</b>	<b>/artifact/:flowFile</b>
<b>Method</b>	GET
<b>Request body</b>	-
<b>Success response</b>	HTTP 200 with response body: <pre> [   {     flow: Flow,     label?: string,     url?: string   } ] </pre>
<b>Error response</b>	HTTP 404 in case of any connection problems

##### ➤ Delete Artifact

Delete artifact from database for given data.



*Table 31 - VW/API- delete artifact*

Path	/artifact
<b>Method</b>	DELETE
<b>Request body</b>	{ documentId: string, logClientId: string }
<b>Success response</b>	HTTP 200 with response body: Number of elements removed
<b>Error response</b>	HTTP 404 in case of any connection problems

Graph Draft Service API

## ➤ Get Draft

Returns all graph drafts from database.

*Table 32 - VW/API- get draft service*

Path	/draft
<b>Method</b>	GET
<b>Request body</b>	-
<b>Success response</b>	HTTP 200 with response body: { "builtFlows": { "flow": Flow, "url": string }[], "unbuiltFlows": Flow[] }
<b>Error response</b>	HTTP 404 in case of any connection problems

## ➤ Create Drafts

Create a graph draft from database.

*Table 33 - VW/API- create graph draft*

Path	/draft
<b>Method</b>	POST
<b>Request body</b>	{ builtFlows: Artifact[], unbuiltFlows: Flow[] }
<b>Success response</b>	HTTP 200 without response body
<b>Error response</b>	HTTP 404 in case of any connection problems

Function Service API

## ➤ Get Functions

Returns all available functions' names.

*Table 34 - VW/API- get function service*

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

➤ **Invoke Functions**

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

*Table 35 - VW/API- invoke function*

Path	/function/invoke
<b>Method</b>	POST
<b>Request body</b>	{ functionName: string, params: JSON object }
<b>Success response</b>	HTTP 200 with response body: String as activation id
<b>Error response</b>	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 36 - VW/API- get activationID*

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

Cluster Service API

➤ **Get Clusters**

Returns all available clusters.

*Table 37 - VW/API- get all clusters service*

Path	/cluster
<b>Method</b>	GET
<b>Request body</b>	-
<b>Success response</b>	HTTP 200 with response body: [ { cluster: string, owurl: string, credid: string, remote: boolean }

	<pre>         }       ]     </pre>
<b>Error response</b>	HTTP 404 in case of any connection problems

#### Import Image API

- Get list of imported images by user

Retrieve the imported image on db. The request is protected with the Keycloak Bearer token, in the “Authorization” header.

*Table 38 - VW/API- get imported image*

<b>Path</b>	<b>/import-image</b>
<b>Method</b>	GET
<b>Request body</b>	
<b>Success response</b>	HTTP 200 with response body: <pre> [   {     _id: string,     user: string,     buildInfo: BuildInfoDto,     buildDate: Date,     imageStatus?: ImageStatus,     jobUrl?: string,     oldaction?: string   } ]         </pre>
<b>Error response</b>	HTTP 404 in case of any connection problems

- Insert import image

Insert the imported image on db. The request is protected with the Keycloak Bearer token, in the “Authorization” header.

*Table 39 - VW/API- Insert import image*

<b>Path</b>	<b>/import-image</b>
<b>Method</b>	POST
<b>Request body</b>	<pre> {   user: string,   buildInfo: BuildInfoDto,   buildDate: Date,   imageStatus?: ImageStatus,   jobUrl?: string,   oldaction?: string }         </pre>
<b>Success response</b>	HTTP 200 with response body: <pre> {   _id: string,   user: string,   buildInfo: BuildInfoDto,   buildDate: Date,   imageStatus?: ImageStatus, }         </pre>

	<pre> jobUrl?: string, oldaction?: string } </pre>
<b>Error response</b>	HTTP 404 in case of any connection problems

➤ Update import image

Update the imported image on db. The request is protected with the Keycloak Bearer token, in the "Authorization" header.

*Table 40 - VW/API- Update import image*

<b>Path</b>	<b>/import-image/:id</b>
<b>Method</b>	PUT
<b>Request body</b>	<pre> {   user: string,   buildInfo: BuildInfoDto,   buildDate: Date,   imageStatus?: ImageStatus,   jobUrl?: string,   oldaction?: string } </pre>
<b>Success response</b>	HTTP 200 if successfully updated
<b>Error response</b>	HTTP 404 in case of any connection problems

➤ Get docker image

Check if the image requested by the user is already present with the Keycloak Bearer token, in the "Authorization" header.

*Table 41 - VW/API- Check if import image already exists*

<b>Path</b>	<b>/import-image/docker-image?dockerImage=string&amp;version=string</b>
<b>Method</b>	GET
<b>Request body</b>	-
<b>Success response</b>	<p>HTTP 200 with response body:</p> <pre> {   registry: string,   public: string,   repo: string,   credid: string,   credDescription: string,   dockerimage: string,   version: string,   actionname: string,   externalregistry: string } </pre>
<b>Error response</b>	HTTP 404 in case of any connection problems

➤ Get all imported image

Get all imported images.

Table 42 - VW/API - Get all imported images

Path	/import-image/all
Method	GET
Request body	-
Success response	HTTP 200 with response body: [ { _id: string, user: string, buildInfo: BuildInfoDto, buildDate: Date, imageStatus?: ImageStatus, jobUrl?: string, oldaction?: string } ]
Error response	HTTP 404 in case of any connection problems

- Get jenkins credentials id created by the user

Get all created jenkins credentials id by the user with the Keycloak Bearer token, in the “Authorization” header.

Table 43 - VW/API - Get credentials id

Path	/import-image/cred-id
Method	GET
Request body	-
Success response	HTTP 200 with response body: [ { id: string, description: string } ]
Error response	HTTP 404 in case of any connection problems

### 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

### Deployment status API

Deployment status exposes a WebSocket with a query branch; for each WS connection the application creates a queue on RabbitMQ bound to the exchange **deployment-status** and sends back the queue object by the the tag branch in rabbitMQ object. The created queue will be closed when the WS connection is ended.

```
{
    stepName: string,
    success: boolean,
    message: string,
    isLastStep: boolean,
    applicationId: string,
    branch: string
}
```

### Artifact Processor API

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

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

### Graph Processor API

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

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

### 3.1.4 Distribution, deployment and configuration

All subcomponents are prepared as separated docker images. Artifact Query Service, Graph Draft Service, Function Service, Build Result Processor, Artifact Processor, Graph Processor, Import Image and Cluster Service 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 utilizing docker-compose files.

### 3.1.5 Control UI

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.

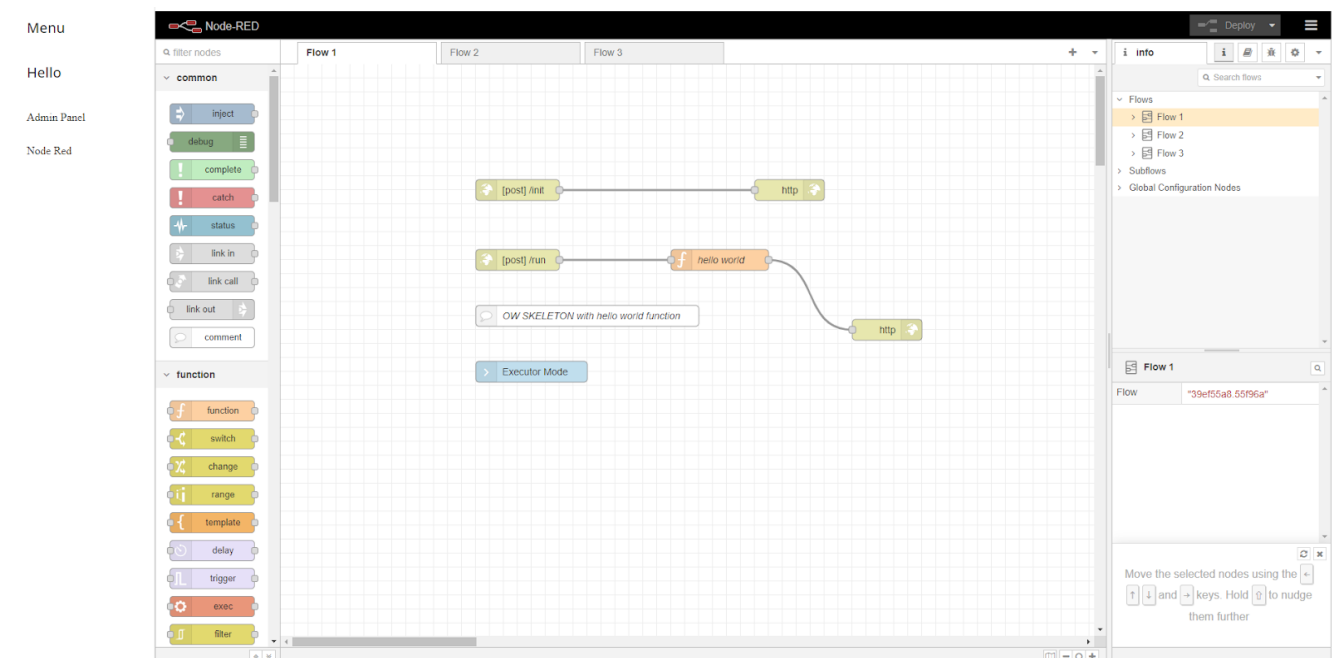


Figure 9 - Node-RED environment

➤ Build tab in Admin Panel

Dialog where developers can choose flow to build artifacts.

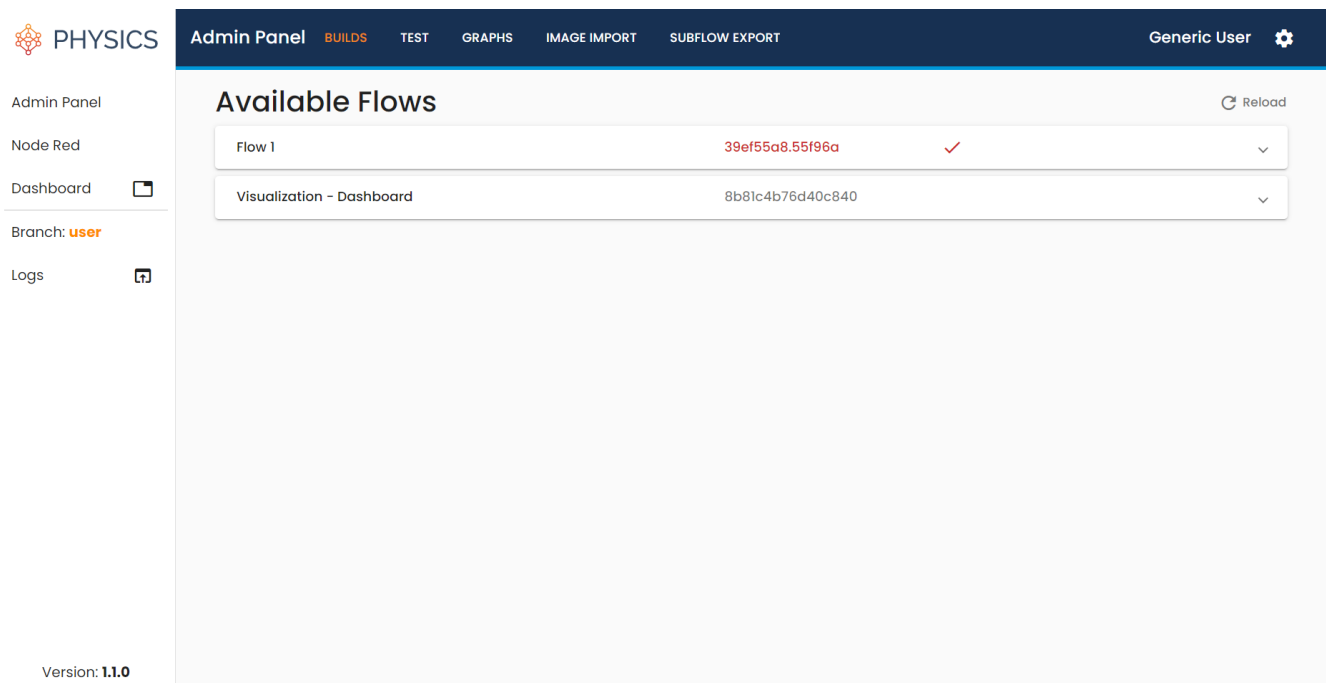
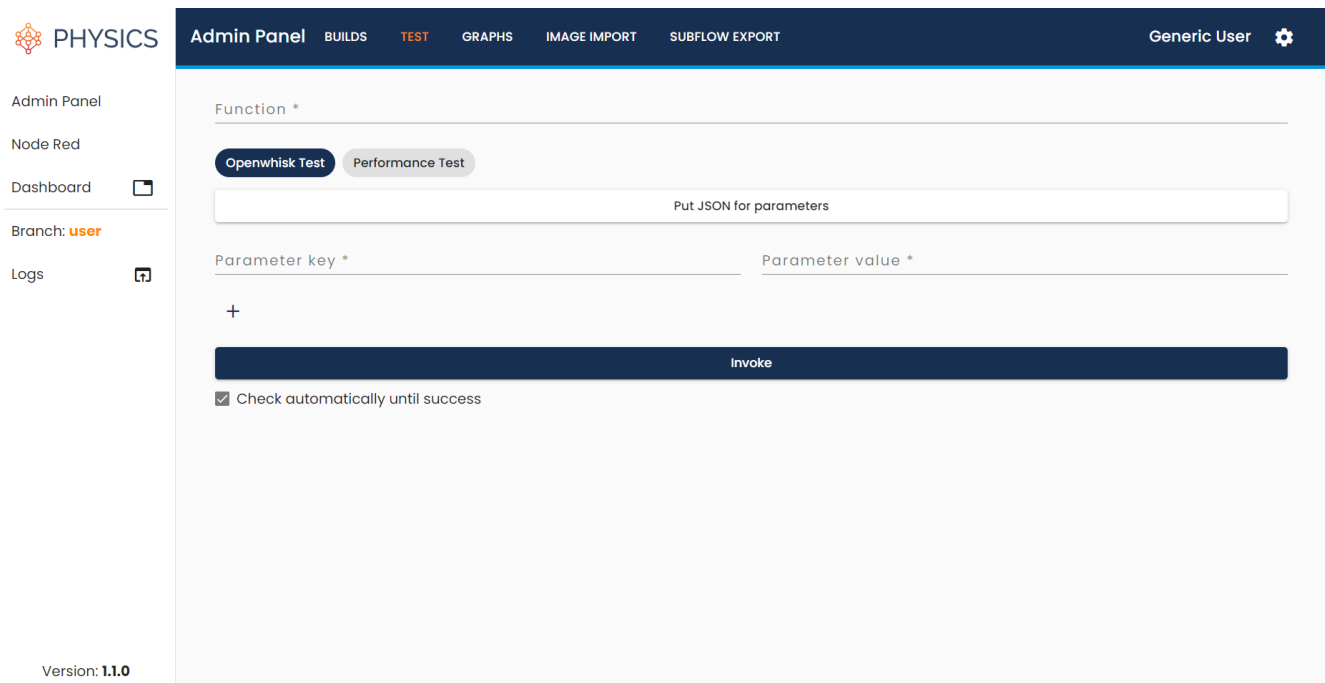


Figure 10 - Build flow

➤ Test tab in Admin Panel

Dialog where developers can test flows deployed to the test OpenWhisk environment or request a performance test on local FaaS platforms or on a remote cluster

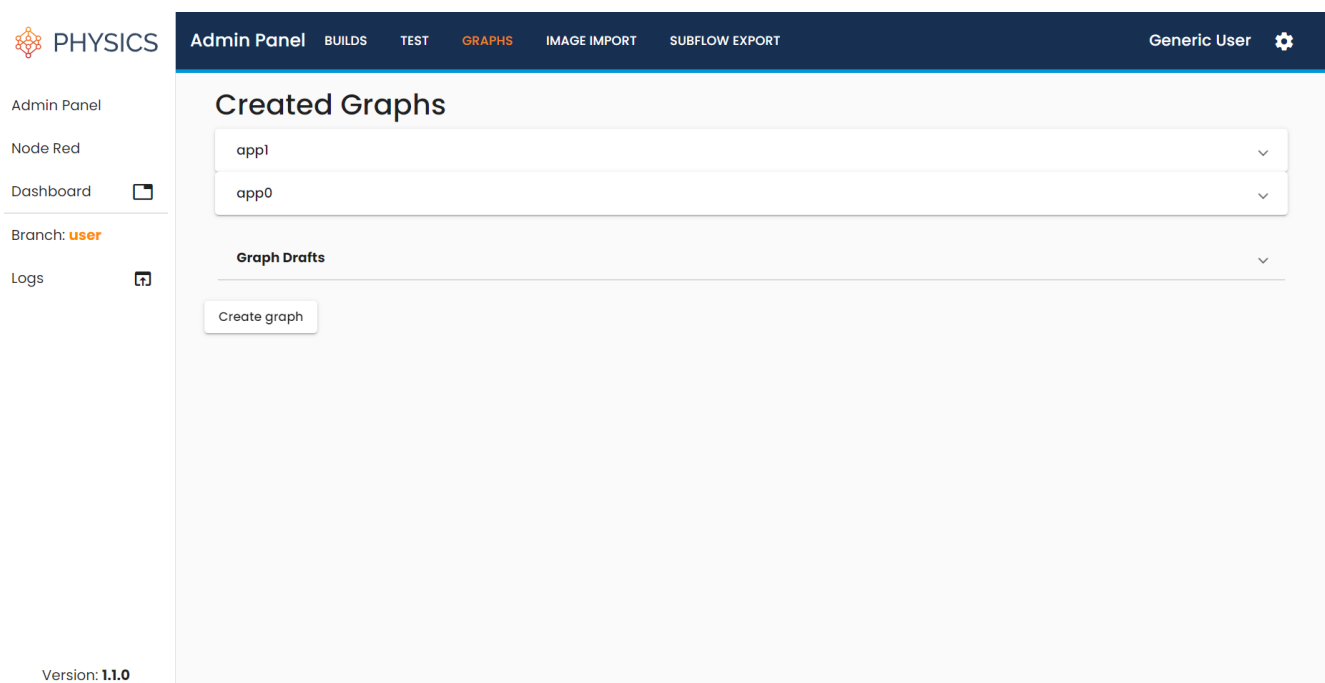


The screenshot shows the PHYSICS Admin Panel with the 'TEST' tab selected. The interface includes a sidebar with 'Admin Panel', 'Node Red', 'Dashboard', 'Branch: user', and 'Logs'. The main content area has a top navigation bar with 'Admin Panel', 'BUILDS', 'TEST', 'GRAPHS', 'IMAGE IMPORT', and 'SUBFLOW EXPORT'. The 'Generic User' profile is shown in the top right. The 'Function \*' section has two tabs: 'Openwhisk Test' (active) and 'Performance Test'. Below the tabs is a text input field labeled 'Put JSON for parameters'. The 'Parameter key \*' and 'Parameter value \*' fields are empty, with a '+' icon below them. A large blue 'Invoke' button is present. A checkbox labeled 'Check automatically until success' is checked. The version '1.1.0' is displayed at the bottom left.

*Figure 11 - Test flow*

➤ Graph tab in Admin Panel

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



The screenshot shows the PHYSICS Admin Panel with the 'GRAPHS' tab selected. The sidebar is identical to the previous figure. The main content area has a top navigation bar with 'Admin Panel', 'BUILDS', 'TEST', 'GRAPHS', 'IMAGE IMPORT', and 'SUBFLOW EXPORT'. The 'Generic User' profile is shown in the top right. The 'Created Graphs' section lists two graphs: 'app1' and 'app0', each with a dropdown arrow. Below this is the 'Graph Drafts' section, which is currently empty. A 'Create graph' button is located at the bottom left of the main content area. The version '1.1.0' is displayed at the bottom left.

*Figure 12 - See created and draft graphs*

Developers can also create new graphs here.



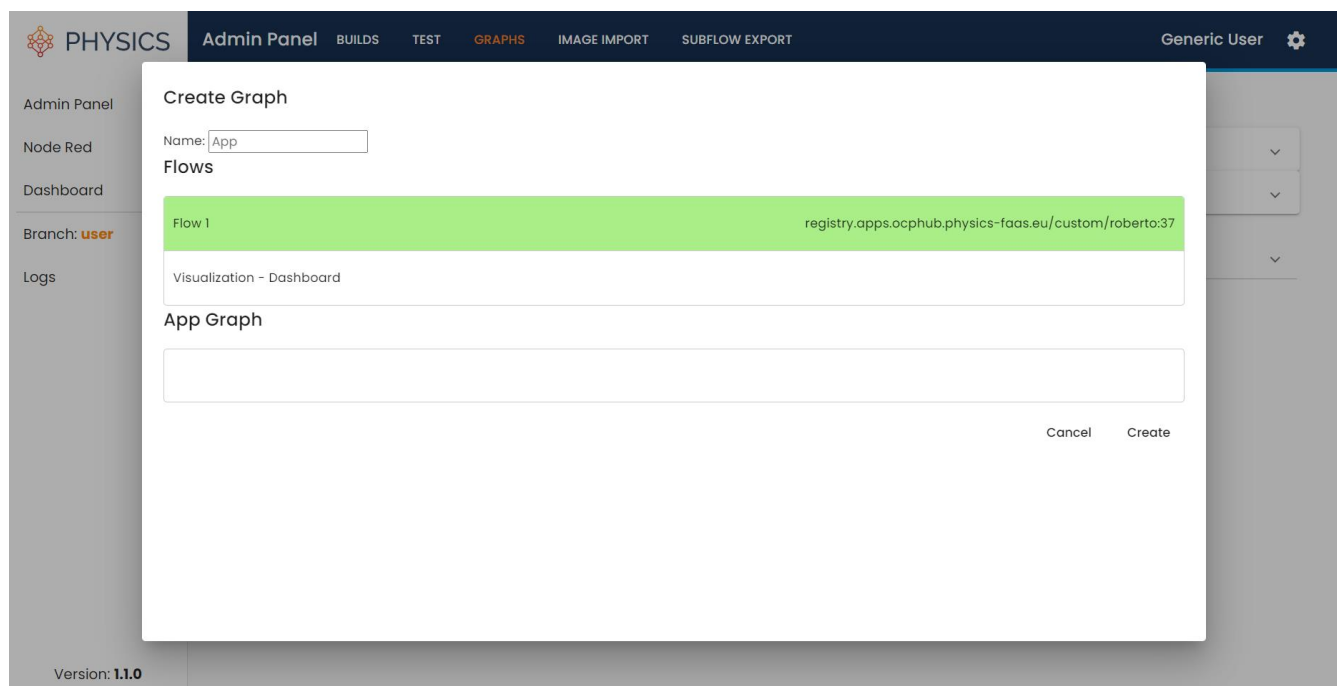


Figure 13 - Create a new graph

### ➤ Import image in Admin Panel

Where the user can view all requested import image and request a new one

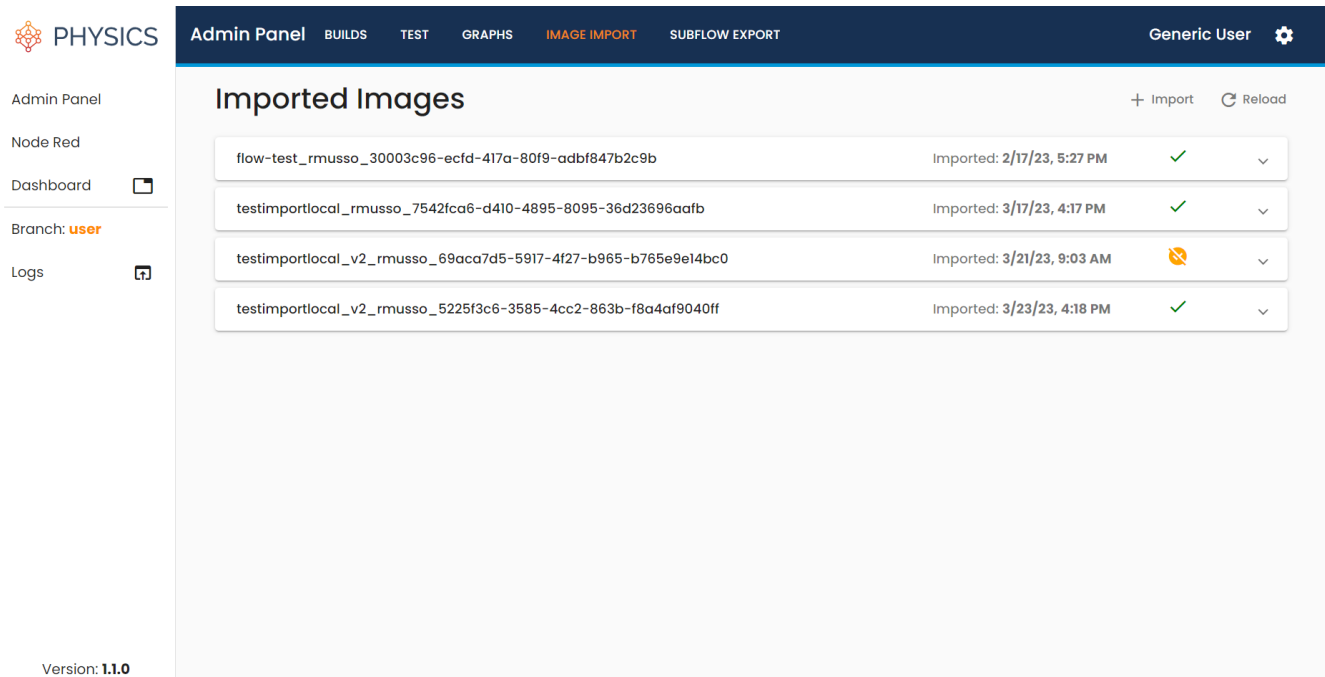


Figure 14 - Import image

### ➤ Export subflow in Admin Panel

Where the user can export the local subflow created on Node-RED

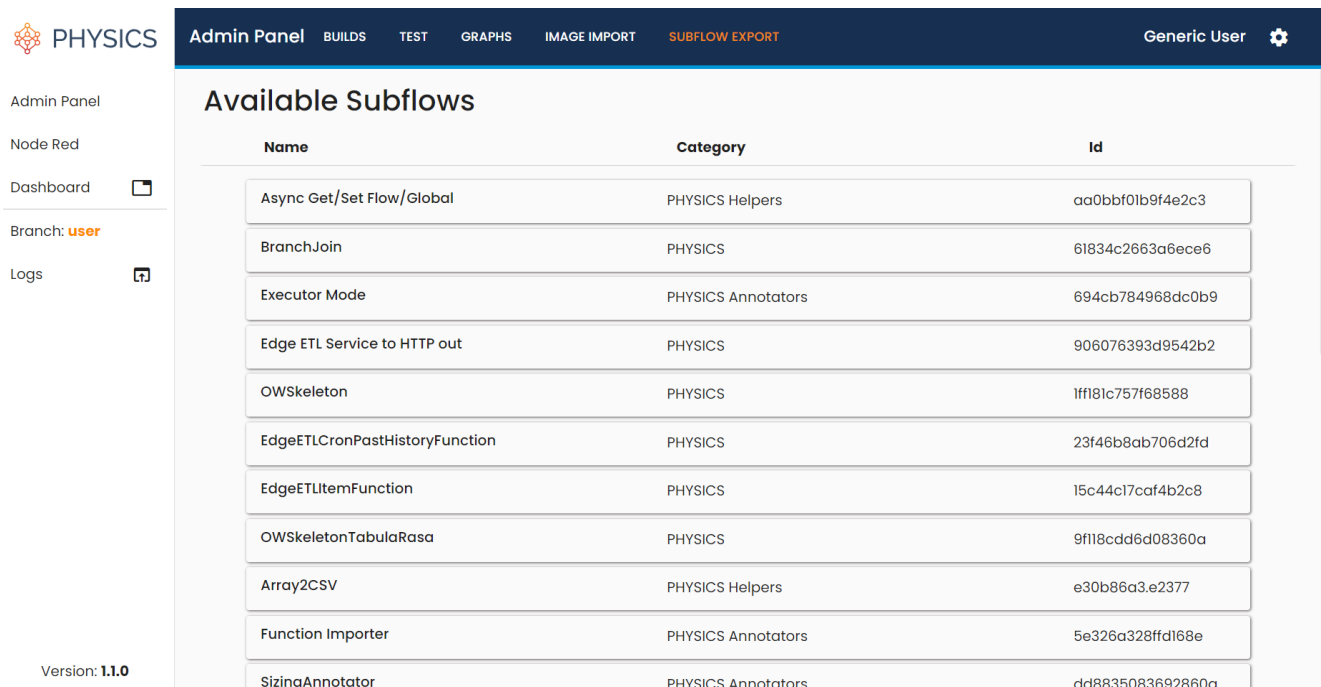


Figure 15 - Export subflow

## ➤ Dashboard

Where the user can see the performance of local running flow on Node-RED

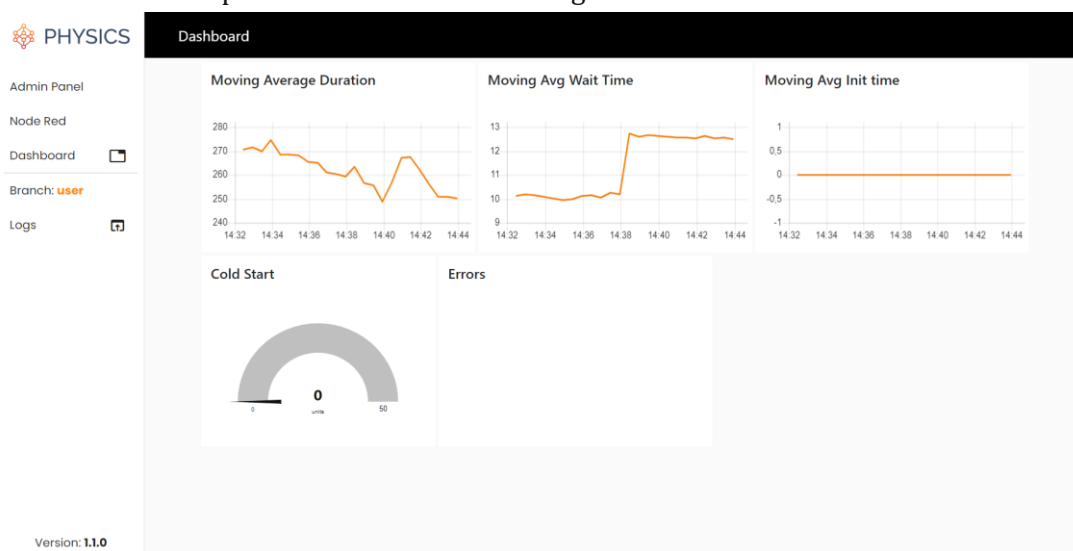
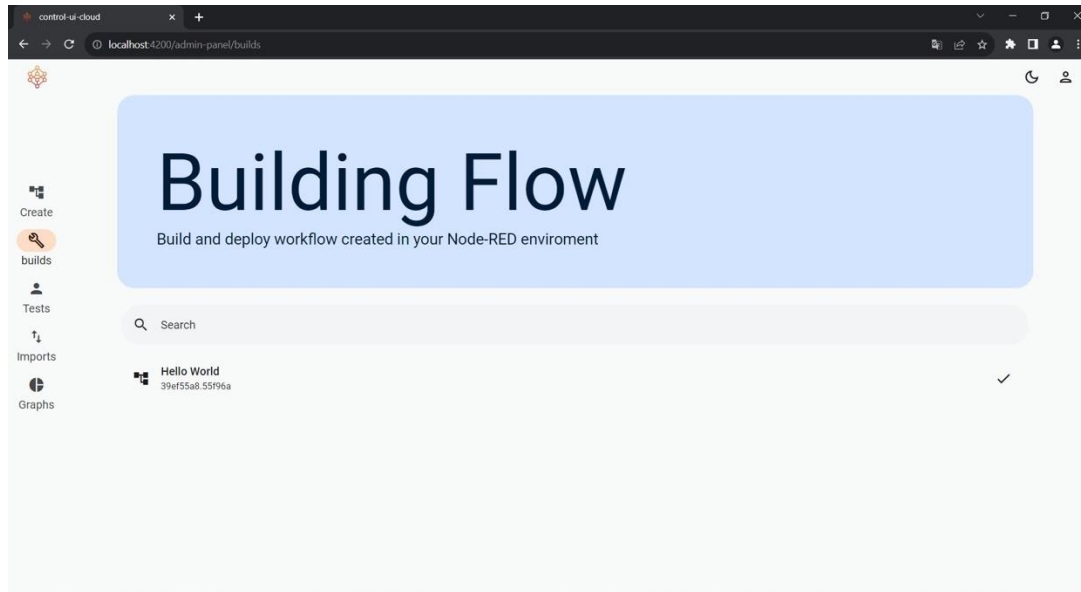


Figure 16 - Dashboard

### 3.1.6 Cloud version



*Figure 17 - Control UI cloud*

With the aim of better usability, portability, accessibility and exploitation, the Control UI was redesigned in a full cloud version architecture. The main features are still the same but the UI is one for all the users as the backend, which is redesigned to be a Nestjs GraphQL compliance application deployed on Apollo server. The resources, described in the following SDL schema, are now available on “/graphql” in POST and it was secured with a JWT Token released by the securing application of the FaaS Physics environment.

```

type JenkinsJobState {
  state: String!
  url: String
  result: String
  progress: Int
  lastUpdate: DateTime
}

# A date-time string at UTC, such as 2019-12-03T09:54:33Z, compliant with the date-time format.
scalar DateTime

type UserCreation {
  username: String!
  isUserRegistered: Boolean!
  isUserInBranch: Boolean
}

type FlowNode {
  id: String!
  name: String
  type: String!
  label: String
  info: String
  category: String!
  wires: [[String!]!]
  func: String
  libs: [Lib!]
  z: String

```

```

    subflow: Boolean
    nodeRed: String
}

type ArtifactInformation {
    _id: String!
    url: String!
    flowFile: String
    timestamp: DateTime!
}

# Node-red flow with artifact informations
type FlowWithBuiltInformation {
    id: String!
    flow: FlowNode!
    artifact: [ArtifactInformation!]
    state: JenkinsJobState
    nodeRed: Boolean
}

type Lib {
    var: String!
    module: String!
}

type SubflowInfo {
    id: String!
    name: String
    info: String
    category: String
}

type Cluster {
    _id: String
    cluster: String!
    owurl: String!
    credid: String!
    remote: Boolean!
}

type DePod {
    id: String!
    username: String!
    jenkinsStatus: JenkinsJobState
}

type DePodStatus {
    AGE: String
    NAME: String
    READY: String
    UP TO DATE: String
    AVAILABLE: String
}

type PerformanceTest {
    cluster: Cluster!
    functionMemory: String
    testDuration: String
    activation: String
    jenkinsStatus: JenkinsJobState
}

```

```

type FunctionTested {
    id: String
    username: String!
    functionName: String!
    testType: String!
    activationId: String
    timeStart: DateTime!
    params: String!
    result: String
    success: Boolean
    performance: PerformanceTest
}

type ImportImageFormData {
    actionname: String!
    externalregistry: Boolean!
    registry: String!
    public: Boolean!
    repo: String!
    dockerimage: String!
    useCredential: Boolean!
    version: String!
    jenkinsCredentialId: String!
}

type ImportImage {
    id: String
    username: String!
    formData: ImportImageFormData!
    buildDate: DateTime!
    oldaction: String
    jenkinsStatus: JenkinsJobState
}

type JenkinsCredentials {
    id: String!
    username: String!
    jenkinsCredentialId: String!
    label: String!
}

type Query {
    # Retrieves user POD based on the JWT token
    dePodByUser: DePod

    # Retrieve the status of the user POD on the cluster based on the JWT token
    dePodStatus: DePodStatus!

    # Retrieve if the user based on keycloak JWT had already make a login on gogs
    isRegistered: UserCreation!

    # Retrieves user flows based on keycloak JWT from Node-RED environment and returns it with
    build information from database
    flows: [FlowWithBuiltInformation!]!

    # Retrieves user subflows based on keycloak JWT from Node-RED environment
    subFlows: [SubflowInfo!]!

    # Check if the username in input is available on keycloak
    isUsernameAvailable(username: String!): Boolean!
}

```

```

# Check if the email in input is available on keycloak
isEmailAvailable(email: String!): Boolean!

# Check if the password in input, based on user in keycloak JWT is correct
isValidPassword(password: String!): Boolean!

# Retrieves users test based on keycloak JWT
functionsTested: [FunctionTested!]!

# Returns all available functions' names
functions: [String!]!

# Retrieve all defined external cluster
clusters: [Cluster!]!

# Get the list of the user's imported images based on keycloak JWT imported image
importImages: [ImportImage!]!

# Get the all the configured user's jenkins credetntials
getCredentials: [JenkinsCredentials!]!

# Get a specific configured user's jenkins credetntials by the id
getCredential(id: ID!): JenkinsCredentials!

# Get the URL for the download of the USER GUIDE
getUserGuide: String!
}

type Mutation {
  # Start the user POD based on the JWT token
  startDePod: Boolean!

  # Stop the user POD based on the JWT token
  stopDePod: Boolean!

  # Request the creation of the user POD by starting the pipeline for the creation
  createDePod(
    # The user password needed for start the pipeline for the POD creation
    password: String!
  ): DePod!

  # Receives flowId as input, triggers Jenkins build for given flow and returns URL to Jenkins
  job
  buildFlow(flowId: String!): String!

  # Delete a specific artifact
  removeArtifact(artifactId: String!): String!

  # Create a user on keycloak
  createUser(keycloakUserInput: KeycloakUserInput!): Boolean!

  # Receives function test input, invoke the function and returns the function input test data of
  the stared test
  invoke(invokeInput: InvokeInput!): FunctionTested!

  # Delete function tested by the id in input
  deleteFunctionTest(id: String!): FunctionTested!

  # Start the pipeline for importing image
  invokeImportImage(importImageDataInput: ImportImageDataInput!): ImportImage!

```

```

}

input KeycloakUserInput {
  firstName: String!
  lastName: String!
  email: String!
  enabled: Boolean
  username: String!
  groups: [String!]
  emailVerified: Boolean
  credentials: [Credentials!]
  attributes: Attributes
}

input Credentials {
  type: String!
  value: String!
  temporary: Boolean!
}

input Attributes {
  uidNumber: String!
  homeDirectory: String!
  gidNumber: String!
}

input InvokeInput {
  functionName: String!
  params: String!
  testType: String!
  clusters: String!
  testDuration: String!
  functionMemory: String!
}

input ImportImageDataInput {
  actionname: String!
  externalregistry: Boolean!
  public: Boolean!
  registry: String!
  repo: String!
  dockerimage: String!
  version: String!
  useCredential: Boolean!
  oldaction: String
  jenkinsCredentialId: String
  jenkinsLabel: String
  jenkinsUser: String
  jenkinsPwd: String
}

```

## 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.

In the final version of the component, it was updated in order to reflect and include new additions in terms of semantic nodes, following the generalized process described in D3.2, as well as incorporate inside the application graph details on a function performance. This relates to performance statistics relevant to the benchmarking of the function to one or more available clusters, as well as the profiling of the function usage on the underlying resources. These two advancements were dictated by the need to integrate the Performance Pipeline described in D3.2, through which the developer can investigate performance issues of a given function. This information is then included in the app graph in order to enable better selection of the candidate clusters for deployment as well as better co-allocation of function execution on the available nodes.

### 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 minimizing 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<sup>3</sup> and jsonld.js<sup>4</sup> 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<sup>5</sup>, validate.io<sup>6</sup>, clean-deep<sup>7</sup> and json-schema-library<sup>8</sup>, 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. The Semantic Extractor is a stateless service, i.e. it does not need a database or any state to be associated with it. Any relevant information is retrieved from external databases (e.g. the Performance database of PEF). Hence it can be easily scaled according to the needs for translation between incoming Node-RED function flows and the according PHYSICS application graph.

### 3.2.3 Interfaces/API

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

*Table 44 - SE-API-semantic retrieval*

Path	/extract
Method	POST
Request body	<pre>-{   "flows":[     "flowID": id of the flow from Node-RED     "flowName": name of the flow from Node-RED     "flow": JSON output of Node-RED flow,     "artefact": URL of the image corresponding to this     function,     "type": either code or image,</pre>

<sup>3</sup> <http://docs.jsonata.org/overview.html>

<sup>4</sup> <https://github.com/digitalbazaar/jsonld.js>

<sup>5</sup> <https://github.com/stdlib-js>

<sup>6</sup> <https://www.npmjs.com/package/validate.io>

<sup>7</sup> <https://www.npmjs.com/package/clean-deep>

<sup>8</sup> <https://www.npmjs.com/package/json-schema-library>



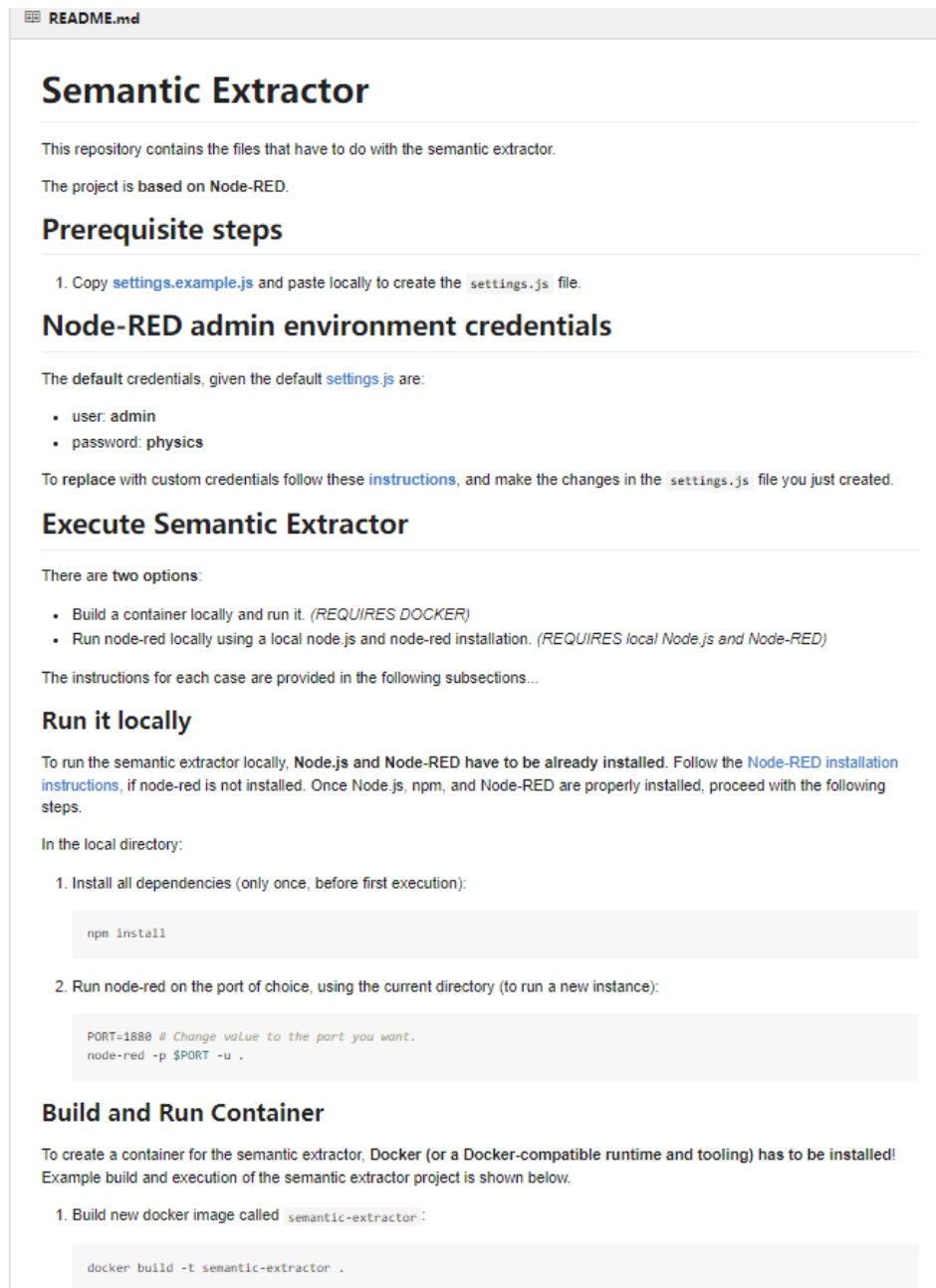
	<pre> ].   "userID/branchID": id of a user or branch,   "appID": optional in case we need to update an app id } </pre>
<b>Success response</b>	HTTP 200 with response body the app ID as generated from the Reasoning Framework if not an update: <pre> {   "appID": "dsdsuJJJaj...." } </pre>
<b>Error response</b>	HTTP 40X

An identical second method (/transform) was added in Y3, so that testing can be performed in an easier manner, without having to forward the test app graphs to the RF.

### 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. 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 includes a source code project in which all the required files are provided (e.g. flows file, settings file, Dockerfile, package.json with package dependencies in node.js etc.). The SE can be executed 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 3.2.2 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. Relevant instructions are provided in the project code repository and appear in the following figure.



*Figure 18 - Build and Execution Process for the Semantic Extractor*

Updating of the relevant flows in the context of PHYSICS is applied through a dedicated Jenkins pipeline.

### 3.2.5 Sample Application Transformation

In this section we provide an example application transformation for the sample application defined in Section 2. The sample application consists of one Hello World function as well as an Orchestrator function that calls dynamically the Hello World one. The corresponding input from the Design Environment app creation appears in Figure 19. The flow tab has all the information as this is extracted from the Node-RED specification of the flow and is hidden for visibility purposes.

```

{
  "flows": [
    {
      "flowID": "39ef55a8.55f96a",
      "flowName": "HelloFunctionV2",
      "flow": [
        {
          "artifact": "registry.apps.ocphub.physics-faas.eu/custom/george:199",
          "type": "image"
        }
      ],
      "flowID": "a714de618203e22e",
      "flowName": "orchestrator",
      "flow": [
        {
          "artifact": "registry.apps.ocphub.physics-faas.eu/custom/george:200",
          "type": "image"
        }
      ]
    },
    {
      "displayName": "p2app",
      "branchName": "george"
    }
  ]
}

```

*Figure 19 - Example DE output towards SE for the sample App*

Given each flow input, the extractor provides the corresponding JSON-LD output, which is JSON-LD that follows the current ontology. This JSON-LD is a serialization format of semantic triples that are inputs for the reasoning engine. In all cases, the different respective annotations used in each flow are mapped to the respective triples, as indicated in Figure 20. 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 semantic 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. The app graph also includes information dynamically retrieved by the SE from the PEF, exploiting the according APIs described in Section 3.10, in order to extract benchmark and profiling information available for the functions included in the app, if these are available.

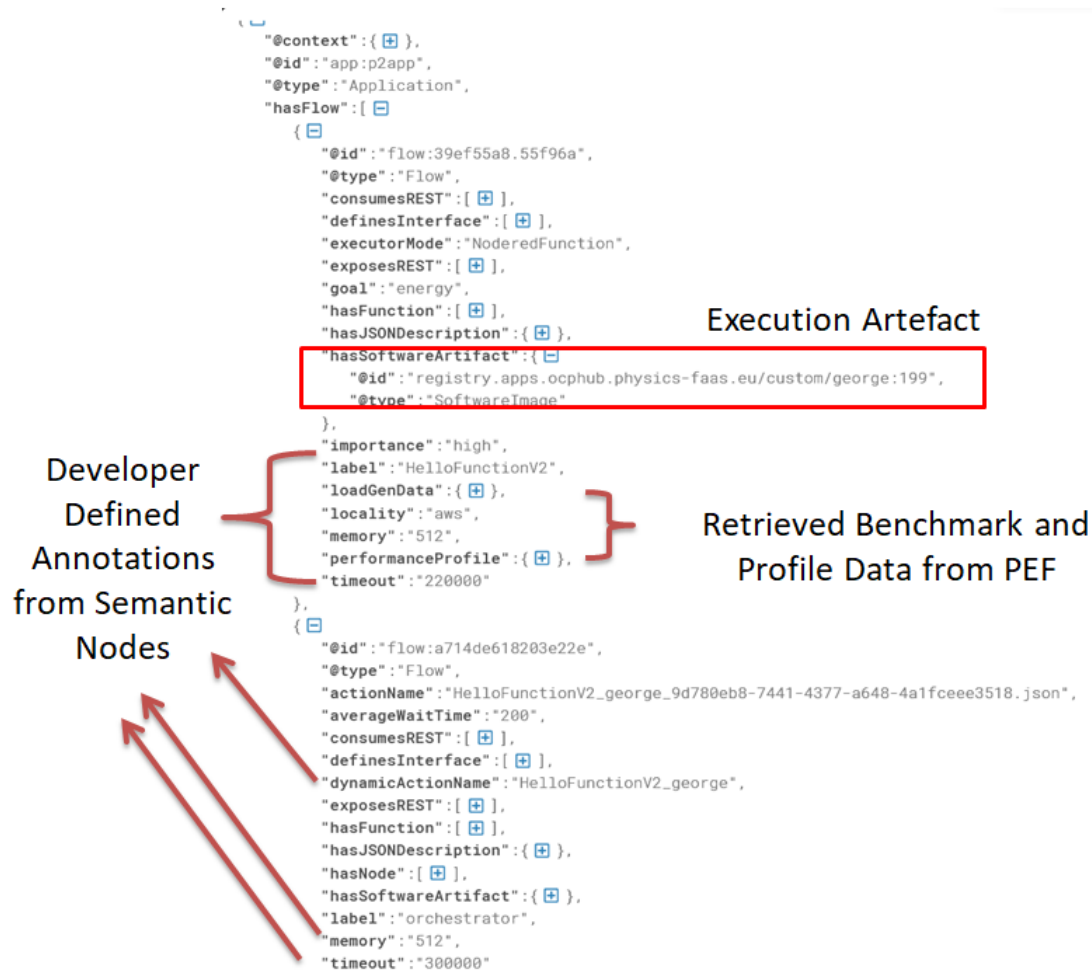


Figure 20 - SE Annotated Output Graph towards RF

### 3.3 Design Patterns

#### 3.3.1 Overview

The Design Patterns aim at offering reusable and parametric operational capabilities to the developers to enable an easier and more abstracted flow creation process. Patterns have been created for workflow enhancement, load distribution, message manipulation etc. In the final version, new patterns have been included to automate aspects such as routing between available endpoints, dynamic orchestration of functions etc.

#### 3.3.2 Technology architecture

The patterns are primarily Node-RED flows and subflows so that they can be integrated directly into the PHYSICS Design Environment. They can be executed as services or as functions, where applicable. 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 D3.2.

### 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 21).

They have also been made available in online repositories (e.g., Node-RED flows repository<sup>9</sup> and the PHYSICS RAMP). 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 22), 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.

---

<sup>9</sup> PHYSICS Patterns Collection, available at: <https://flows.nodered.org/collection/HXSkA2JLcGA>

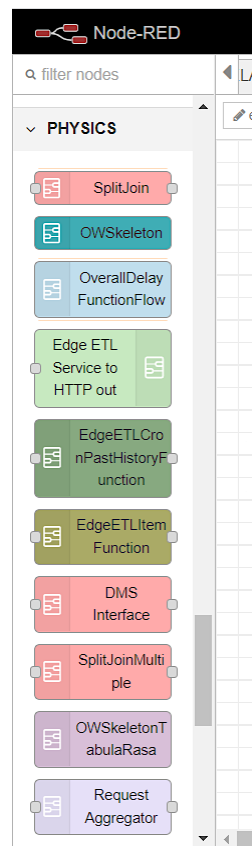


Figure 21 - PHYSICS Patterns Palette in Node-RED

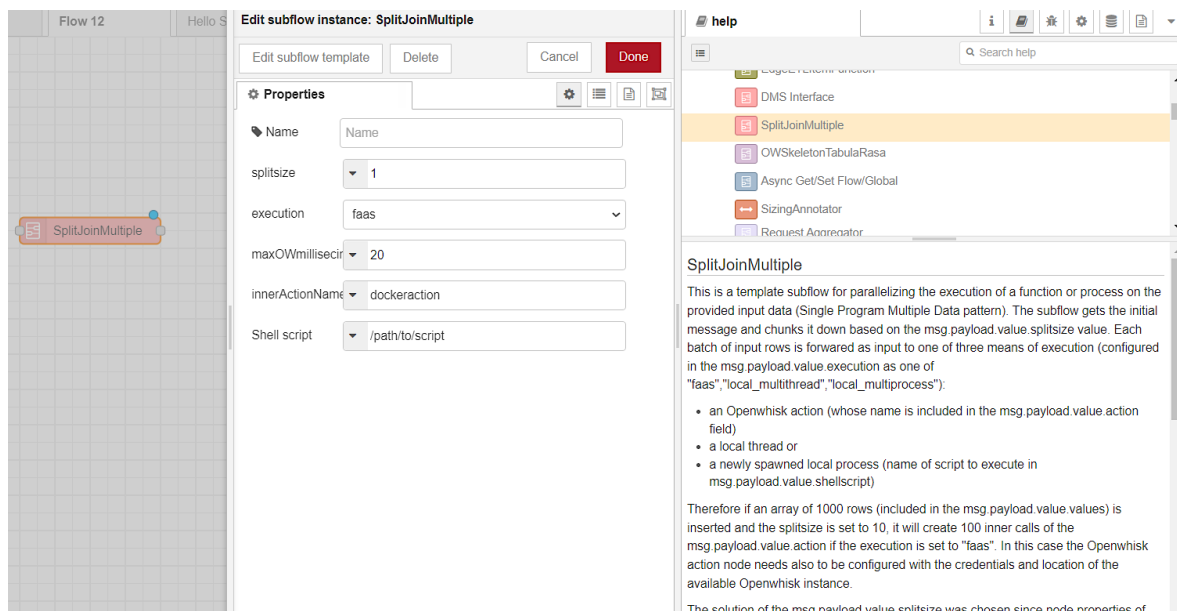


Figure 22 - Example UI configuration and README file in Pattern Node

This type of distribution has the benefit that the developer can also edit the subflow template, meaning that they can change some part of the pattern implementation to better suit their needs. On the other hand, it means that the developer needs to have pre-installed any external dependency (e.g.

a Node-RED node that is not in the default Node-RED environment). For this reason, a second mean of distribution has been investigated and is described in the next paragraphs.

#### Distribution based on npm nodes packaging

A second way for distributing the created subflows is through packaging them into Node-RED modules<sup>10</sup> that are then published in the main NPM repository. This enables a more automated way of installation as well as update of a pattern node, however it removes the freedom from the developer to change any internal details of the node operation. Given that this ability to adapt was a feature that was significantly appreciated by the project Use cases, only indicative subflows were converted to this process (e.g., the security and privacy ones<sup>11</sup>, the monitor<sup>12</sup> and semaphore service). However, one of the Y3 features of the Design Environment includes the ability to export subflows as npm packages, as detailed in D3.2. Thus any interested PHYSICS developer can follow that way of publication, supported also by the DE for its creation. The process is based on the nodegen tool<sup>13</sup>. Before exporting the subflow we need to include information on the description of the node as shown in Figure 23, since a number of these fields are mandatory for the package creation. Finally, a set of files is created that is downloaded from the DE and contains all the main files of our node (README, etc.) as exported from the subflow definition, as well as the code files, that can be installed through a typical npm install command.

The screenshot shows a dialog box titled "Edit subflow template: node-red-contrib-owmonitor". At the top right are "Cancel" and "Done" buttons. Below the title bar is a tabbed interface with "Module Properties" selected. The properties are as follows:

Property	Value
Module	node-red-contrib-owmonitor
Node Type	8262c8e75cfbabdd
Version	0.0.1
Description	This is a subflow node in order to monitor the latest per
License	▼ Apache-2.0
Author	George Kousiours <gkousiou@hua.gr>
Keywords	openwhisk, performance, monitor, sliding window

*Figure 23 - Subflow Description Information for npm node conversion of a subflow*

<sup>10</sup> Subflow modules packaging: <https://nodered.org/docs/creating-nodes/subflow-modules>

<sup>11</sup> Security and privacy modules npm packages: <https://www.npmjs.com/~doth-j>

<sup>12</sup> Monitor npm package: <https://www.npmjs.com/package/node-red-contrib-owmonitor>

<sup>13</sup> Node-RED node generator tool. Available at: <https://github.com/node-red/node-red-nodegen>

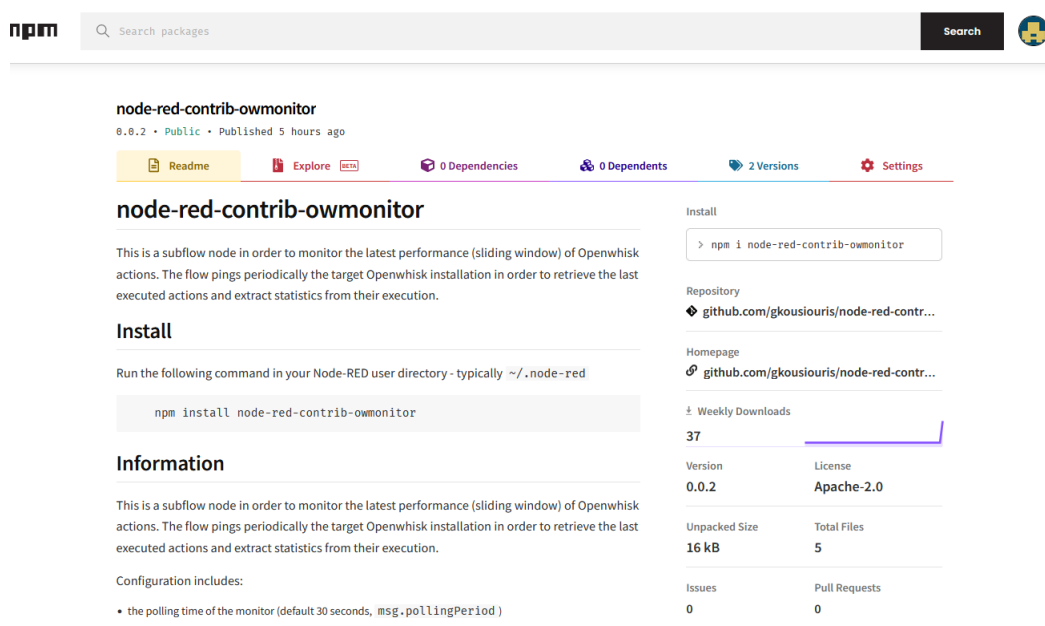
In order to publish in the npm repository, from the previous folder generated by the DE we can push the node on npm, following the inclusion of the desired npm account:

```
npm adduser
```

Then we can publish the node folder contents on npm through the:

```
npm publish
```

The result appears in Figure 24.



*Figure 24 - Example Subflow Node published on npm*

For adding in the Node-RED node repository, in order for it to be also available for installation directly (including all its dependencies) from the Node-RED main palette environment, we also need to register it through the Node-RED site<sup>14</sup> and add the npm module name. This declares the node in the Node-RED community repository (Figure 25).

<sup>14</sup> Node addition in Node-RED repository: <https://flows.nodered.org/add/node>



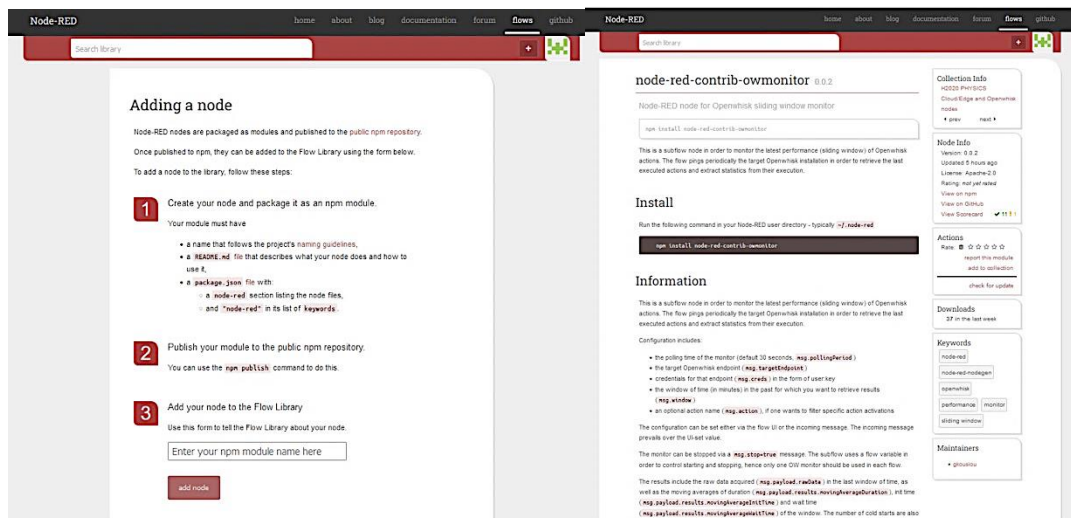


Figure 25 - a) Node Addition Process in Node-RED b) Available Example Node on Node-RED repo

Following that step, the contribution is now packaged as a Node-RED node and can be found directly from the built-in palette management functionality of Node-RED (Figure 26).

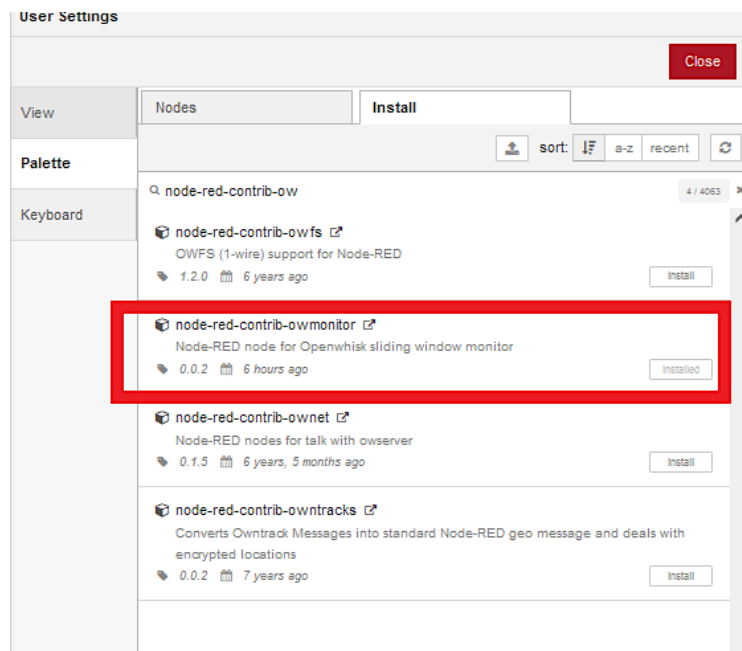


Figure 26 - Direct Installation of Node through the built-in Palette Management of Node-RED

In our case it was also added in a specific collection that is organized by PHYSICS specifically for nodes<sup>15</sup> and is different than the one for the subflows<sup>16</sup>.

<sup>15</sup> PHYSICS node collection on Node-RED repository, available at: <https://flows.nodered.org/collection/9C3h7Hnru943>

<sup>16</sup> PHYSICS Flows collection: <https://flows.nodered.org/collection/HXSkA2JLLcGA>

This way of publishing (NPM and Node-RED node) also allows to keep track of usages, comments etc. for our node. Thus, it is a very good way for reusability of the results as well as statistics of usage of a node artefact. It can also be linked to a GitHub repository, in which the testers or users can interact in the form of issues etc.

### 3.3.5 Flow update process

Provided pattern flows by the PHYSICS project may be periodically updated or extended. These updates may be a result of newly needed features, specific requests (e.g., by the use cases) or debugging and improved parameterization. To import the updated versions in the environment, if the pattern is packaged as a node, this process is done through the palette management of the environment. However, if the pattern is packaged as a subflow or typical flow, then different variations of import may be performed.

Initially the JSON description of the pattern needs to be retrieved from the PHYSICS collection on Node-RED repo and copied (Figure 27).

The current flow targets a test container action (dockeraction) that applies an artificial delay in a loop manner. For that purpose it receives two arguments: iterations, for the number of loops, and delay (in milliseconds) for the delay in each loop. The specific function is available at <https://hub.docker.com/r/gkousiou/noderedaction> and can be registered in OW with the command:

```
wsk action create dockeraction --docker gkousiou/noderedaction
```

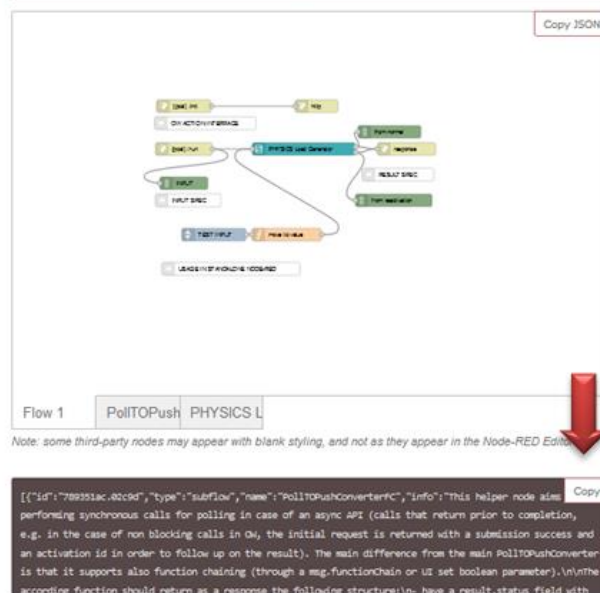


Figure 27 - Retrieval of Pattern Subflow from PHYSICS Node-RED repo collection

Then one can navigate in the Node-RED menu Import option (Figure 28), select it and paste the contents of the updated flow.

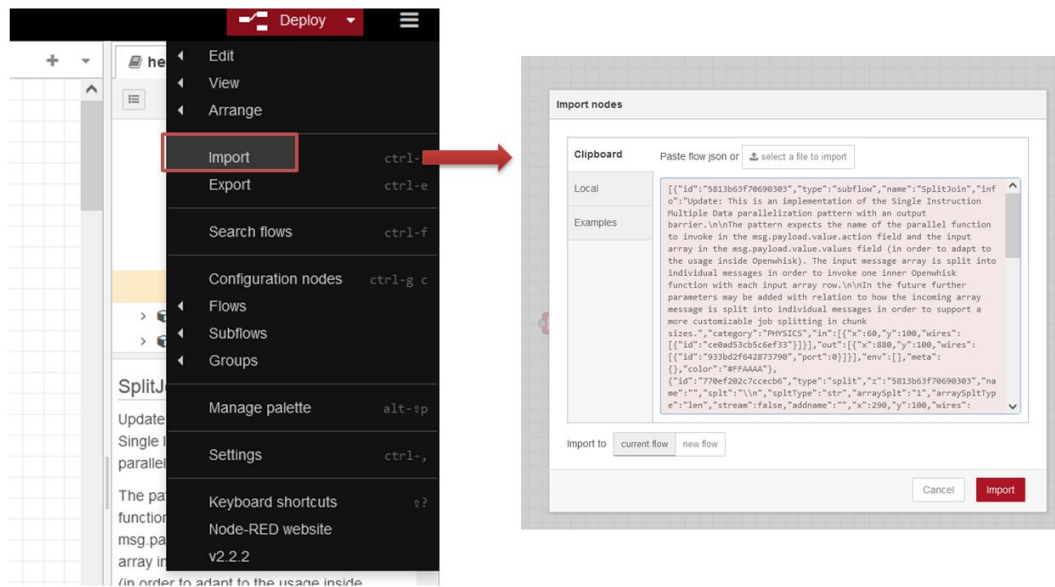


Figure 28 - Import of Updated Pattern Subflow in the DE Node-RED Editor

If a previous version of the flows exists in the environment (it should), the user will get a warning message for importing existing nodes (Figure 29).

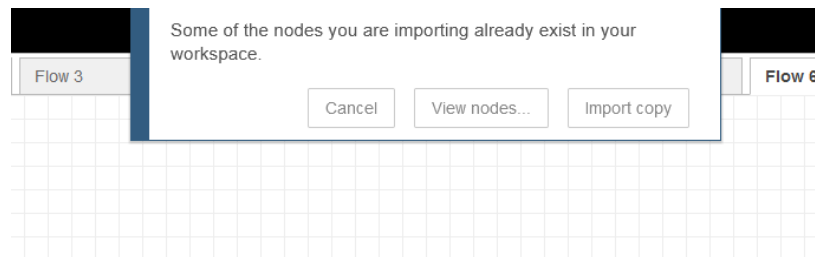


Figure 29 - Warning Message for Duplicate Nodes

By selecting the “View nodes” option, the user will get a relevant screen about conflicting nodes (Figure 30). If they do not select the grey subflow in the Subflows section of import, only the node reference will get imported, so the imported flow will use the locally existing (previous) version of the subflow.

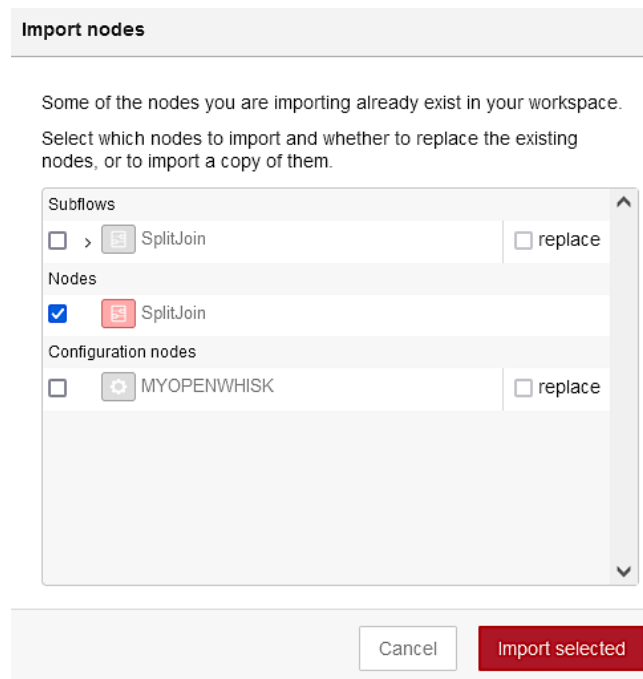


Figure 30 - Selection of Node and Subflow Replacement Option

If we select the SUBFLOW and REPLACE check box, the subflow is replaced across all flows and the palette has only one instance of the subflow. This is a **complete update** of the subflow across the entire environment (this and all other existing flows that may use the subflow). If we don't check "replace", a new version is included in the palette (but without differentiation in the name) and the subflows are not replaced in the previously existing flows. For any new flow we can select whatever of the two versions from the palette, although there is no differentiation in the name appearing in the palette. This is in essence versioning of the subflow.

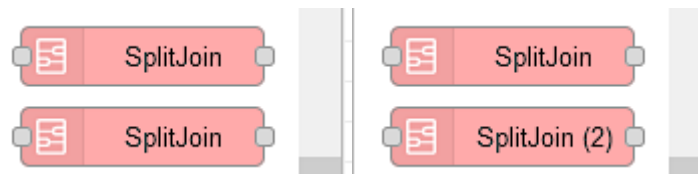


Figure 31 - Different Appearance of Versions a) without name differentiation b) with name differentiation

Given that this creates confusion afterwards, if we want the versioning option, we should use the direct Import Copy option from Figure 29. This would result in maintaining the current version of the subflow in existing flows and the new version in the imported flow, while the new version will be clearly marked in the palette. This is needed in cases where a PHYSICS developer may have performed additions or alterations in the initially provided subflow.

**3.3.6 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 2 main OpenWhisk (OW) actions that are foreseen for read/write operations, as indicated in the respective

section. In order to make that more abstract, a relevant DMS interface node has been created in the Node-RED environment. Its description and usage has been included in D3.2.

### 3.4 Elasticity Controllers

#### 3.4.1 Overview

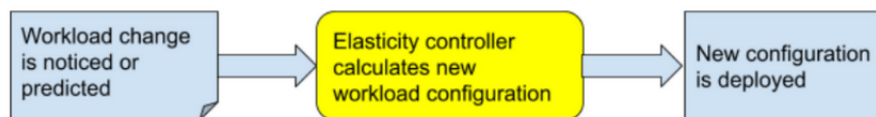
The Elasticity controllers target is to scale resources associated to the functions running on pods on the Kubernetes clusters. We increase/decrease the associated resources for functions by either changing the memory/CPU associated to the pod hosting the function (VPA), or by creating more pods for running the functions (HPA), or even by creating more Kubernetes workers to host more functions/pods.

In the first part of the project, we designed the elasticity controllers to be integrated into the infrastructure layer through the already supported Kubernetes Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA) and then enhancing them with the usage of Custom Metrics as CPU and memory was not going to be sufficient. In the second phase of the project, and after evaluating the evolution of upstream projects, we decided to shift the focus and adopt KEDA<sup>17</sup> (Kubernetes Event-driven Autoscaling). KEDA is better suited for PHYSICS use cases that are based on Function as a Services, which is also (usually) event driven – e.g., output of a function will trigger the execution of the next one.

#### 3.4.2 Technology architecture

Our idea for the elasticity controller flow, as depicted in Figure 32, is that:

1. some events are noticed/predicted
2. the selected elasticity controller (associated to the application) gets executed and calculates the new configuration
3. the new configuration is applied



*Figure 32 - Elasticity Controllers Flow*

KEDA matches this very well. It follows the same pattern we followed in PHYSICS and it is implemented by extending the Kubernetes API with a set of Custom Resource Definitions (CRDs). It works alongside the Kubernetes HPA but allows for scaling to 0, as well as using custom metrics. In addition it provides a catalog of autoscalers that can be tuned or adjusted for different use cases. This is what we have done in PHYSICS, for more information see Deliverable D3.2.

By using and tuning the different scalers at the catalog we are able to extend the existing Kubernetes scalability controllers (HPA) to support different metrics that cover extra requirements such as supporting green computing (using less energy) or saving deployment cost (using cheaper instances). It provides a broader view of the cluster as well as from the applications.

In our case we have focused on:

- Scaling the Kubernetes (OKD) cluster nodes depending on the number of functions. For this we have adjusted the KEDA kubernetes-workload scaler, which allows us to set a target number of functions per Kubernetes nodes.

---

<sup>17</sup> <https://keda.sh/>

- Scaling the number of OpenWhisk invokers depending on the OpenWhisk Kafka queues status. For this we have adjusted the KEDA kafka scaler, feeding in the information from the OpenWhisk Kafka groups so that more invokers can be created so that events from the queue can be processed faster.

### 3.4.3 Interfaces/API

As KEDA is extending the Kubernetes API with CRDs, the interface/API is the Kubernetes API. The mechanisms are the same as for any other Kubernetes object. Thus it supports the standard get/list/create/update/delete methods.

KEDA provides a standard API (KEDA ScaledObject) to deploy and associate the scalers to the relevant workload(s), with different configurations depending on the scaler. As regards to the CRD itself, the ScaledObject has 3 main blocks:

- *scaleTargetRef*: specifies the object that needs to be scaled as a result of the scaler decision. Basically, it needs to state the name of the resource to scale and its kind. In addition, it supports scaling custom objects (as long as they implement the Kubernetes scale subresource). We took advantage of this to scale the cluster nodes when managed through the Kubernetes Cluster API, i.e., when the nodes are represented with machinesets (as in the OKD case).
- *replicaCount*: allows setting limits for the scale actions, i.e., a min and a max number of replicas that can be created by the scaler.
- *triggers*: selects the type of trigger to be used, i.e., the scaler to use (e.g., Kubernetes-workload or Kafka). It also has associated metadata per scaler where each scaler defines its own, for instance:
  - Kubernetes-workload: <https://keda.sh/docs/2.11/scalers/kubernetes-workload/#trigger-specification>
  - Kafka: <https://keda.sh/docs/2.11/scalers/apache-kafka/#trigger-specification>

### 3.4.4 Distribution, deployment and configuration

KEDA is an open source upstream project distributed under Apache License. It supports different deployment options: Helm, operator and yamls.

As for the configuration options:

- It allows associating scalers to workloads through ScaleObject CR objects.
- It has a common section to configure the object to scale as well as the scaling limits.
- And each scaler defines its own configurable option (triggers section). For example the kubernetes-workload option allows you to define a podSelector and a value. This is used to decide how much should be scaled the targeted object, depending on its relation (value) with the workload pointed by the podSelector.

## 3.5 Gaming Platform

### 3.5.1 Overview

Gamification is the process of applying game mechanics and artifacts into non-game contexts to improve the learning experience and practical knowledge gained. These game mechanics can transform a potentially tiresome learning experience into one that is fun and engaging for the user, who benefits from instant feedback since it enables them to promptly correct errors and comprehend topics. In addition, gamification can develop a healthy feeling of competition and give users the flexibility to learn at their own pace. Gamification can be a useful method of encouraging and motivating learners, particularly when traditional software training methods are seen as difficult to follow or tedious. In the context of PHYSICS, the development of game mechanics can potentially streamline the onboarding and adaptation of the project's platform and artifacts. During the project's



2<sup>nd</sup> hackathon, a gamification approach was proposed, and implemented as a gaming server, with the goal of allowing users to interact with the Node-RED flow programming environment, where they could easily onboard and interact with the PHYSICS developed patterns in a gamified environment. This game environment can also assist in the dissemination of the PHYSICS platform while providing a way to train and educate the users on how to use its findings. To realize this approach, a gaming server employing the appropriate mechanisms to foster a gamified learning environment for training was constructed. The gaming server features a game, offered as a web page application with choices for local mode use, targeting flow-programming training tutorials and development of storylines, the online mode, focused on online competitions and storyline sharing, and the flow marketplace, allowing for flow/subflow sharing and trading as NFTs using a Smart Contract.

### 3.5.2 Technology architecture

The gamification platform incorporates multiple components for the available modes, these include a local server that hosts the game and interconnects a Node-RED visual environment, an online server, that allows progress tracking, storyline sharing and interacting with other players within the game, and the flow marketplace, which enables sharing and trading developed patterns, subflows and flows. The initial gaming server architecture can be found on D3.2 in section 5.2., while the gaming platform architecture including the marketplace is shown in (Figure 33).

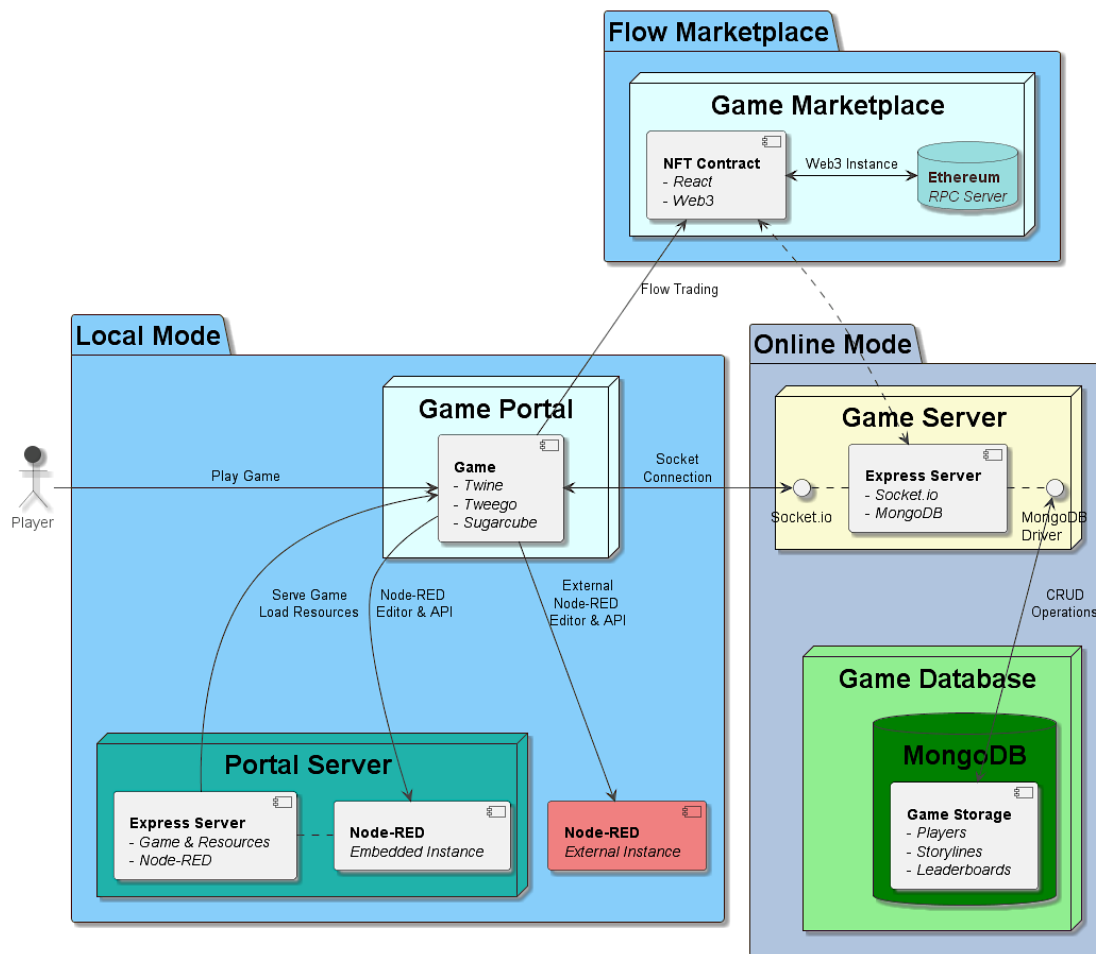


Figure 33 - Gaming Platform Architecture

- **Local Mode:** The required components necessary to play the game locally.

- **Portal Server:** The Portal Server, implemented in Typescript, consists of a NodeJS Express server with an embedded Node-RED instance and hosts the Game Portal. The server provides the necessary assets and resources needed by the Game Portal to load the Game.
  - **Game Portal:** The Game is served as an HTML web application developed using Tweego, a command line compiler for creating Twine/Twee story formats, and Sugarcube, a JavaScript library built for Twine that handles story logic, assets & media, playback functions, passage linking and UI elements.
  - **External Node-RED:** The Game can be customized to utilize an external instance of Node-RED provided with the necessary credentials or authentication token.
- **Online Mode:** The required components necessary to play the game online.
- **Game Server:** The Game Server implemented in Typescript, consists of another NodeJS Express server that utilizes the Socket.io module to create and handle real-time communication channels with every Game Portal connected to it. This component allows the Game Portals to connect to the Game Database, utilizing a MongoDB driver, to store each player's progress using CRUD operations and serves additional online storylines and resources.
  - **Game Database:** The Game Database, utilizing a Mongo Database, stores the document records regarding registered Players, their progress, any deployed storylines, ongoing competitions, and leaderboards.
- **Flow Marketplace:** The necessary components to access and interact with the game's marketplace.

### 3.5.3 Interfaces/API

The gamification platform implements the following API interfaces:

#### **Local Mode APIs:**

*Table 45 - Local Mode APIs*

Method	Path/URI	Description	Request/ Parameters	Response
GET	/	Main API for accessing the Game	-	HTML Page
GET	/play	API for accessing the embedded Node-RED Editor. (Can also be configured from the environmental variables)		HTML Page
GET	/red	API for accessing the embedded Node-RED HTTP APIs. (Can also be configured from the environmental variables)		JSON/Text

#### **Online Mode APIs:**

*Table 46 - Online Mode APIs*

Method	Path/URI	Description	Request/ Parameters	Response
GET	/	Main API for connecting to the Game Server.	-	HTML Page
GET	/storylines	API for getting available online storylines		JSON
GET	/player/:id	API for getting a player's profile		JSON/Text
POST	/player/:id	API for creating a player's profile		JSON/Text



<b>PUT</b>	/player/:id	API for updating a player's profile		JSON/Text
<b>DELETE</b>	/player/:id	API for deleting a player's profile		JSON/Text

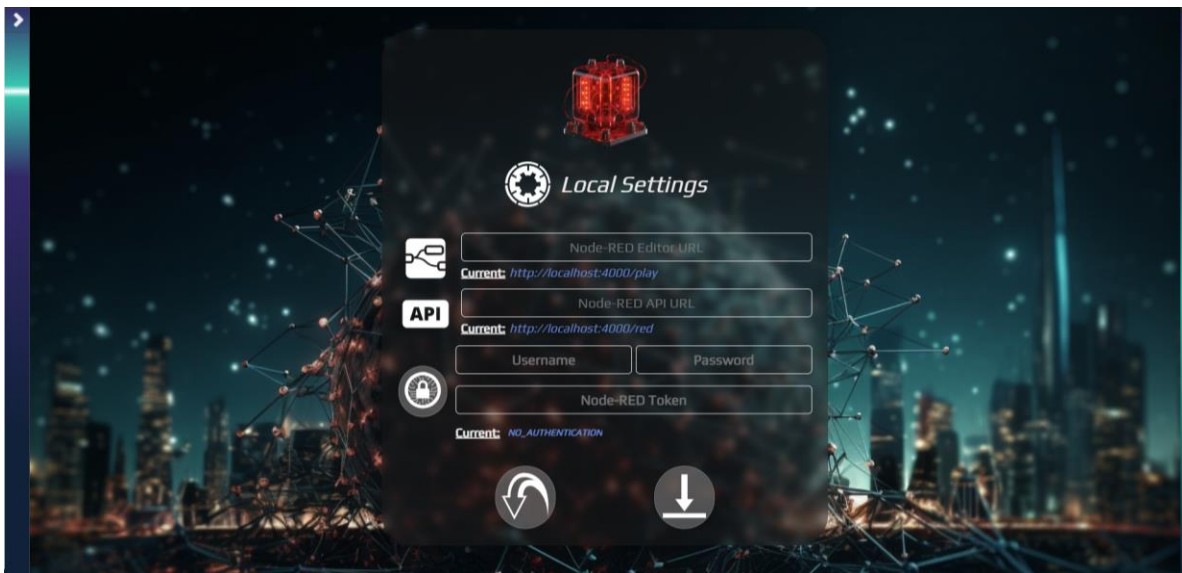
### **Flow Marketplace APIs:**

*Table 47 - Marketplace APIs*

Method	Path/URI	Description	Request/ Parameters	Response
<b>GET</b>	/	Main API for connecting to the Game Marketplace.	-	HTML Page

### 3.5.4 Distribution, deployment and configuration

Each part of the gamification platform is dockerized and can be accessed in the docker-hub. There is also a docker-compose file allowing for automated builds and deployment of the infrastructure. Each component can be configured in the adjacent. *env* file they have. The Game Portal can also be configured from within the game allowing for the connection of external Node-RED instances to be used in the Game, this is shown in (Figure 34)



*Figure 34 - Settings Screen*

### 3.5.5 Game Portal Tutorials

The Game Portal features “*Flowchain Champions*”, a Game built using Twine and Sugarcube. Once the Game starts, a loading screen appears, which preloads all assets from the Portal Server. The starting screen once the assets load, is shown below in (Figure 35).

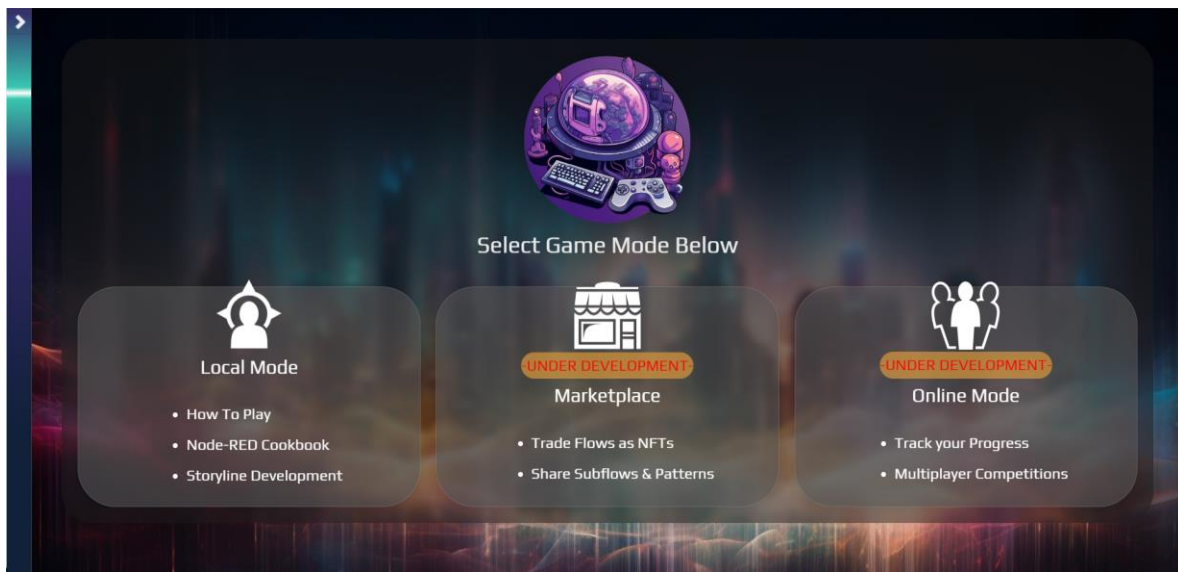


Figure 35 - Game Starting Screen

### ➤ **Local Mode**

The local mode of the platform allows users to learn using through recipe tutorials and create their own Storylines. The definition for the Game Storylines can be found on D3.2 in section 5.3. Once choosing this mode the user is navigated to the Local Menu shown in (Figure 36)

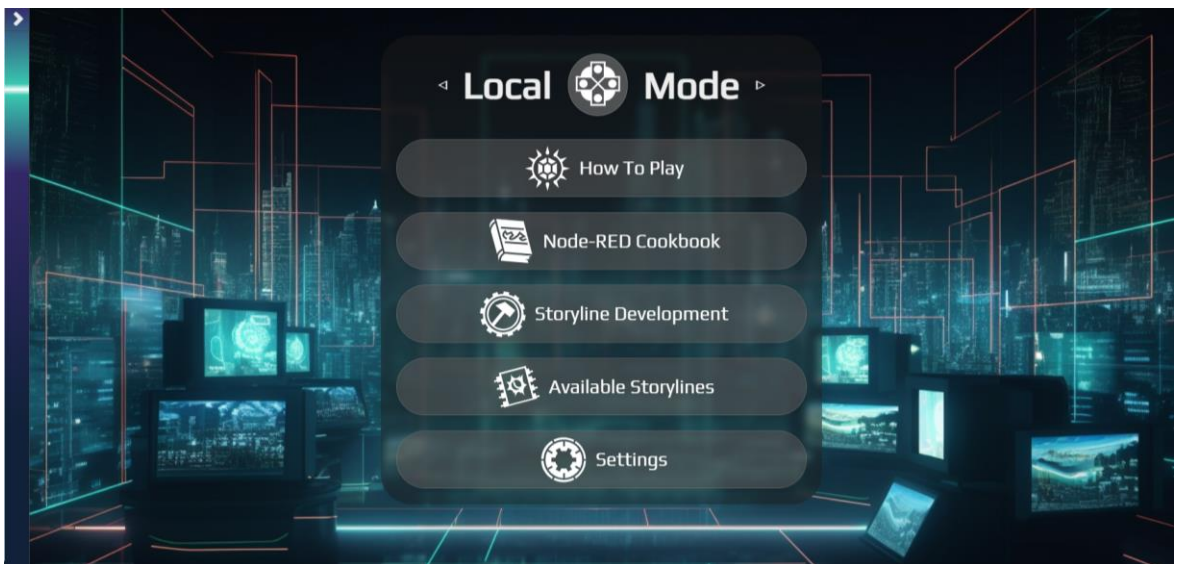


Figure 36 - Local Menu Screen

### ➤ **Node-RED Tutorials**

Users can access Node-RED tutorials through the “Node-RED Cookbook” option that allows them to pick from several training topics regarding Node-RED flow programming. These can be seen below in (Figure 37), (Figure 38) and (Figure 39) below.



Figure 37 - Node-RED Cookbook Screen

Each tutorial topic has several tutorials depending on the kind of action the user wants to do. Each tutorial provides the corresponding flow that the user can import and interact by clicking on its picture.

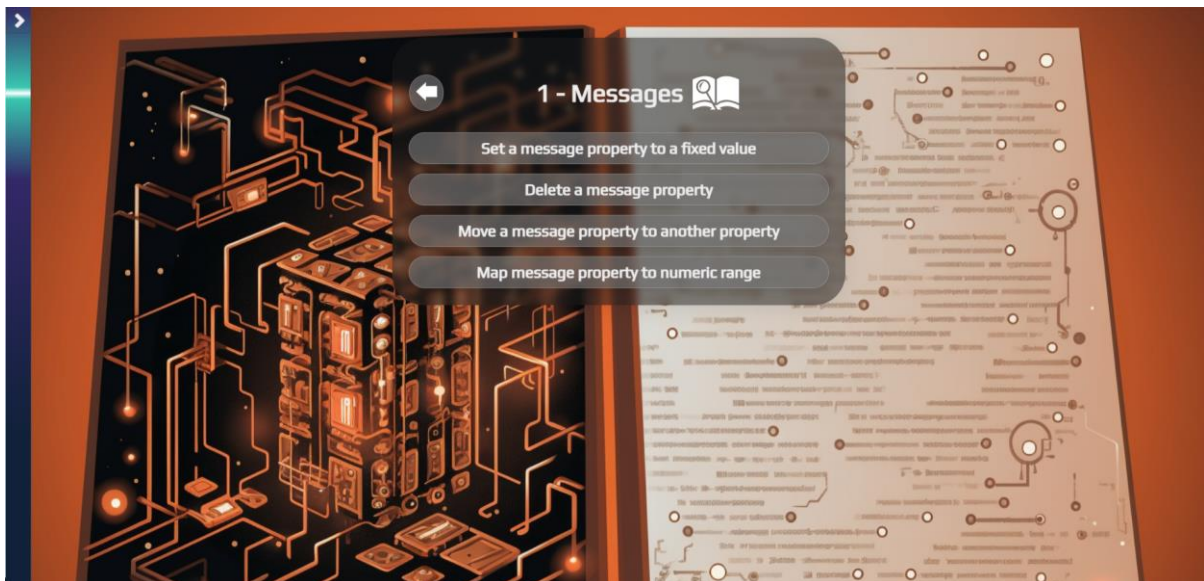


Figure 38 - Messages Tutorial Screen

The tutorial provides a problem situation and its respective solution in Node-RED. The button on the right (*joystick*) can be clicked to open the embedded Node-RED Editor, shown in (Figure 39)





Figure 39 - Messages #1 Tutorial Screen

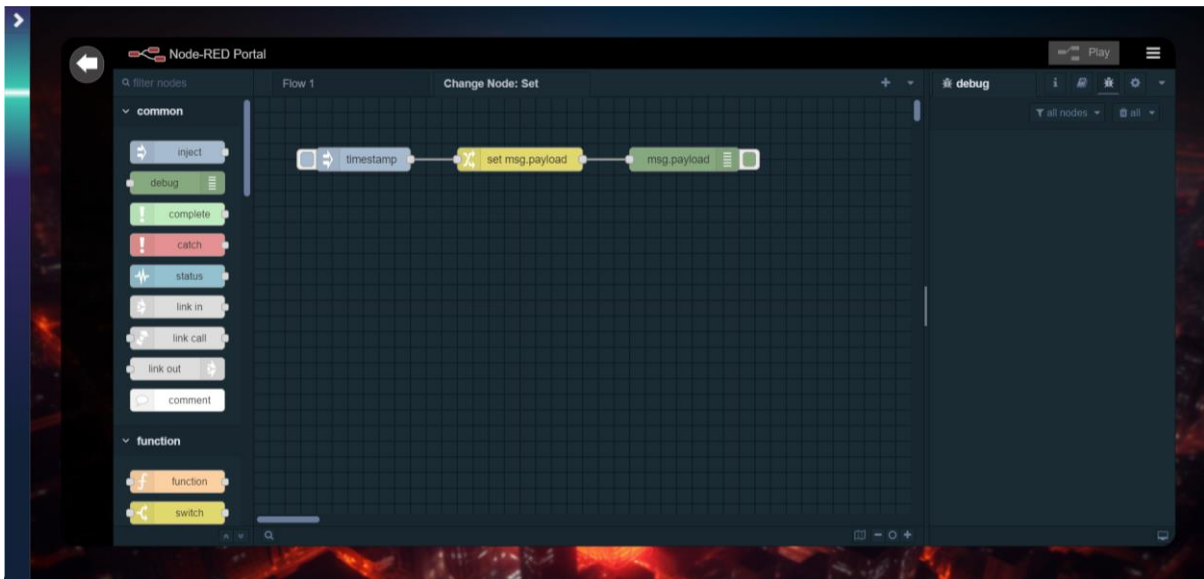


Figure 40 - Node-RED Portal Screen

The user can access the Node-RED editor directly from the Game and interact with the nodes of the tutorial (Figure 40). The Game Portal takes care of installing any modules or importing any subflows the flow might have, so the user can be onboarded faster.

## 3.6 Reasoning Framework

### 3.6.1 Overview

The Reasoning Framework component manages application and resource metadata within the context of graphs for the PHYSICS platform. It oversees the storage, retrieval, and semantic reasoning over this metadata to create global graphs, facilitating optimal application deployment strategies by harmoniously interlinking suitable compute nodes from the available resource graphs with respective nodes from the application graphs.

Key features and improvements in its final iteration include enhanced application-placement strategies thanks to the integration of performance data divided into three vital categories: benchmark data acquired during cluster onboarding, function-specific benchmarks from Performance Evaluation component and dynamically forecasted runtime metrics to allow real-time updates to the global graph. For detailed information refer to Section 2 of D4.2. The overall interactions with other platform components are illustrated in Figure 41.

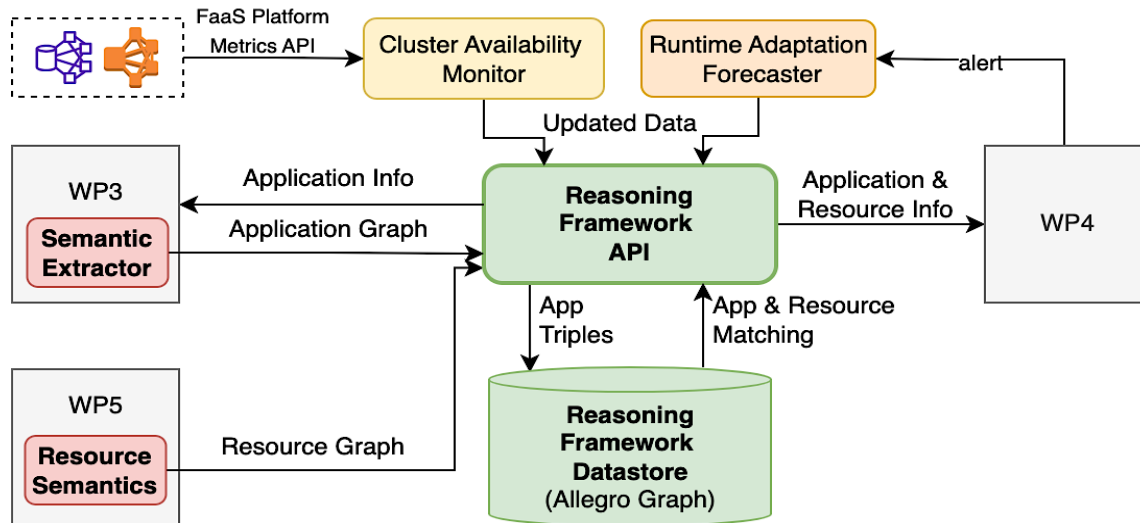


Figure 41 - Reasoning Framework interactions with other components

### 3.6.2 Technology architecture

Reasoning Framework consists of two integrated components the one serving as an API and the second as datastore of the semantics graphs. Both are REST microservices deployed as Docker containers. The API, compliant with the OpenAPI<sup>18</sup> standard, is based on the Flask<sup>19</sup> framework. The datastore 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<sup>20</sup> was opted for the data storage needs of the Semantics Block. AllegroGraph provides an architecture through the REST protocol and is characterized by the efficient 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 maximize value from semantic technologies [2]. The internal components of the Reasoning Framework and their main interactions are shown in Figure 42.

<sup>18</sup> <https://spec.openapis.org/oas/latest.html>

<sup>19</sup> <https://flask.palletsprojects.com/en/2.0.x/#api-reference>

<sup>20</sup> <https://allegrograph.com/>

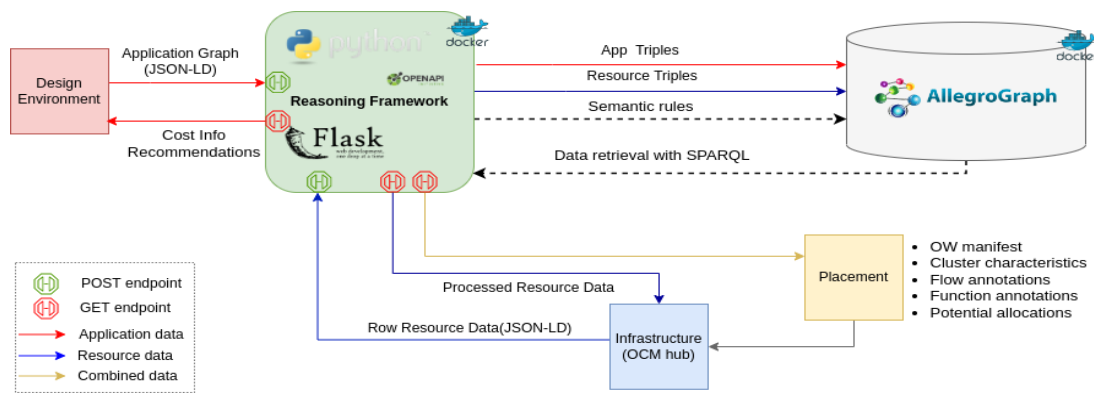


Figure 42 - Reasoning's Framework internal components

### 3.6.3 Interfaces/API

This section lists all the available endpoints of the Reasoning Framework offered by its API. It should be noted that these endpoints are reached programmatically facilitating back-end platform operations.

Table 17 – Reasoning Framework endpoints for Resources

Method	Path/URI	Description	Request/Parameters	Response
POST	/cluster	Register a new cluster	JSON-LD	JSON
GET	/cluster	List all clusters and their spec	-	JSON
GET	/cluster/{id}	Get cluster and its complete specs	string:id	JSON
DELETE	/cluster/{id}	Delete a cluster	string:id	JSON
PUT	/cluster/{id}	Update a cluster	string:id	JSON
POST	/cluster/register	Register cluster from OCM hub	JSON-LD	JSON
POST	/cluster/availability	Update Cluster Availability Score	JSON	JSON
POST	/cluster/performance	Update Cluster Performance Score	JSON	JSON

Table 18 - Reasoning Framework endpoints for Applications and Semantic Matching

Method	Path/URI	Description	Expects/Parameters	Response
POST	/application	Store a new application graph	JSON-LD	JSON
GET	/application	Get id and names of stored applications	-	JSON
GET	/application/{id}	Get application graph	string:id	JSON
DELETE	/application/{id}	Delete an application	string:id	JSON
GET	/application/run/{id}	Get the required information for the deployment of the given application (used by Optimizer)	string:id	JSON
PUT	/application	Update a stored application	JSON-LD	JSON

### 3.6.4 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 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 (see section 5.1). The deployment of the Semantics Block follows the same process as the other platform components in the PHYSICS AWS testbed in a dedicated namespace (i.e., WP4). The continuous delivery (CD) part is also automated through the Jenkins pipeline.

The specific instructions required to parametrize, build and deploy the Semantics Block (i.e., Docker-Compose, Jenkins, and K8s YAML config files) are also available in its Gogs repository. A screenshot of the logs from the Kubernetes Pod serving Reasoning Framework (former Semantics Block) is illustrated in Figure 43.

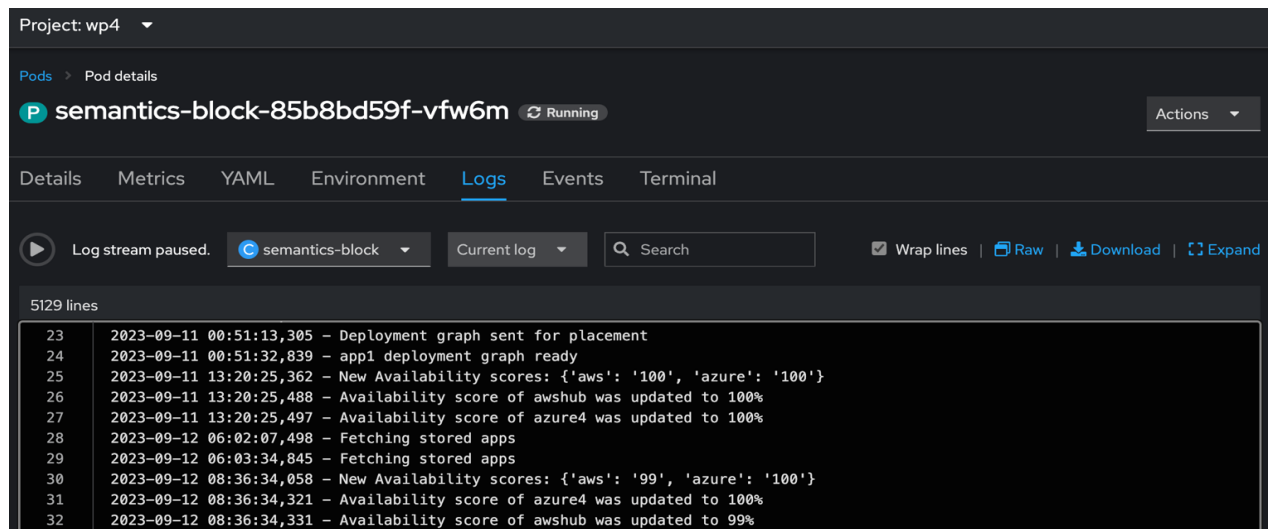


Figure 43 - Screenshot of Reasoning framework logs

## 3.7 Runtime Adaptation

### 3.7.1 Overview

Towards enabling runtime adaptation in the context of PHYSICS, two additional components were developed during the second period of the project: A Monitoring-alert and a Forecasting component. These components, integrated with the Adaptive Platform (see 3.13) and Reasoning Framework components, work together to ensure the robustness and quality of the platform by responding to changes in performance and predicting latency issues. This runtime adaptation process is orchestrated to maintain the platform's performance and quality of service. The collaboration of these components ensures that any detected issues are swiftly addressed, allowing for a resilient and adaptable system.

### 3.7.2 Technology architecture

The technology architecture for the Runtime Adaptation component is designed to facilitate dynamic adjustments to the execution environment based on real-time performance metrics. This adaptation process is an integral part of the platform's response to changing workload characteristics, ensuring optimal performance and reliability.

Runtime Adaptation involves two main REST services: the Monitoring-Alerts and the Forecaster. These components work in tandem to monitor and predict system performance. The Monitoring-Alert component continuously assesses the quality of service (QoS) and detects deviations, while the Forecaster component predicts future performance trends. Both components communicate seamlessly via the RabbitMQ message broker, allowing for efficient data exchange and coordination.

To ensure portability and scalability, both the Monitoring-Alerts and the Forecaster components are encapsulated within Docker containers. These containers are deployed within the WP4 namespace of the project's AWS testbed. A visual representation of this overall architecture and process for runtime adaptation can be found in the accompanying figure.

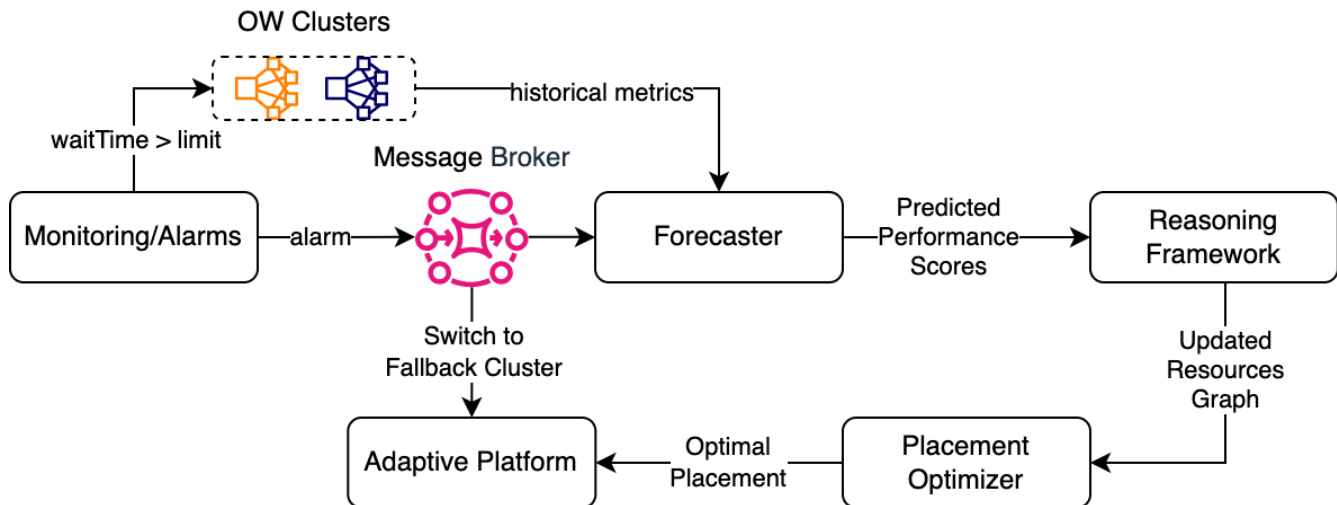


Figure 44 - Runtime Adaptation Architecture

Here's a breakdown of how they operate:

1. **Monitoring-Alerts (REST service):** This component is responsible for continuously monitoring the performance of the deployed applications. It relies on monitoring tools, like Prometheus, where applications and infrastructures metrics values are collected and stored, and it operates under the principle of quality of service (QoS) and creates alarms when QoS violations occur. A violation, in this context, is defined as a situation where the function wait time (i.e., the corresponding metric value stored in Prometheus) exceeds a specified limit. When such a violation is detected, the Monitoring-Alert publishes an alert message to a dedicated channel within RabbitMQ (a message broker).
2. **Adaptive Platform:** Upon receiving an alert from the Monitoring-Alert, the Adaptive Platform takes immediate action. The Translator of the Adaptive Platform routes incoming requests to a fallback OpenWhisk cluster, ensuring that users experience minimal disruption even when performance issues are detected.
3. **Forecaster (REST service):** Upon receiving an alert from the Monitoring-Alert, it performs the following tasks:
  - a. Retrieves the most up-to-date performance metrics from the OpenWhisk API of each registered cluster.
  - b. Utilizes an exponential smoothing model (see D4.2 for more details) to predict the future average latency for each cluster.
  - c. Transforms the predicted latencies into performance scores for each cluster, providing a quantitative measure of their performance.
  - d. Updates these cluster performance scores within the Reasoning Framework, ensuring that the adaptation process is informed by the latest data.
  - e. Triggers the Reasoning Framework to re-initiate the deployment of the application when QoS violations are detected, ensuring that corrective measures are taken promptly.



### 3.7.3 Interfaces/API

This section presents the endpoints of all the different components involved in the Runtime Adaptation as described in Table 48.

*Table 48 - Interfaces and Endpoints for Runtime Adaptation*

Component	Method	Path/URI	Description	Expects/ Parameters	Response
Monitoring-Alerts	<b>GET</b>	<i>/api/v1/qos</i>	Returns the list of all QoS definitions (i.e., function's wait time metric constraint)		JSON containing the list of all QoS definitions
Monitoring-Alerts	<b>POST</b>	<i>/api/v1/qos</i>	Creates a new QoS definition and starts the evaluation process	JSON	JSON
Monitoring-Alerts	<b>GET</b>	<i>/api/v1/qos/{id}</i>	Returns the information about the QoS definition identified by <i>{id}</i>	QoS identifier in path	JSON
Monitoring-Alerts	<b>DELETE</b>	<i>/api/v1/qos/{id}</i>	Deletes the QoS definition identified by <i>{id}</i> , and finish the evaluation process	QoS identifier in path	JSON
RabbitMQ				JSON message with the information about the QoS violation and the corresponding function	
Forecaster	<b>GET</b>	<i>/predict/app_id?action_name</i>	Triggers the forecaster to predict the future performance score per cluster	JSON {app_id:"<id>", "action_name": "<name>"}	JSON {"clusterA": <score>, "cluster": <score>}
Reasoning Framework	<b>POST</b>	<i>/cluster/performance</i>	Update Cluster Performance Score	JSON	JSON
Reasoning Framework	<b>GET</b>	<i>/application/run/{id}</i>	Get the required information for the deployment of the given application (used by Optimizer)	string:id	JSON

### 3.7.4 Distribution, deployment and configuration

The Monitoring-Alerts component is a Golang application that can be containerized as a Docker image. This way it can be deployed in a Kubernetes type platform like the project test environment. It requires the following environment variables to be defined:

- **“monitoring\_adapter”**: this variable specifies the monitoring tool used by the Monitoring-Alerts component to get the metrics values needed to do the assessment of the QoS definitions / constraints.
- **“prometheusUrl”**: in the case the monitoring adapter is “prometheus”, this variable defines the URL to connect to it.
- **“notifier\_adapter”**: this variable defines the notifier used to send alerts and notifications (i.e., RabbitMQ).
- **“rabbitMQ”**: in the case the notifier adapter is “rabbitmq”, this variable defines the connection string.
- **“rabbitMQExchange”**: this variable defines the exchange topic used to send the alerts and notifications.

Additionally, the component starts the REST API listening in port 8080. Initially, there are no QoS definitions created by default. Thus, to create one QoS definition and start the process of QoS evaluation, users must connect to the REST API (*POST /api/v1/qos*) to create a new one. The following JSON shows a QoS definition example:

```
{
  "id": "<QoS_ID>",
  "name": "<QoS_Name>",
  "state": "started",
  "details": {
    "name": "<QoS_Name>",
    "guarantees": [{
      "name": "<constraint_name>",
      "constraint": "[<metric1>/<metric2>] < <Threshold_value>"
    }]
  }
}
```

The Docker and YAML files used to deploy this application in Kubernetes, including the project test environment can be found in the git repository, under folder “resources/deployment”:

<https://gogs.apps.ocphub.physics-faas.eu/WP4/monitoring-alerts-app.git>

## 3.8 Cluster Availability Monitor

### 3.8.1 Overview

The Cluster Availability Monitor is a Python-based service developed to monitor the optimal operation of OpenWhisk clusters. Leveraging real-time monitoring, it conducts systematic checks on the availability of specified clusters at regular intervals and computes their availability scores. These scores are computed based on successful and failed connections over a period of one week, facilitating a robust understanding of each cluster's average availability over time. Upon changes on cluster's availability, the service updates the Reasoning Framework (T4.1) enabling WP4 to decide on the placement of the income requests accordingly.

### 3.8.2 Technology architecture

The Cluster Availability Monitor operates on a Python-based image, ensuring lightweight and efficient performance. It utilizes the requests library for API communications and the Python built-in threading library to handle parallel execution of availability testing and score calculation.

The core of the architecture is built upon two main threads that operate concurrently:

1. **Cluster Availability Testing Thread:** Periodically tests the availability of each cluster and stores the results in a data structure implemented using Python's deque class, which holds a week's worth of data.
2. **Availability Score Calculation Thread:** Analyzes the stored data every minute to calculate and update the availability score based on the most recent week of data.

In terms of data storage, it leverages Python's in-memory data structures, keeping the operation fast and efficient without requiring a separate database system.

### 3.8.3 Interfaces/API

The tool interfaces with the OpenWhisk clusters through REST APIs, invoking actions and registering new actions as needed. It also communicates with the Semantics Block's Reasoning Framework to update the availability scores of the registered clusters as shown in Table 49.

*Table 49 - Cluster Availability Monitor Interfaces*

Endpoint	Method	Input Parameters	Output Parameters	Description
/cluster/update_availability	POST	Cluster name, availability score	200 OK, error codes	Updates the availability score of a specified cluster in the reasoning framework.
ClusterAvailabilityMonitor (OpenWhisk action)	POST	Cluster name, parameters for action	Action output, error codes	Invokes the ClusterAvailabilityMonitor action in OpenWhisk with the specified parameters.

### 3.8.4 Distribution, deployment and configuration

The Cluster Availability Monitor is packaged into a Docker container, making it portable and easy to deploy across various environments, including the PHYSICS AWS infrastructure. The source code of the service is available in the project's git repository (Gogs), while the latest container image is stored in the image repository of PHYSICS in Harbor. Jenkins CI/CD pipelines are set with the service deployed on the WP4 namespace. A screenshot of the logs from the Kubernetes Pod serving Cluster Availability Monitor is illustrated in Figure 45.

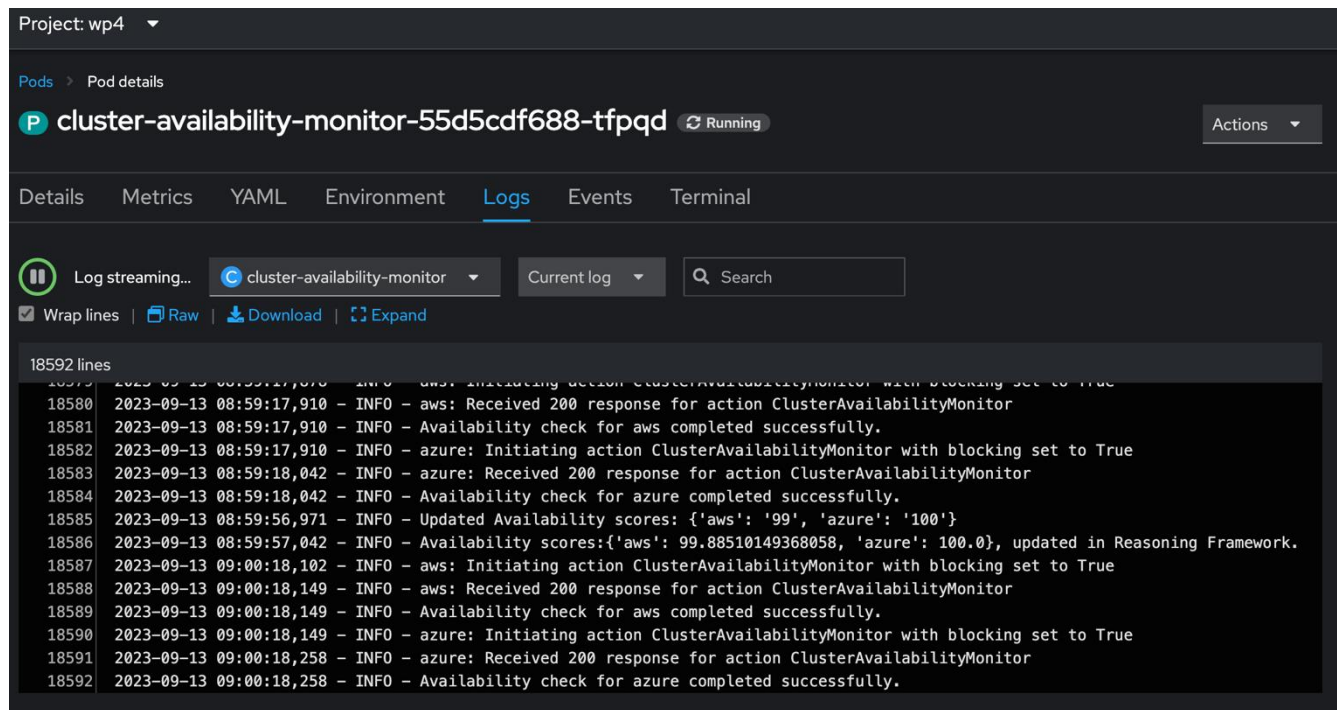


Figure 45 - Screenshot of Cluster Availability Monitor logs

### Deployment:

- Build the Docker image from the Dockerfile.
- Run the Docker container, with the CLUSTERS\_CONFIG environment variable to specify the clusters to monitor.
- The tool requires the configuration of environment variables to function correctly. The critical variable being CLUSTERS\_CONFIG, holding the cluster credentials in a JSON format.
- The Python script *app.py* serves as the entry point to the application, initializing the necessary data structures and starting the monitoring threads.

Instructions on how to set up, configure, and deploy the Cluster Availability Monitor can be found in the accompanying README.md file in the project's Gogs repository. Furthermore, an open-source version of the Cluster Availability Monitor is available on the [RAMP](#).

## 3.9 Resource Semantics

### 3.9.1 Overview

The second iteration of the component includes the necessary changes to allow the Resource Semantics component to be deployed in each of the managed clusters by the PHYSICS platform and communicate with the Reasoning Framework. Previously, this component was part of the “Semantics Block”, integrated through the deployment of Reasoning Framework awaiting from external requests to retrieve a cluster’s information. With the new approach, the capabilities expand and now the resource semantics incorporates methods to automatically extract a cluster’s functional information, transform it according to the designed resource ontology and finally propagate it to the central cluster to be reasoned upon and enable application to cluster matching.

More specifically each time a new cluster is added to the platform, the cluster onboarding process takes place which includes the component's installation. Additionally, the cluster onboarding also deploys a benchmark application which precedes the Resource Semantics so the latter can afterwards retrieve benchmark related information.

This new approach is feasible due to the lightweight nature of the component and provides added benefits such as automated by the component processes, easier debugging, and access to cluster-wise semantic modifications.

### 3.9.2 Technology architecture

The component has been built as a python-based REST API service through Flask to accommodate HTTP communication between components in the platform and to serve to the cloud engineer valuable debugging information outside the components main pipeline. OWLready is the library used to manipulate and populate the ontology which is ingested as an OWL file built through Protégé. Additionally, the component communicates with each cluster's Kubernetes and Prometheus API. This is achieved through the Kubernetes-api-client and Prometheus-api libraries. Additionally, the component also includes a couple of html files served through flask to provide a graphical interface for the user.

### 3.9.3 Interfaces/API

The component API has been automatically documented through Swagger, a popular python library for documentation. Decorators and methods have been implemented to export the expected models and output for each endpoint, when necessary. The whole documentation can be found on the service's URL /documentation endpoint.

Essentially the endpoints have been divided into the main and debug endpoints. The first group contains the essentials for the component's execution and the latter contains endpoints that aid the cloud engineer to debug the component. The exact endpoints can be found in Table 20 and Table 21.

Table 20 – Resource Semantics debugging endpoints.

Method	Path/URI	Description	Request/ Parameters	Response
GET	/debug/logs	Retrieve the execution logs of the component.	-	Web Interface
GET	/debug/nodes	Retrieve the response of the Kubernetes API command “describe nodes” to test component to API connection.	-	Web Interface
GET	/debug/post-cluster-semantics	Posts the cluster semantics to the Reasoning framework to test communication.		200

Table 21 – Resource Semantics main endpoints.

Method	Path/URI	Description	Request/ Parameters	Response
GET	/main/home page	A page with generic information on the component and a couple of redirection buttons.	-	Web Interface
POST	/main/kube-semantics-trigger	This is the main endpoint to trigger the component's pipeline. It is triggered by the cluster onboarding process and sends the semantics to the Reasoning Framework	string: benchmark_pod_label	200

GET	/main/cluster-semantics	Retrieves directly the cluster semantics in an json-ld format.	-	Web Interface
GET	/main/ontology-raw	Displays the ontology designed to describe cluster resources.	-	Web Interface
GET, POST	/main/annotate-cluster	An endpoint for manual annotation of the ontology with specific individuals. To be used in the future by domain experts to easily import new aspects.	Json: {"subject": string, "predicate": string, "object": string}	Web Interface, 200

### 3.9.4 Distribution, deployment and configuration

The component is deployed as a dockerized service. It follows the necessary structure to ensure compatibility with swagger for automated documentation production. As such endpoints, according to their category (debug, main) are under the respective folder and then are declared in the main app.py file. Additionally, the docker also includes the templates folder with all the necessary HTML files and the OWL file which depicts the resource ontology.

In order to be able to access the Kubernetes API the component also comes with a set of configurations that bind a cluster-role to the service which enables the listing of different Kubernetes resources. The ClusterRole syntax of the configuration file follows.

#### Code Snippet – Resource Semantics ClusterRole YAML configuration file.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: list-resources
rules:
- apiGroups: [""]
  resources: ["pods", "nodes", "services", "namespaces", "configmaps", "persistentvolumes"]
  verbs: ["list"]
```

## 3.10 Performance Evaluation Framework

### 3.10.1 Overview

The Performance Evaluation Framework (PEF) aims at enabling REST based launching of relevant performance driven tests against a target function. To do so, it incorporates adapted function clients that are parametric and can be used 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, or through the inclusion in the semantic model of the function, supporting also multiple locations of that function. In the latter case, the results are acquired through the relevant interfaces of the Reasoning Framework

In the second part of the project, the relevant PEF APIs and provided functions have been extended in order to enable the realization of the Performance Pipeline, described in D3.2. Furthermore, new queries are enabled, to collaborate with the Semantic Extractor component, so that performance information can be included in the semantic application graph. The component has been rearranged around a function operation, so that most of its operations can be directly integrated and executed in a FaaS platform.



### 3.10.2 Technology architecture

PEF consists of a REST API service and an adapter to regulate executions of deployed function-based load generators. Compared to the previous version, which was based on Docker containers provide load injection, it is more adapted to a FaaS platform operation. To guarantee the validity of the measurement process and acquired performance information, PEF includes all the necessary details of the executed load generator functions, including an abundance of timestamps across the process, resulting in the reporting of a large number of performance metrics for different stages of the execution.

For each submitted test, the relevant needed function load generator containers are spawned through the FaaS platform API. All the coordination is performed through relevant flows implemented in Node-RED, which also implements the REST API layer to expose the results and the control of the tests.

### 3.10.3 Interfaces/API

The main APIs of PEF, including the input and output format are included in the following tables.

*Table 50: PEF Raw Profiling Data API*

Method	Path/URI	Description	Request/ Parameters	Response
<b>POST</b>	/data	Save the prometheus data for a profiling run inside the PEF. Used by the performance pipeline to push obtained data	{ "activationid": "3f45...", "testFunctionPayload": stringified json "actionName": "flow_1.json", "branchName": "vkatevas", "flow": "flow_1", "cpu": 2, "memory": 32424, "networkReceived": 12342, "networkTransmitted": 65464, "fsReads": 87686, "fsWrites": 95465, "location": OWendpoint}	200
<b>GET</b>	/data	Get all monitoring data. To be used by the cluster creation process inside PEF	-	Array output as above input

The load generator (benchmarking) output is a stringified JSON of the load generation, which includes information on the load setup, the function input as well as average and standard deviation statistics on wait, initialization and execution time. It also includes the raw measurements from which these averages have been calculated as well as indicative timestamps in the load generation process. More information on the specific load generation function outputs can be found in D4.2 and are not included in this document to avoid repetitions.

*Table 51: PEF Benchmarking Data API*

Method	Path/URI	Description	Request/ Parameters	Response
<b>POST</b>	/loadgendata	Save the loadgenerator data inside the PEF.	{ "activationid": "3f45...", "flow": "test",	200

		Loadgenerator data need to be stringified	"activationId":"test", "location":"OWEndpoint" "timestamp":unix time "output": "loadgenerator output" }}	
<b>GET</b>	/loadgendata	Get the loadgenerator data from inside the PEF	-	Array output as above input
<b>GET</b>	/loadgendata /:flow/:branch	Get load generator data for the more recent timestamp, filtered by flow and branch, and grouped by location	-	Array output as above input, with one element per location

Table 52: PEF Clustering Push Data API

Method	Path/URI	Description	Request/Parameters	Response
<b>POST</b>	/clusters	Used by the Clustering process to store the cluster boundaries. Each time the process is run, the new boundaries get inserted into the DB. There is no update so that there is a record of the evolution of the clustering process	{ "owendpoint": "location", "cpu": { "low":numeric value, "medium": numeric value, "high": numeric value }, "memory": { "low": numeric value, "medium": numeric value, "high": numeric value }, "networkReceived": { "low": numeric value, "medium": numeric value, "high": numeric value }, "networkTransmitted": { "low": numeric value, "medium": numeric value, "high": numeric value }, "fsReads": {	200



			"low": numeric value, "medium": numeric value, "high": numeric value }, "fsWrites": { "low": numeric value, "medium": numeric value, "high": numeric value }}	
--	--	--	---	--

Table 53: PEF Clustering Retrieval API

Method	Path/URI	Description	Request/ Parameters	Response
<b>GET</b>	/clusters	Get latest cluster centers (to be used by the Classifier)	-	<pre>{   "cpu": [     {       "category": "low",       "value":         0.005801420285783651     },     {       "category":         "medium",       "value":         0.029655391993717292     },     {       "category": "high",       "value":         0.1852146552507543     }   ],   "fsReads": [ ... ],   "fsWrites": [ ... ],   "networkReceived":     [ ... ],   "networkTransmitted":     [ ... ],   "memory": [ ... ] }</pre>
<b>GET</b>	/clusters/:endpoint	Get latest cluster centers for a given endpoint	-	Same as above
<b>GET</b>	/clustersall	Get cluster centers as they evolve through time	-	Array of above outputs from different runs in the past

Table 54: PEF Function Profile API

Method	Path/URI	Description	Request/ Parameters	Response
<b>POST</b>	/profile	Used to store the classification for a given function	{ "id": auto-increment field "actionName": "flow_1_532...563.json", "branchName": "vkatevas", "flow": "flow_1", "cpu": "low", "memory": "high", "networkReceived": "medium", "networkTransmitted": "low", "fsReads": "low", "fsWrites": "low", "location": OWendpoint }	
<b>GET</b>	/profile	Get all available profiles from PEF	-	Array with elements same as above input
<b>GET</b>	/profile/:actionName	Get classification data for a function that are stored into PEF. It returns the most recent classification (higher auto-increment id) for the function version that is defined in the input, grouped by available locations	-	Array of above inputs with one element per available location
<b>GET</b>	/profile/:actionName/:location	Same as above but filtered for a given location	-	Same as the input in the /POST method
<b>GET</b>	/profile/:flowName/:branch	Get more recent classification data based on the flow name (higher auto-increment. It returns the most recent classification for this flow, grouped by available locations. This is the main method to be used by the Semantic Extractor	-	Array of above inputs with one element per available location

### 3.10.4 Distribution, deployment and configuration

The main PEF tool is available as a code project, following the general DevOps design process of the project, including the main flows file for the Node-RED implementation, a relevant Dockerfile for building the main image as well as all needed declared dependencies of used npm packages. Building the Dockerfile will result in the creation of the relevant PEF image that can be deployed as a server. Due to its function-oriented implementation during Y3, the PEF is accompanied by relevant Docker Images that implement its main functionalities (load generation, profile clustering etc.):

- Load generation function image<sup>21</sup> and flow description<sup>22</sup>
- Clustering process for function resource profiles<sup>23</sup> and flow description<sup>24</sup>
- Classification process<sup>25</sup> and flow description<sup>26</sup>
- Image with relation to the packaged GNU Octave environment needed for model creation as well as model inference<sup>27</sup>
- Images with relation to other helper function packaging (e.g. functions packaged as containers for the Node-RED orchestration execution<sup>28</sup>, artificial delays etc.)

The above images need to be registered as actions in the OpenWhisk FaaS platform in order to be used by the PEF and the Performance Pipeline in the context of the PHYSICS project and beyond.

```
wsk action create loadgen --docker gkousiou/physicspef_loadgenclient:latest
wsk action create classifier -docker vkatevas/node-red_data_classification
wsk action create clustering -docker vkatevas/node-red_data_clustering
```

Specific care has been given to the fact that minimal configuration is needed. Hence all relevant function implementations get the necessary configuration information from the incoming message (e.g. target url in which to push results of a load generation). This results in a very decoupled implementation in which components can be changed at any time with minimal interventions as long as the respective interfaces are respected. This was also evident during the 3<sup>rd</sup> PHYSICS hackathon, in which participants undertook the role of creating versions of these functions based on different algorithms, resulting in a rich externally contributed library of relevant implementations<sup>29</sup>.

The detailed function interfaces as well as JSON specifications for the PEF function inputs are included in detail in D4.2.

### 3.10.5 Individual integration points of PEF with other components

The integration points of PEF with other components include from a functional point of view:

- The available PEF function endpoints (load generation, classification) used by the Performance Pipeline (described in D3.2) during the performance analysis of the target application function

---

<sup>21</sup> [https://hub.docker.com/r/gkousiou/physicspef\\_loadgenclient](https://hub.docker.com/r/gkousiou/physicspef_loadgenclient)

<sup>22</sup> <https://flows.nodered.org/flow/53bf7adbb6ef140ab7e9395c9a9feb1b/in/HXSkA2JLLcGA>

<sup>23</sup> [https://hub.docker.com/r/vkatevas/node-red\\_data\\_clustering](https://hub.docker.com/r/vkatevas/node-red_data_clustering)

<sup>24</sup> <https://flows.nodered.org/flow/48b1f88464634f5601f45e29725a764b/in/HXSkA2JLLcGA>

<sup>25</sup> [https://hub.docker.com/r/vkatevas/node-red\\_data\\_classification](https://hub.docker.com/r/vkatevas/node-red_data_classification)

<sup>26</sup> <https://flows.nodered.org/flow/48b1f88464634f5601f45e29725a764b/in/HXSkA2JLLcGA>

<sup>27</sup> <https://hub.docker.com/r/gkousiou/octavefunction2>

<sup>28</sup> <https://hub.docker.com/r/gkousiou/noderedaction>

<sup>29</sup> <https://flows.nodered.org/collection/zIYKJ6MAudpC/>

- The aforementioned results pushing APIs, used by the Performance Pipeline to store the results generated during its execution
- The results retrieval APIs, from which information on benchmark and profile results of a function are given, is exploited by the Semantic Extractor, in order to enrich the application graph with performance details.

Also, from a semantic point of view, the outputs of PEF (e.g. classification categories, available metrics from benchmarking runs) are also taken into consideration in other dependent processes (e.g. benchmark metrics in global function placement and classification categories in coallocation strategies).

The PEF has also been used as the integration point between the Request Aggregator pattern and the eHealth UC. To do so, a version of the Request Aggregator has been deployed inside PEF to act as the aggregation point of the requests. The relevant implementation appears in Figure 46. The adaptation needed in this case is a minor adjustment to the way the prediction vectors are received and processed, to be compatible with the way the eHealth function needs them.

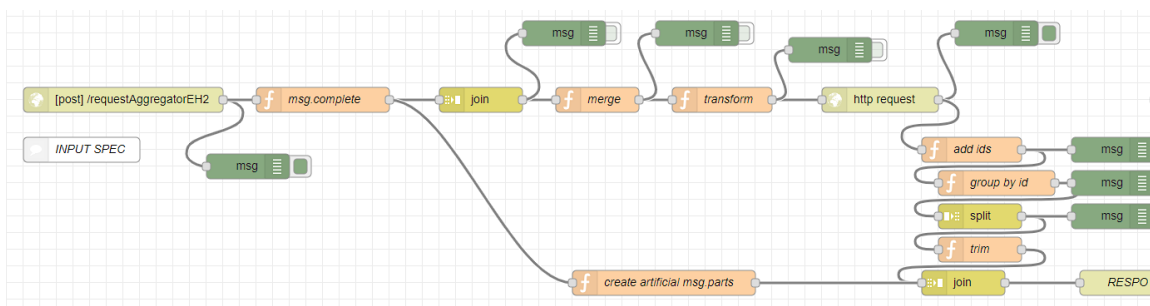


Figure 46 - Request Aggregator Instantiation in PEF for the eHealth Use Case

### 3.11 Global Continuum Placement

#### 3.11.1 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.

The developments during the last period of the project were focused in enabling the placement at the global continuum in a way to take into account different multi-objective algorithms based on both simple (first-fit) and complex (linear programming) policies allowing users to define objectives related to performance, energy and availability. In parallel the component enables the configuration of constraints related to the usage of resources such as the type of architecture (x86\_64, arm64) and level of the continuum (edge, cloud, HPC) to be used for the executions while always allowing the demand of the number of resources such as CPUs and memory. Furthermore, the last version of the Global Continuum Placement component enabled a fine integration with the different PHYSICS components to allow the coherent exchange of information between the components, needed for the optimal placement of the workflows on the continuum. The following sections provide the details of the technology architecture, involved APIs and installation and configuration details.

### 3.11.2 Technology architecture

The Global Continuum Placement Component architecture has been provided in Deliverable D4.2 but it is provided also here as well to facilitate comprehension.

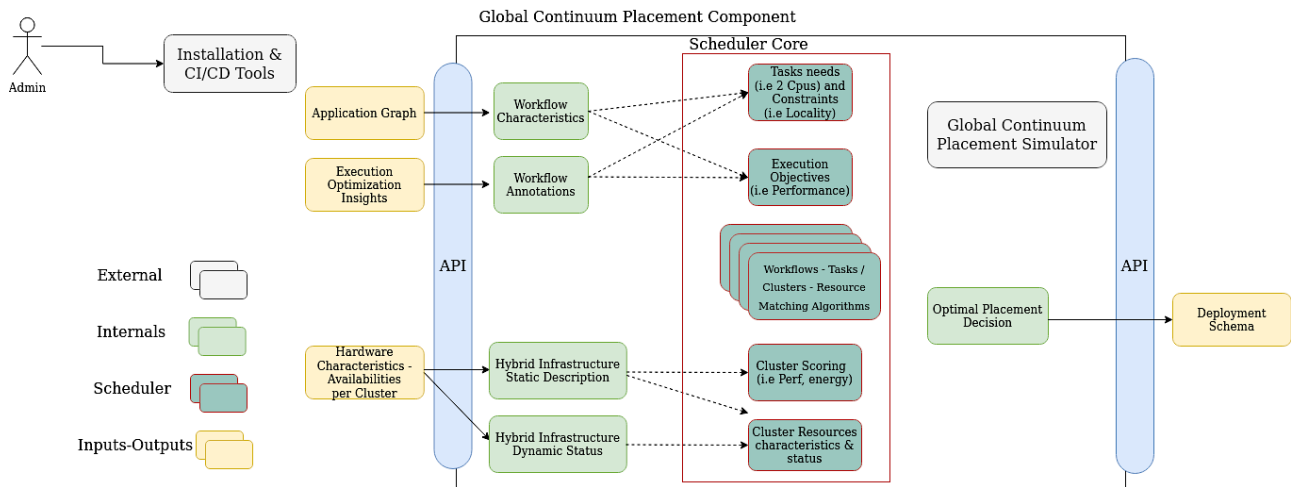


Figure 47 - Global Continuum placement component

As shown in the Figure 47 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 push 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.

### 3.11.3 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 Table 55 shows the API operations for scheduler:

Table 55 - Global Continuum API for the scheduler

Method	Path/URI	Description	Expects/ Parameters	Response
--------	----------	-------------	------------------------	----------

<b>POST</b>	/api/v1/clusters	Initialize scheduler. Provide platform characteristics to the scheduler	JSON	200 (Success), 400 (Failed)
<b>POST</b>	/api/v1/applications	Schedule the workload. Run the scheduler on the given workload. Returns placement mapping for each allocatable task.	JSON	200 (Success) JSON, 400 JSON (Failed)

The Table 56 shows the API operations for monitoring:

*Table 56 - Global Continuum API for monitoring*

<b>Method</b>	<b>Path/URI</b>	<b>Description</b>	<b>Expects/ Parameters</b>	<b>Response</b>
<b>GET</b>	/api/v1/healthz	Information about monitoring. Check service status.	-	200 (Success), 400 (Failed)

The deliverable 4.2 contains examples of the latest version of the Global Continuum Placement features and specific cases of JSON files to use for the scheduler initialization and workload scheduling POST APIs methods mentioned above. In particular, the extensions involved the adaptation of the APIs JSON structure to enable a fine integration with the rest of the PHYSICS components and to support the new features. The new features are related with the updates in the objectives to cover Energy, Performance and Availability along with the extensions to support the different scheduling algorithms first-fit, FOA and FOA-e. The detailed structure of the JSON to be used can be found in detail on the GitHub account where the code is hosted<sup>30</sup>.

In terms of workflow annotations, the following are finally supported:

- **cores**: number of CPU cores
- **memory**: memory in MB
- **locality**: cluster type one of: "HPC", "Cloud", "Edge", and "On-premise".
- **architecture**: Hardware architecture, one of: "x86\_64", "arm64"
- **optimizationGoal**: could be one or more of the following "Energy", "Performance", or "Availability"
- **importance**: the level associated to each selected goal : "Low", "Medium", "High"

#### 3.11.4 Distribution, deployment and configuration

The implementation of the Global Continuum Placement component is done using Python programming language. The code can be found in PHYSICS registry. 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 on a Gitlab repo. Various tests are also implemented and incorporated in the CI/CD pipelines with a current coverage of 80%. The whole software can be packaged and installed as a container.

For the initial setup users can use the following commands:

```
git clone https://github.com/RyaxTech/global-continuum-placement.git
https://gogs.apps.ocphub.physics-faas.eu/WP4/Global-Continuum-Placement.git
```

<sup>30</sup> <https://github.com/RyaxTech/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:main
```

A complete example of usage has been added in D4.2 and some examples can be also found in the repository where the source code is available.

## 3.12 Distributed In-Memory Service

### 3.12.1 Overview

The DMS component has been reimplemented since September 2022. The first version of this component was based on Pocket. This research prototype was based on Apache Crail project. However, this project has not been maintained since July 2022. The new DMS service implementation is now based on KeyDB. KeyDB is a multithreaded fork of the in-memory Redis database. KeyDB can run on several cores taking advantage of multi-core computers. KeyDB provides persistence, replication and security. The DMS provides a set of OpenWhisk actions (functions) for accessing the in-memory datastore (KeyDB).

### 3.12.2 Technology architecture

The architecture description of the DMS can be found in deliverable D4.2 *Cloud Platform Services for a Global Space-Time Continuum Interplay V2*. The architecture consists of two main components (Figure 48) the *HAProxy or Kubernetes Service*, one or several *KeyDB instances*. A *KeyDB instance* stores some data. The *HAProxy or Kubernetes Service* component is in charge of distributing the requests among *KeyDB instances*.

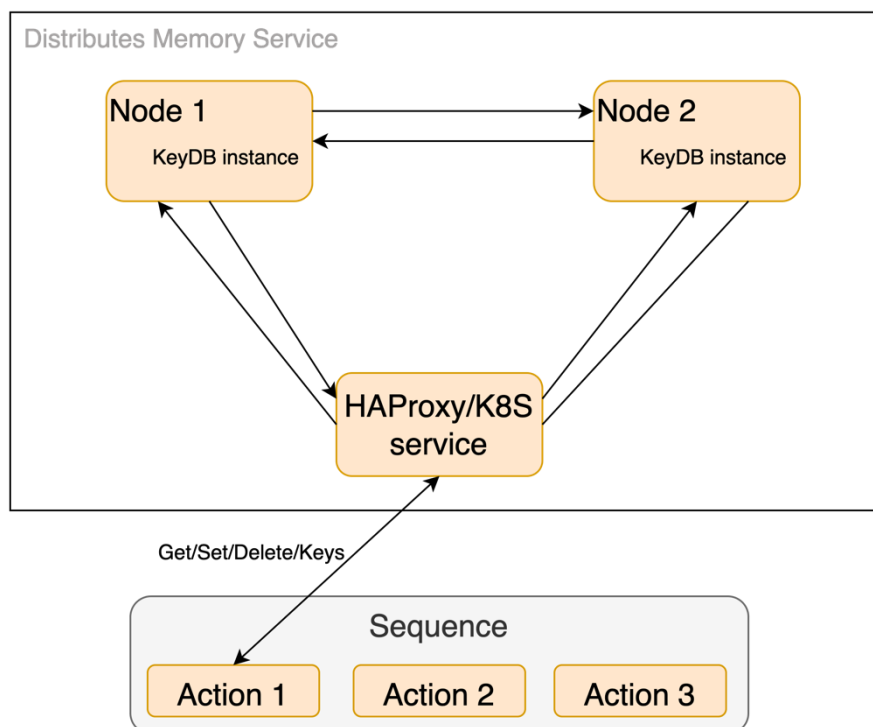


Figure 48 - DMS component



Data shared between functions can be persistent, which means that the data remains in the DMS even if the service is stop or fails.

### 3.12.3 Interfaces/API

The API of the DMS provides these operations:

Method	Parameters	Description
<b>KeyDB()</b>	Ip, port, dbid	Receives the KeyDB service address (IP:PORT) and the database identifier, opens a connection with the KeyDB Kubernetes service and returns a KeyDB connection object to be used in the rest of the methods.
<b>Close()</b>		Closes a connection with the KeyDB service.
<b>Keys()</b>	regex	Receives a regular expression to filter the keys stored in the DMS; it returns the list of keys that matches the regular expression.
<b>Set()</b>	Key, value	Receives a key and the associated data to be stored. It returns <i>OK</i> if the data has been stored, or an error otherwise.
<b>Get()</b>	key	Receives a key and returns the stored value associated to that key or an error, if the key does not exist.
<b>Getset()</b>	Key, value	Receives a key and a value. It reads the stored value which is replaced by the new value. It returns the value associated to that key. If the key does not exist, it behaves like the set method.
<b>Append()</b>	Key, value	Receives a key and a value. If the key is already stored, the value is appended to the existing one. If the key does not exist, it behaves like the set method. If the stored values are larger than 512MB, max per key) the method returns an error.
<b>Delete()</b>	key	Receives the key to be deleted. If the key does not exist, it returns an error message.

*Table 57 - DMS-API*

### 3.12.4 Distribution, deployment and configuration

The DMS component distribution is available in the PHYSICS git repository <https://gogs.apps.ocphub.physics-faas.eu/WP4/DMS.git>. Each sub-component is containerized 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 are created, the DMS-DEPLOY Jenkins pipeline is executed. The DMS-DEPLOY Jenkins pipeline deploys the DMS components in the PHYSICS infrastructure (AWS or other cloud provider). This CI/CD pipeline is executed each time a new version of the DMS code is uploaded into the PHYSICS git repository.

## 3.13 Adaptive Platform Deployment, Operation & Orchestration

### 3.13.1 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. The final version of the component includes new features, like the improvement of the Workflow Operator CRD Operator and the Translator applications, and the implementation of a new subcomponent (Monitoring-Alerts) responsible for doing the monitoring and assessment of the QoS constraints needed for the migration of functions executions in different clusters. In the case the QoS constraints defined for the functions or infrastructure don't meet the values expected by the users, this component sends QoS violation notifications to a message broker which is connected to other subcomponents responsible for the update or migration of the corresponding functions.

### 3.13.2 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 parse 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 is 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 implemented a Kubernetes Operator (WorkflowCRD) that defines this specific CRD that describe the workflow 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 49 shows the flow of actions to be executed when deploying an application by the Orchestrator:

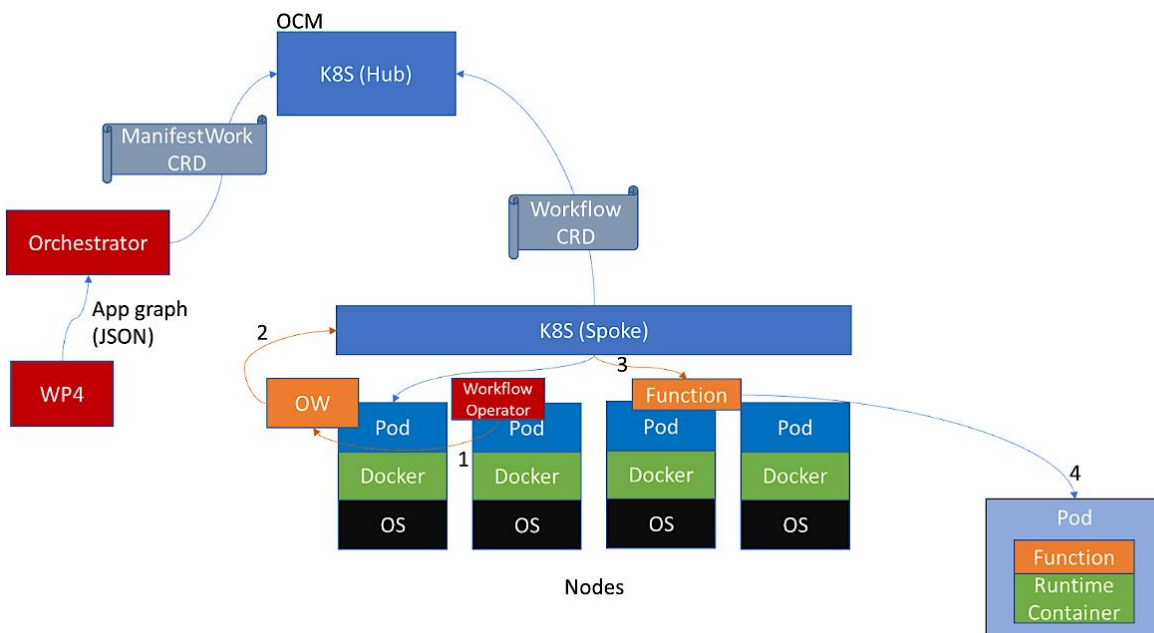


Figure 49 - Orchestrator flow

These functions are monitored by tools like Prometheus. This way the Monitoring-Alerts component can gather the metrics defined in the QoS definitions associated to these functions and can perform

the assessment of them. 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.

### 3.13.3 Interfaces/API

The available endpoints were described in the deliverable D4.1 (section 6.2.3 Interfaces and integrations) and updated in D4.2, 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
<b>GET</b>	/api/v1/function/{id}	Get information details of a function that exists in the function catalogue.
<b>POST</b>	/api/v1/function	Create or register a function in the functions catalogue.
<b>PUT/PATCH</b>	/api/v1/function/{id}	Update a function in the functions catalogue.
<b>DELETE</b>	/api/v1/function/{id}	Delete a function from the functions catalogue.
<b>GET</b>	/api/v1/functions	List all the functions from the catalogue.

The following table shows the API operations for invocation of runtime actions (RPC API call style):

*Table 36 - Orchestrator component runtime RPC API*

Method	Path/URI	Description
<b>POST</b>	/api/v1/functions/deploy	Install a standalone function in the target FaaS engine/platform.
<b>POST</b>	/api/v1/functions/undeploy	Uninstall a function from the target FaaS engine/platform.
<b>POST</b>	/api/v1/functions/run	Execute a standalone function in the target FaaS engine/platform.

Finally, the Translator application offers the following operation:

*Table 58 - Orchestrator component runtime RPC API*

Method	Path/URI	Description
<b>POST</b>	/api/v1/deploy	Creates a ManifestWork per function included in the JSON description of the application (deployment graph)

### 3.13.4 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<sup>31</sup> 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<sup>32</sup>. For the deployment destination (testbed) we will use OpenShift as CaaS (Container as a Service) installed in

<sup>31</sup> <https://gogs.apps.ocphub.physics-faas.eu/WP4/orchestrator>

<sup>32</sup> (<https://openapi-generator.tech/>).

the PHYSICS testbed hosted by Amazon AWS. The CD pipeline will manage the deployment of the components in the assigned namespaces of the cluster.

### 3.14 Scheduling Algorithms (Local Adaptive Scheduler)

#### 3.14.1 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.

In PHYSICS architecture, the scheduling algorithms are implemented as OpenWhisk-Kubernetes coordinated scheduling and are responsible for actual execution of the tasks on the selected underlying computing resources of one single cluster.

The final version of the local adaptive scheduler involves developments mainly in Kubernetes scheduling while keeping a close coordination with the OpenWhisk scheduler to correctly set the adapted Kubernetes policies to optimize the FaaS applications executions. The new scheduling algorithm implemented for the local adaptive scheduler is called *LayersLocality* and allows the minimization of cold start delays by favouring the placement of tasks on nodes that already have layers of the containers to be deployed.

#### 3.14.2 Technology architecture

Based on the analysis provided in D5.2 and our focus to provide a scheduling algorithm that will minimize the time a task is executed, by addressing the delays related to the downloading of containers and their layers. Hence, the proposed scheduling algorithm has been designed 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 Kubernetes codebase.

We have implemented a new version of *imagelocality* plugin named *LayersLocality* which is based on the following basic concepts:

- The layers available with their size on each of the nodes at startup
- For each new pod compute a score regarding the cumulative size of already present layers.
- Based on the above try to favor the execution of functions on nodes where layers of the containers to be deployed already exist, which will help in minimizing the download time of images and hence the cold start of containers.
- For this we had to extend not only Kubernetes itself but also its Container Runtime Interface (CRI). In our case we considered CRI-O which is one of the most widely known Kubernetes CRI.

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<sup>33</sup>.

So, in this regard we collected the layers’ info from the manifest CRI and added it in the *ImageSpec.annotations* field in the CRI API, see Git repository<sup>34</sup>.

For that we implemented 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

<sup>33</sup> <https://github.com/opencontainers/image-spec/blob/main/manifest.md>

<sup>34</sup> <https://github.com/kubernetes/cri-api/blob/master/pkg/apis/runtime/v1/api.proto#L673>

interface to be propagated to the internal *NodeStatus.Images* interface that is already accessible by the scheduler. We implemented our solution in our Kubernetes fork<sup>35</sup>. Besides the changes taking place within Kubernetes, we also extended 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).

For that we have modified the Container Runtime CRI-O to get available layers name and size on node and send them through annotations (without API change) on these forks<sup>36,37</sup>

Finally, as mentioned in D5.2, we have pushed the LayersLocality Scheduler to the Kubernetes community by proposing to integrate it in the upstream version.

### 3.14.3 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<sup>38</sup>

Nevertheless, to support the changes mentioned above certain modifications had to be made to enhance the default API with the container layer details. For that we had to modify the ContainerImage API from the default version here<sup>39</sup> to also contain a map with the layers ID and layer size. This allowed us to GET these new details related to the container layers and perform the right scheduling decision taking into account these details as well.

### 3.14.4 Distribution, deployment and configuration

As mentioned previously the code of the new LayersLocality Scheduling algorithm component is implemented within different open-source projects. The whole effort, tests and different techniques used can be found online here<sup>40</sup>

To configure and deploy the 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
```

---

<sup>35</sup> <https://github.com/RyaxTech/Kubernetes/compare/v1.22.6...image-layer-locality-scheduler>

<sup>36</sup> <https://github.com/RyaxTech/cri-o>

<sup>37</sup> <https://github.com/RyaxTech/cri-o/compare/v1.22.1...image-layer-locality-scheduler>

<sup>38</sup> <https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.23/>

<sup>39</sup> <https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.23/#containerimage-v1-core>

<sup>40</sup> <https://github.com/RyaxTech/k8s-container-layer-locality>

```
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.

Pull the particular Kubernetes fork with:

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

Build Kubernetes binaries with embed libc to avoid portability issues:

```
CGO_ENABLED=0 make all
```

Create a dockerfile for the Kubernetes API server

```
cat > _output/Dockerfile.kube-apiserver <<EOF
FROM busybox
ADD ./local/bin/linux/amd64/kube-apiserver /usr/local/bin/kube-apiserver
EOF
docker build -t ryaxtech/kube-apiserver:latest --file _output/Dockerfile.kube-
apiserver ./_output
docker push ryaxtech/kube-apiserver:latest
docker tag ryaxtech/kube-apiserver:latest ryaxtech/kube-apiserver:v1.22.6
docker push ryaxtech/kube-apiserver:v1.22.6
```

Create a dockerfile for the Kubernetes controller:

```
cat > _output/Dockerfile.kube-controller-manager <<EOF
FROM busybox
ADD ./local/bin/linux/amd64/kube-controller-manager /usr/local/bin/kube-controller-manager
EOF
docker build -t ryaxtech/kube-controller-manager:latest --file _output/Dockerfile.kube-
controller-manager ./_output
docker push ryaxtech/kube-controller-manager:latest
docker tag ryaxtech/kube-controller-manager:latest ryaxtech/kube-controller-manager:v1.22.6
docker push ryaxtech/kube-controller-manager:v1.22.6
```

Create a dockerfile for the scheduler:

```
cat > _output/Dockerfile.kube-scheduler <<EOF
FROM busybox
ADD ./local/bin/linux/amd64/kube-scheduler /usr/local/bin/kube-scheduler
EOF
docker build -t ryaxtech/kube-scheduler-llocality:latest --file _output/Dockerfile.kube-
scheduler ./_output
docker push ryaxtech/kube-scheduler-llocality:latest
docker tag ryaxtech/kube-scheduler-llocality:latest ryaxtech/kube-scheduler-llocality:v1.22.6
docker push ryaxtech/kube-scheduler-llocality:v1.22.6
```

Some experiments related to the scheduler can be found in D5.2 and on the GitHub repository. The coordination with OpenWhisk and the new scheduler developed in Kubernetes goes through the webhook mechanism which is described in the following section. Moreover, the upstream work is being done related to this with the expectations of merging it in the relevant upstream projects so that this does not need to be done and will get it in any vanilla k8s deployment.

### 3.15 Resource Management Controllers

#### 3.15.1 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 oversees providing the needed APIs for other PHYSICS components, so that they can better control both the infrastructure itself and the applications running on top. This means offering the functionality and APIs to be able to perform wiser scheduling and co-allocation decisions, as well as enabling applications (in PHYSICS use cases, functions) deployments across different clusters, including information about energy consumption/estimations (through Kepler). In addition, it includes the automation needed for onboarding new clusters with required configuration and PHYSICS components, as well as connecting them to the other PHYSICS components in the hub cluster.

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.

In the second phase of the project, we have focused on:

- Enhancing the K8s API extensions (through the workflow CRD) to provide all the needed information from components in the upper layers (the ones developed in WP3 and WP4) to the relevant infrastructure components (the ones developed in WP5, in this case the co-location engine)
- Provide a cluster onboarding mechanism that automatically detects new clusters being added to the hub and deploys relevant PHYSICS components (in our case the semantic components and some benchmarking load) on them. It also connects them to the PHYSICS component in the HUB – in this case the Reasoning Framework.
- Enhancing the Kepler project to obtain energy consumption metric (estimations) on top of cloud providers (AWS, Azure) and with enough granularity (sampling) for function as a service use cases. This is leveraged by other PHYSICS components (e.g., semantic (WP5) or scheduler (WP4)).

#### 3.15.2 Technology architecture

We have developed the next subcomponents to support the resource management controllers:

- Scheduler selection webhook<sup>41</sup> - This subcomponent 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. In this case we enforce the locally aware scheduler for the pods that runs the functions, to limit the impact of creating new containers. In addition, this webhook also calls the co-location engine to obtain the affinity/anti affinity rules to be used.
- The Workflow Custom Resource Operator (Workflow CRD) - CRD is one of the native Kubernetes mechanisms for extending its API. It is used to define a new object type in Kubernetes and create a controller (Operator) for reacting to them. Specifically, the Workflow CRD is used as an API between PHYSICS components, where all the required information about the sequence of functions is available, including metadata added by other PHYSICS components, such as performance profiling information. The API and the operator reacting to it has been enhanced during the project duration to ensure new information/parameters were

---

<sup>41</sup> <https://wordpress.org>



included upon need by other PHYSICS component. For example, the defaultParams needed to deploy functions across clusters or the performance profile information needed by the co-location engine.

- The Event-Driven Cluster Onboarding mechanism based on Knative Serverless: This new component is in charge of detecting new cluster being added to the hub (OCM managed clusters) and perform a set of actions for their onboarding, more specifically:
  - leverages Knative services and APIServerSource service for detecting the event and triggering the actions
  - leverages OCM to deploy Kubernetes objects in the remote clusters
  - leverages Submariner for the communication across clusters
  - leverages Kepler for the energy consumption metrics.
  - run some benchmarks so that its performance can later be evaluated by the semantic component
  - deploy the semantic PHYSICS component (note others can be added too) and pass to it the needed information about the benchmark so that it can make use of Kepler information to estimate the energy consumption score of the new cluster
  - ensure the PHYSICS component(s) is reachable through submariner and pass the endpoint to the Reasoning Framework

The relationship between different components in the context of cluster onboarding can be seen in the next figure (Figure 50):

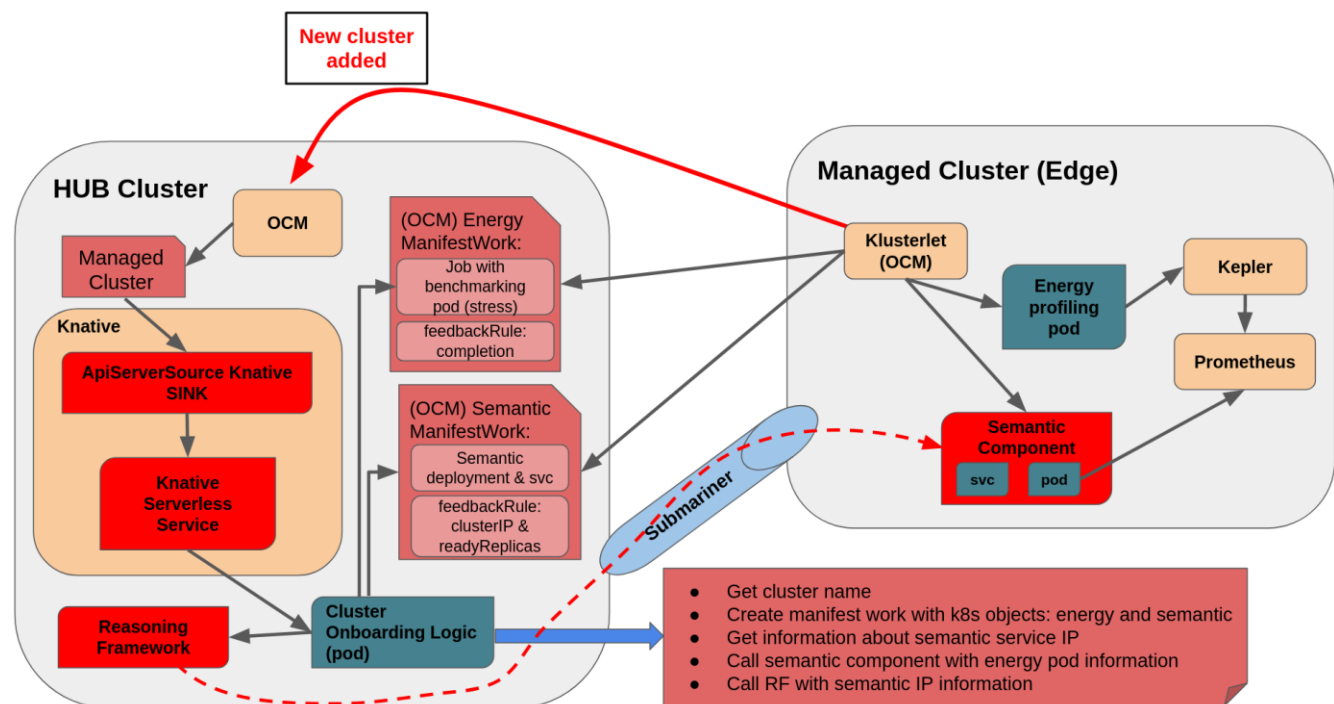


Figure 50 - PHYSICS cluster onboarding

In addition, we contributed to several upstream components in order to make them work better in the cloud compute continuum and especially with edge nodes and function as a service use cases. The components we contributed to are:

- Microshift<sup>42</sup> - A small form factor Openshift that is optimised for deployment on edge nodes. We contributed with testing, bugs identification and engagement with the community to fix them.
- Submariner<sup>43</sup> - This is a tool for connecting kubernetes clusters. We pushed bug reports and fixes in order to make it work better with Microshift and ovn-kubernetes CNI.
- Kepler<sup>44</sup> - This is a prometheus exporter that uses eBPF and linux kernel tracepoints to obtain or estimate the energy consumption per node/pod. We worked together with the community to make it work on top of cloud providers (i.e., on top of Vms) as well as with enough sampling frequency to cover Function as a Service use cases.
- Knative<sup>45</sup> - This is a platform-agnostic solution for running serverless deployment. We have engaged in discussions with their upstream community due to the similarities with OpenWhisk and the work on the WorkflowCRD. As a result we have a hackathon about building an operator around Knative to do similar things to the WorkflowCRD. In fact the WorkflowCRD has been enhanced with Knative support as part of this effort.

Figure 51 shows an overview of the different components and their relations:

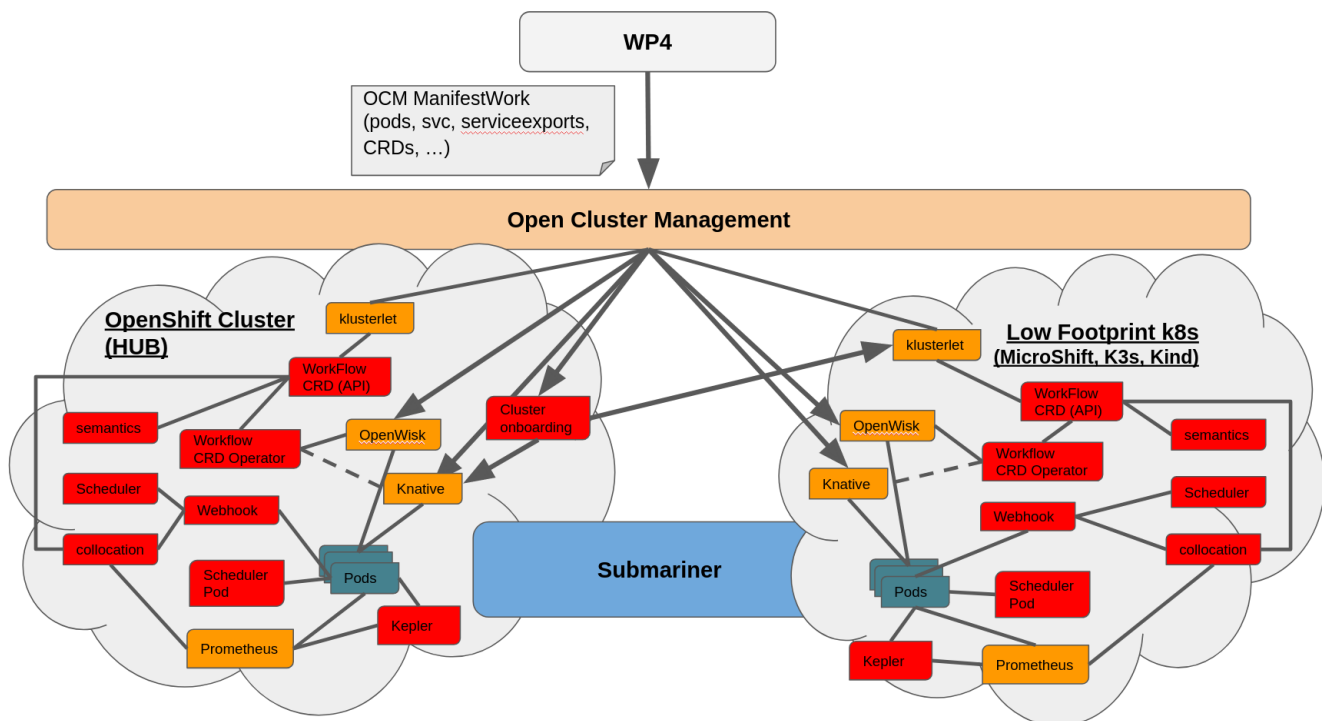


Figure 51 - Resource Management Components and interactions overview

### 3.15.3 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

<sup>42</sup> <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>

<sup>43</sup> <https://www.okd.io/#what-is-okd>

<sup>44</sup> <https://sustainable-computing.io/>

<sup>45</sup> <https://knative.dev/docs/concepts/>



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).

### Workflow CRD

The Workflow CRD is managed by a Kubernetes Operator<sup>46</sup>. This is in charge of managing the events once CRs of type Workflow are created/deleted/updated.

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 spec, and how internally the operator manages it. Like 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 <yaml file>
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>

### Cluster Onboarding

The cluster onboarding components<sup>47</sup> make use of the Knative/OCM/Kubernetes APIs for detecting and triggering the actions:

- Detection: Knative APIServerSource object watching for OCM ManagedCluster objects creation

---

<sup>46</sup> [https://aws.amazon.com/what-is-aws/?nc1=h\\_ls](https://aws.amazon.com/what-is-aws/?nc1=h_ls)

<sup>47</sup> <https://github.com/luis5tb/physics-cluster-registration>

- Triggering: Knative Service that runs the cluster onboarding logic, and that handles the scaling up/down depending on the events – i.e., if there are no clusters being added for some time it will remove the service pods to save resources.

As for the logic itself, it is a Python Flask service that reacts to POST events, which are generated by the APIServerSource. The API for the user/admin is basically to onboard a new cluster with OCM. This creates a) create an OCM ManagedCluster object which triggers b) the Knative APIServerSource, which in turns calls c) the Knative Service running the Python Flask service (POST). Finally, the service is d) creating OCM ManifestWork objects that contains the Kubernetes objects to create in the remote cluster (k8s Job and Semantic service) and the information to retrieve (Semantic service IP) and e) calling the Reasoning Framework and Semantic component REST APIs (POST) with the relevant information about the IPs and k8s jobs.

### 3.15.4 Distribution, deployment and configuration

The Webhook mechanism is composed of several yaml files that should be deployed on the Kubernetes cluster<sup>48</sup>. 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 -- `kubectl apply -f XXX.yaml`. 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 is implemented as a Kubernetes Operator, therefore it is also be distributed as a set of yamls to deploy on the Kubernetes cluster. The configurable options are implemented in the operator and are 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.

For the Cluster Onboarding mechanism, it is distributed as an open source project (Apache License), easily installable with a set of yamls, following instructions in here: <https://github.com/luis5tb/physics-cluster-registration/blob/main/README.md>. For making modifications to the code or changing the default behaviour the steps are the next:

- change the cluster-registration.py python flask application as needed
- build a new container image using the provided dockerfile
- push the image to a container repository (e.g., quay.io or dockerhub)
- change the knative service definition so that it uses this new container instead of the default one

For Kepler, we have made our contributions upstream, therefore is directly distributed when using the latest released version of Kepler. It is installed in the default way, detailed in the project website for the different options: <https://sustainable-computing.io/installation/kepler/>. As for the configuration, the same applies, see: <https://sustainable-computing.io/usage/general-config/>.

## 3.16 Co-Allocation Strategies

### 3.16.1 Overview

The Co-location strategies component was designed and implemented under the scope of task T5.4 *Optimized service co-location strategies*. The description of this component was reported in deliverable D5.2 *Extended Infrastructure Services with Adaptable Algorithms* Scientific Report and Prototype Description V2. This component is triggered just before a new pod (function) is going to

---

<sup>48</sup> <https://www.gartner.com/smarterwithgartner/the-science-of-devops-decoded#:~:text=Gartner%20defines%20DevOps%20as%20a,between%20operations%20and%20development%20teams>.

be created in the infrastructure. The co-location component analyses the cluster status (pods running at each node and node resources), the requirements of the function to be deployed (defined in a workflow), the possible interferences of the pod with pods already running on the cluster nodes 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.

In this second version of the co-location component, the internal architecture has been modified by adding new components to analyze the resource consumption of function pods and the function performance in different scenarios (standalone execution, co-located with other functions). This information is used to find the most suitable nodes for the pod deployment.

### 3.16.2 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 52)

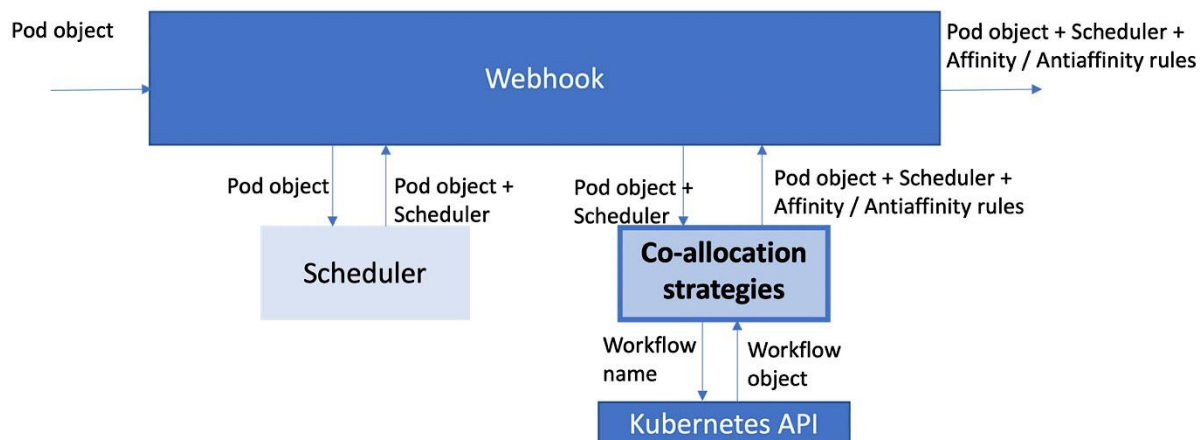


Figure 52 - Co-allocation invocation Technology architecture

The co-location component is made of six sub-components: the *cluster information*, the *cluster status*, the *Function metrics and interferences*, the *co-location database*, the *Data collector* and the *Rules generator* (Figure 53). There are three types of sub-components, the sub-components that are executed periodically (grey boxes in Figure 53), the database and the sub-components that are executed when a new pod is intercepted by the mutating webhook.

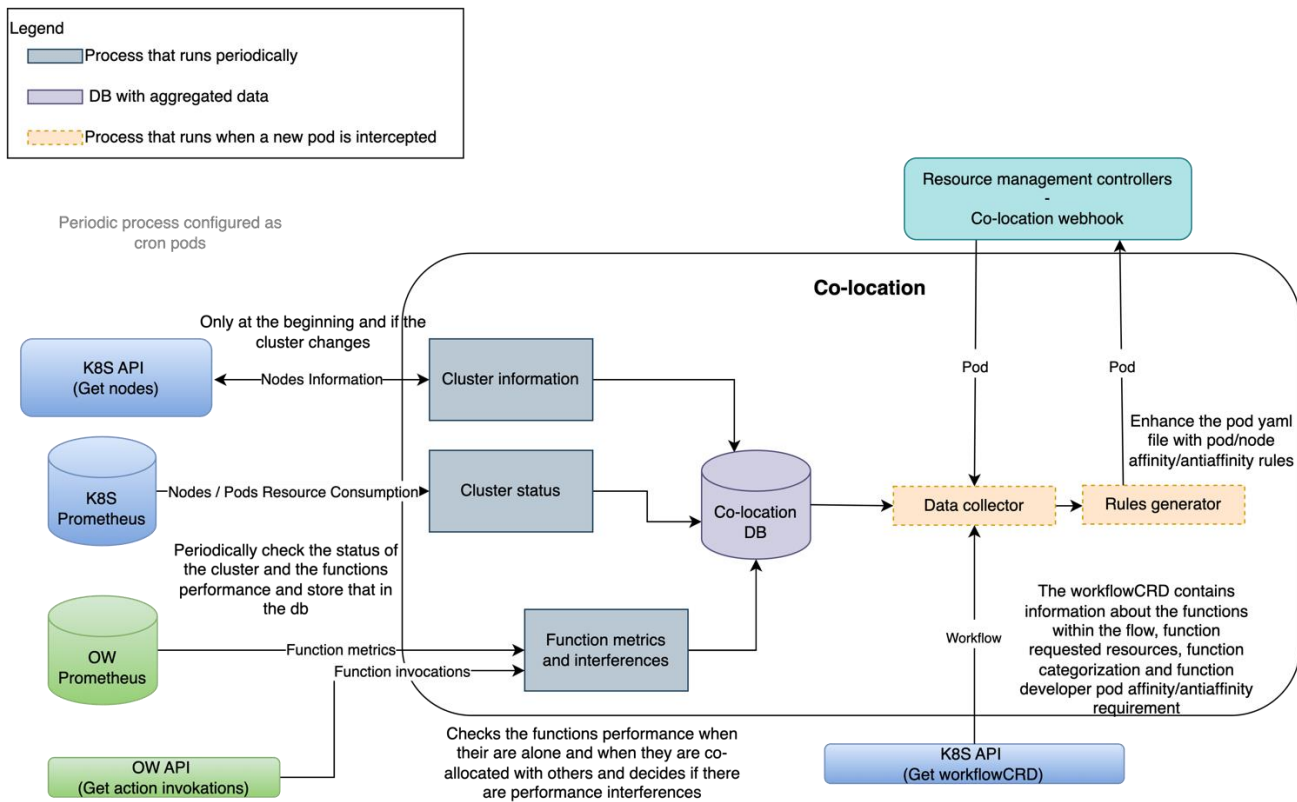


Figure 53 - Co-allocation strategies component internal architecture

Several processes run periodically to collect data from the Kubernetes cluster and OpenWhisk using their own API and their own Prometheus instances. The performance of functions execution and interferences are analyzed. This information is stored in a time-series database (*Co-location DB* in Figure 53) and is used by the *Data collector* sub-component to define the best co-location strategy. That information is sent to the *Rules generator* sub-component that modifies the pod YAML object, intercepted by the mutating webhook, adding pod/node affinity/anti-affinity rules. An example of the output produced by the co-location component is shown in Code 1. In this case two rules have been added. The first one is a node affinity rule to ensure that the function is executed in a node that has a disk of type SSD and a GPU. The second rule is a pod anti-affinity rule which prevents the function from being executed in a node where a pod of the function Model training is being executed.

```
apiVersion: v1
kind: Pod
metadata:
  name: wskowdev-invoker-00-1-guest-hello
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: disktype
                operator: In
                values:
                  - ssd
              - key: gpu
```

```

      operator: In
      values:
      - yes
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: openwhisk/action
            operator: In
            values:
            - Model training
        topologyKey: topology.kubernetes.io/zone
  containers:
  - name: wskowdev-invoker-00-3278-guest-hello
    image: "gkousiou/noderedhelloaction"

```

*Code 1 - Affinity/Antiaffinity rules example*

### 3.16.3 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 59 - Co-Allocation/API-get affinities*

Method	<b>get_affinities(pod)</b>
<b>Input</b>	pod: pod object
<b>Success response</b>	pod: pod object enhanced with the affinity and anti-affinity rules
<b>Error response</b>	<ul style="list-style-type: none"> <li>- Workflow name not defined in pod object</li> <li>- Missing workflow object</li> <li>- Not enough resources to allocate the pod</li> </ul>

### 3.16.4 Distribution, deployment and configuration

The Co-location strategies component is implemented in Python, some of its sub-components are part of the mutating webhook logic. The Co-location strategies component has been integrated in the PHYSICS 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/WebhookCo-locationStrategies.git>.

The Co-locationStrategies-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-location component is uploaded to the PHYSICS git repository. Next the deployment of the webhook and the co-location strategies sub-components will be done by the PHYSICS platform administrator. The Co-locationStrategies-DEPLOY Jenkins pipeline deploys the webhook and co-location logic in the PHYSICS AWS platform.

## 4. REUSABLE ARTEFACTS MARKETPLACE IMPLEMENTATION

This chapter covers the design and implementation of the final version of the Reusable Artefacts Marketplace (RAMP) application. RAMP has been refurbished to cater to a broader audience to facilitate easy access to the various project solutions. Special attention has been given to emphasizing reusable cloud patterns in FaaS applications.

In addition to project's solutions, the RAMP offers training materials, including informative videos and webinars, aiding users in leveraging the full potential of the platform's offerings through demonstrations based on real-life use cases. The enhancements in the marketplace facilitate the exploitation of the project results to a more extensive degree.

### 4.1 Overview

RAMP implementation relies on WordPress<sup>49</sup>, 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 specialized functionality, and can be customized 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.

In this version, RAMP has updated to foster a better user experience, which include:

- **Enhanced User Interface (UI):** The revised UI now includes market-oriented texts that not only resonate better with the end-users but also streamline navigation, paving the way for a more intuitive user journey.
- **Expanded Artefact Repository:** Towards offering a more robust solution repository, we have augmented our marketplace with a wider array of artefacts, thereby broadening the spectrum of resources available to aid in FaaS applications.
- **X (former Twitter) Integration:** To foster better community engagement and keep users updated with the latest project developments, a Twitter feed integration has been facilitated to bring real-time project posts directly to the marketplace platform.

These upgrades provide a more responsive, user-friendly, and resource-rich marketplace, the details of which are presented in the following subsections.

### 4.2 Technology architecture

The RAMP service developed 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

---

<sup>49</sup> <https://wordpress.org>



facilitates the DNS server of the RAMP which is provided by the Amazon Route 53 service (see also Figure 54).

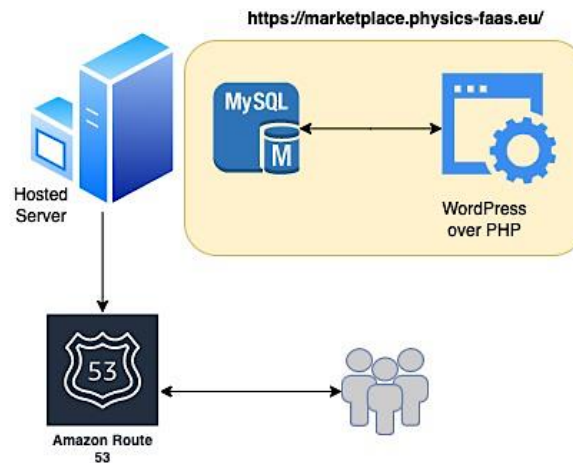


Figure 54 - RAMP Architecture

In the current RAMP version, the user categorizations include three distinct groups: administrators, registered users, and visitors. The administrator, encompassing INNOV and GFT, retain the privilege to modify the website's structure and augment its content pool directly through a web browser, necessitating access to the site's URL and respective logins.

Enhancements have been introduced in the access tiers given to visitors and registered users. In the upgraded version:

- **Visitors:** To allow greater inclusivity and accessibility, visitors are now granted open access to view the artefacts, thereby enabling them to get insights and leverage the available resources to a substantial extent. The primary distinction between visitors and registered users lies in the capability to upload assets to the RAMP, a functionality exclusive to the registered users.
- **Registered Users:** Apart from enjoying unrestricted access to the resources available on the platform, registered users are empowered to upload new assets, thereby fostering a rich and evolving repository. Further, they can engage directly with the consortium or artefact owners to request specific information or propose modifications to an artefact, promoting a collaborative and iterative development environment.
- **Administrative Review:** To maintain the quality and relevance of the content, all new uploaded assets undergo a meticulous review by the administrators before being granted approval for integration into the site.

*This user structure not only ensures the streamlined operation of the platform but also facilitates a harmonious collaboration among diverse user groups while upholding the integrity and quality of the content hosted on the RAMP*

### 4.3 Artefacts

In its infancy, RAMP carries a foundational set of artefacts mainly sourced from the project's technical WPs. However, it embodies a vision of growth, as these project's results are disseminated through various channels, including conferences, workshops, webinars, and scientific publications. Even at this stage, it has successfully managed to onboard a notable number of assets from external

contributors, laying a solid foundation for a rich and diverse library of artefacts. The existing collection encompasses a broad spectrum of offerings such as:

- Patterns, Flows, and Functions: Pertaining to the FaaS domain, facilitating an intuitive understanding and application in various related settings.
- Services, Semantics, and Interoperability Tools: Offering essential tools and services promoting efficient interoperability in diverse environments.
- Optimization Artifacts: Enhancing application deployment at the edge, promising optimized performance.
- End-to-End Use Cases: Leveraging the project's offerings to furnish users with practical, real-world applications of the resources available on RAMP.
- Educational Resources: Encompassing a wide array of materials such as tutorials, webinars, and workshops, honing skills in functional programming.

The following table summarizes the artefacts hosted on RAMP as of September 2023. It should be noted that these numbers are expected to grow further before the end of the project as the technical work progresses towards finalization.

*Table 39 - Summary of Artefacts hosted on RAMP*

Type	Number of Artefacts per Owner		Total
	PHYSICS	External Contributor	
Pattern	9		9
Flow	6	4 + 1 collection of 26	11
Function	1		1
Semantics	1		1
Service	4	1	5
Dataset	1		1
Use Case	3		3
Training Resource	5		5
<b>Total</b>	<b>30</b>	<b>6</b>	<b>36</b>

#### 4.4 Distribution, deployment and configuration

RAMP is deployed in a dedicated cloud-based Hosting and Database Server hosted in Germany. The utilized 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.

#### 4.5 User Story

The PHYSICS market platform is structured thoughtfully to include several distinct pages, each serving a specific purpose to enrich the user experience. These pages are delineated as follows:

- Homepage
- About
- Assets
- Use Cases



- Training Videos
- FAQs

The **homepage**, represented in Figure 55, is crafted to immediately engage users, offering previews of the most recent assets and use cases alongside PHYSICS social media links and a live feed of the latest Twitter posts to keep users informed of the most recent project's developments. Registration and sign-in options are available to facilitate easy access for both new and existing users. A further feature of the homepage is a section at the bottom that highlights the latest assets introduced in the marketplace, complemented by a direct link that enables registered users to contribute by uploading new assets.

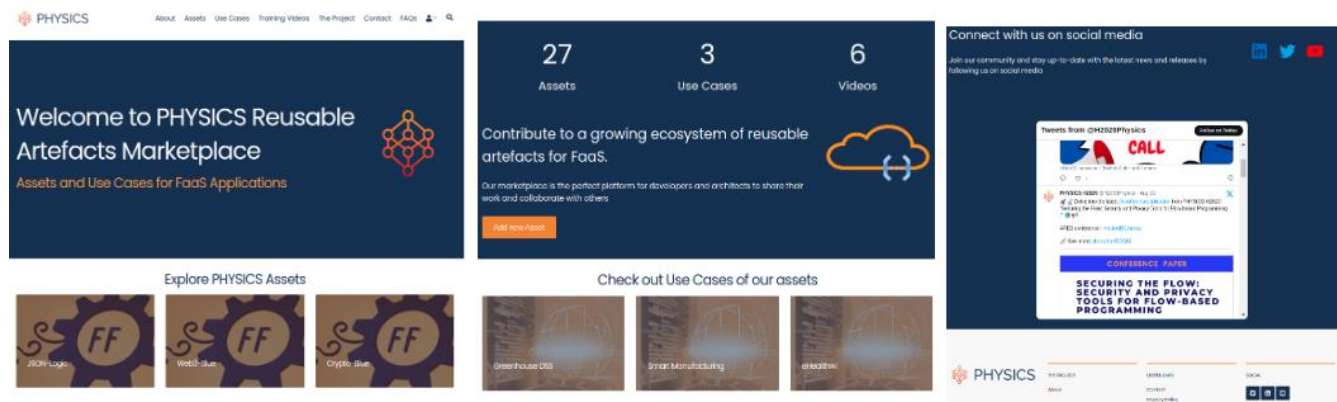


Figure 55 - Homepage

The **about** page serves as a gateway to understanding the core principles and objectives of the project. Here, visitors can inform themselves with the vision, mission, and goals that steer RAMP, grasping a well-rounded perspective on the project's underpinnings.

Moving on, the **Assets** page, as shown in Figure 56, presents a varied array of solutions, tools, artefacts, and services categorized based on attributes such as pattern, service, and semantics, among others, to foster easy navigation. A click on a specific asset takes the user to a space detailing a holistic view of that asset complete with comprehensive descriptions and guidelines for use. For users keen on contributing (requesting) to (from) RAMP, the Add New Asset (Request Asset) button, opens up a form designed for uploading (requesting) new assets; a process delineated in Figure 58. For visitors who are yet to register, this section navigates to the registration/login form, encouraging participation.

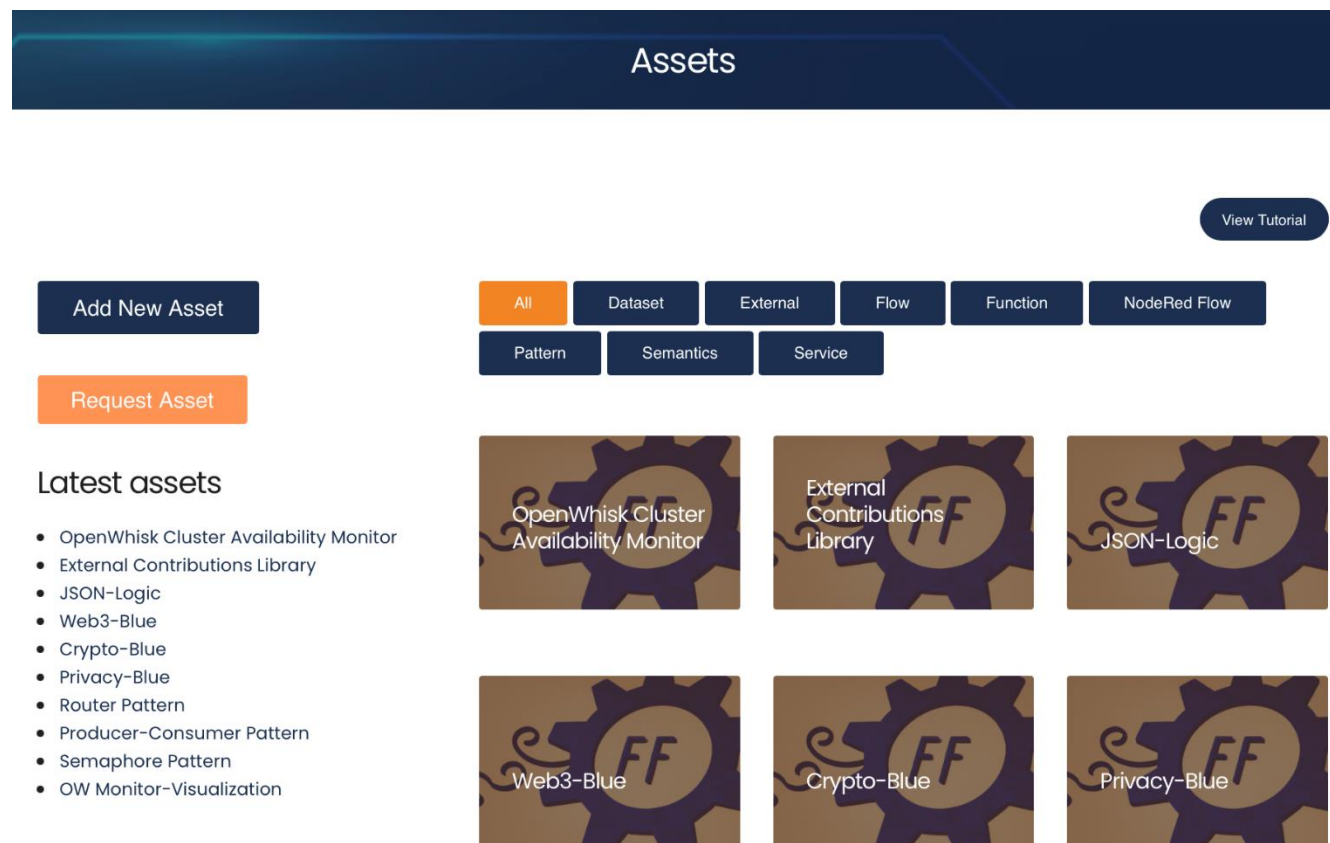
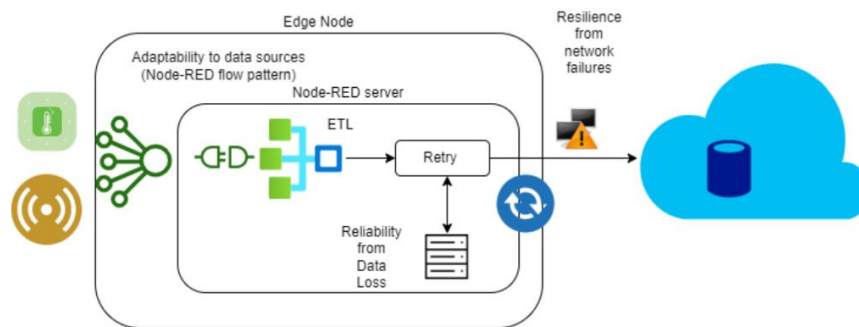


Figure 56 - Assets page

## Edge ETL Data Collector Service Pattern

This subflow handles the pushing of data to an HTTP out endpoint. It accepts JSON input (object or array) in the msg.payload, which is considered the data to be pushed. The flow is intended to be run as a service at the edge, serving as an intermediate data collector. The main goal is to ensure a robust and reliable process for data collection, that is not affected by network connectivity loss or other disturbing factors at the edge.

The main logic of the flow appears in the following figure.



The Node-RED input layer can be used to adapt to diverse protocols needed at the Edge IoT side. An example of an HTTP In local interface that then uses the Edge ETL subflow appears in the following figure.

**Owner:** Harokopio University of Athens

**PoC:** Prof. George Kousiouris

**email:** gkousiou@hua.gr

**Release Date:** 09/07/2022

**License:** Apache License v2.0

**Field of Use:** Application design



Data Collection, Edge, ETL, reliability, retry pattern

Figure 57 - An asset in RAMP

Figure 58 - Form to add/request Asset in/from RAMP

Presently available, the **Use Cases** page offers users a glimpse into the real-world applications and potentials of the PHYSICS project, illustrated through various case studies and scenarios, effectively demonstrating the applicative spectrum of the project outcomes. In Figure 59 depicts the post of the eHealth use case.

## eHealth

The eHealth use case infers on the patient outcomes based on their data. Sensing and data vector preparation is done by Healthentia – an eClinical platform that is used for collecting data in decentralized clinical trials. The inference on patient outcomes is a new service of the platform, part of its transformation towards a Digital Therapeutics (DTx) solution.

The traditional cloud-based implementation of the online inference in Healthentia consists of an inference service that exposes a single API endpoint expecting the input vectors to infer upon and the name of the model to be used. The use case consists of an alternative approach using a PHYSICS function, deployed from a Node-RED flow. The function expects the exact same parameters as the API endpoint. At the heart of the function is a Python script that performs the inference.

### Utilised RAMP Artefacts

OW Skeleton Interface for Node-RED flows as functions

Request Aggregator

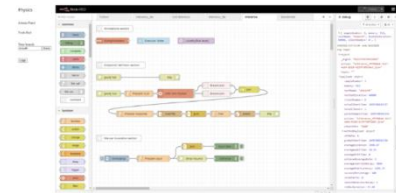
Branch Join

Split Join Pattern

**Owner:** Innovation Sprint

**PoC:** Dr. Aristodemos Pnevmatikakis

**License:** Apache License v2.0



Inference flow for the eHealth use case, as created and tested using the PHYSICS Design Environment.

### Benefits

The transformation to a FaaS architecture enables more dynamic and real-time applications, enabling arbitrary estimations to be acquired on demand by the customers. In the traditional approach, data are collected every 4 hours and inference is performed on the overall batch. This creates the limitation that an interested party needs to wait until the respective results of the entire batch are ready. However, in the new model, predictions can be launched and managed on demand with much lower needed resources compared to a traditional threadpool server, while exploiting the core features of a FaaS platform, such as queue based load leveling management and warm executions. This may lead to a more responsive and interactive application towards the end users.

Furthermore, the solution benefits from:

- Abstraction from the specifics of creating, compiling, deploying and executing a function
- Two resource optimizations over the traditional approach:
- Large arrays of input vectors are processed for inference by different instances of the function after being automatically split.
- Small arrays of input data are automatically concatenated together for a single function invocation.

Figure 59 - eHealth Use Case Post

The **Training** page is a space dedicated to hosting a range of educational materials including videos, webinars, and workshops that are in harmony with the project's directives, offering users resources to enhance their knowledge and skills.

Lastly, the **FAQs** page stands as a quick help center, addressing common questions and providing clarifications on the platform's functionalities, thereby aiding users in navigating the platform effortlessly and making the most of what RAMP has to offer

## 5. PHYSICS SOLUTION FRAMEWORK INTEGRATION ENVIRONMENT

This chapter summarizes the final 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

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<sup>50</sup> orchestrator provided by OKD<sup>51</sup> and deployed inside the AWS Cloud provider<sup>52</sup>. 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 utilization and clean environments to ensure build reproducibility.

The development strategy has been implemented using the DevOps<sup>53</sup> methodology through the tools (shown in Figure 60) and hosted in a specific namespace named “devops” inside the OKD cluster.



Figure 60 - DevOps Tools

The DevOps tools are:

- Gitlab<sup>54</sup> is a Git repository manager that lets developer teams collaborate on PHYSICS’s source code.
- Jenkins<sup>55</sup> is the de-facto standard open-source automation server for orchestrating CI (Continuous Integration)/CD (Continuous Delivery) workflows.
- Harbor<sup>56</sup> is a popular CNCF compliant Docker registry.

<sup>50</sup> <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>

<sup>51</sup> <https://www.okd.io/#what-is-okd>

<sup>52</sup> [https://aws.amazon.com/what-is-aws/?nc1=h\\_ls](https://aws.amazon.com/what-is-aws/?nc1=h_ls)

<sup>53</sup> <https://www.gartner.com/smarterwithgartner/the-science-of-devops-decoded#:~:text=Gartner%20defines%20DevOps%20as%20a,between%20operations%20and%20development%20teams.>

<sup>54</sup> Gitlab (<https://about.gitlab.com/solutions/agile-delivery/>)

<sup>55</sup> Jenkins (<https://www.jenkins.io/doc/>)

<sup>56</sup> Harbor (<https://goharbor.io/docs/2.3.0/install-config/>)

- OpenLDAP<sup>57</sup> is used as the single user directory for all tools, centralising authentication and simplifying management of developer accounts.
- Helm<sup>58</sup> is a package manager that streamlines installing and managing Kubernetes applications.

The interaction between these tools and the workflow that shows how they can be used by a partner is shown in Figure 61. 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 is pushed 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<sup>59</sup> 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.

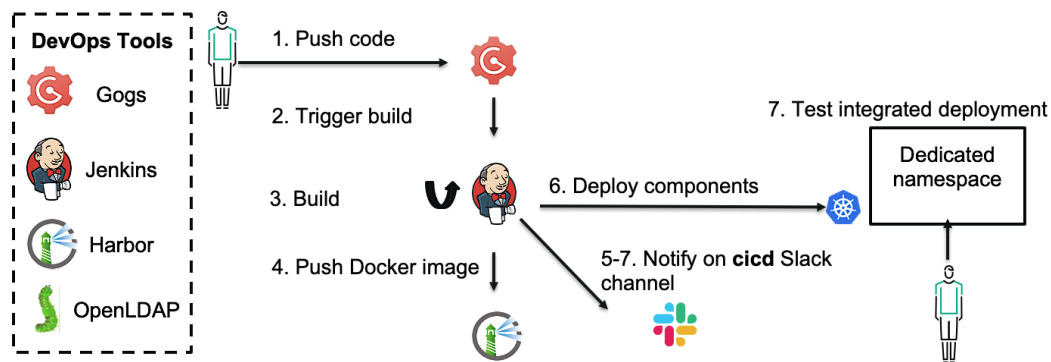


Figure 61 - CI/CD flow

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 62.

<sup>57</sup> OpenLDAP (<https://www.openldap.org/doc/admin25/>)

<sup>58</sup> Helm (<https://helm.sh/docs/intro/>)

<sup>59</sup> Slack (<https://slack.com/intl/en-pt/features>)



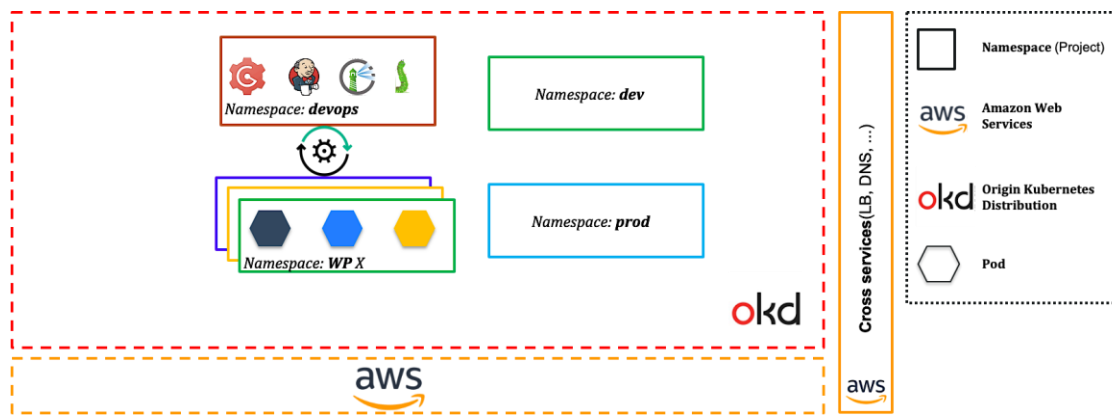


Figure 62 - Integration namespaces

The interaction of developers and partners with the integration environment can take place with two methodologies or through the OKD GUI (Figure 63) or through the use of the `oc`<sup>60</sup> client (Figure 64).

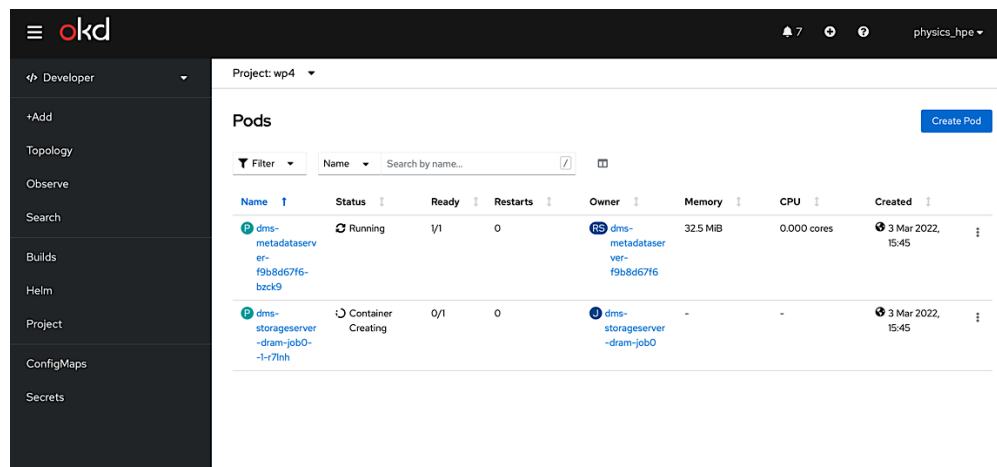


Figure 63 - OKD GUI

```
[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 64 - OC client

The same segregation implemented into integration environment has been reported within the Gogs structure.

Specifically, 5 “Organizations” have been defined in Gogs as presented in Figure 65.

<sup>60</sup> [https://docs.openshift.com/container-platform/4.7/cli\\_reference/openshift\\_cli/getting-started-cli.html](https://docs.openshift.com/container-platform/4.7/cli_reference/openshift_cli/getting-started-cli.html)

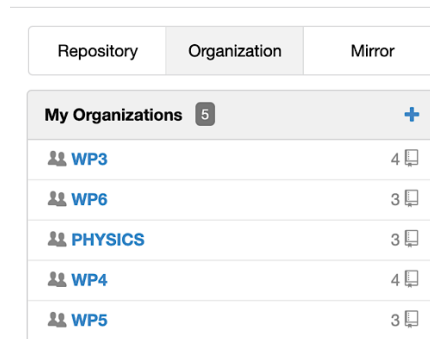


Figure 65 - Organizations inside Gogs

In this way, the developer accesses his organization, 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 66)

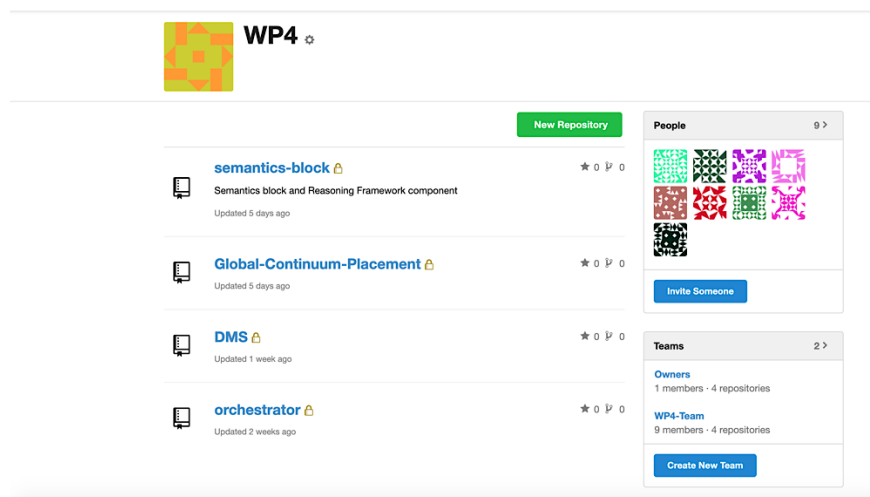


Figure 66 - Repositories inside one organization

### 5.1.2 Deployment strategy

The PHYSICS platform is designed to be able to operate from on-premises to the cloud and up to the edge. The variety of the environments in which it can be deployed as well as the diversity of the components that constitute it, requires the use of deployment methodologies that are simple, general purpose and minimize the possibility of errors.

With this in mind, it was considered to use IaC (Infrastructure as Code) tools like Terraform<sup>61</sup>. Such tools are preferred, as they can easily recreate “on demand” the blueprint environment. We selected OpenTofu<sup>62</sup> is a fork of Terraform that is open-source and 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.

Figure 67 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 OpenTofu 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

<sup>61</sup> Terraform (<https://www.terraform.io/intro/index.html>)

<sup>62</sup> OpenTofu (<https://opentofu.org/docs/intro/>)



used to install and configure the components into the environments created by OpenTofu. The only prerequisites that the customer deploying PHYSICS needs are the OpenTofu and HELM clients alongside the resources needed by PHYSICS to run.

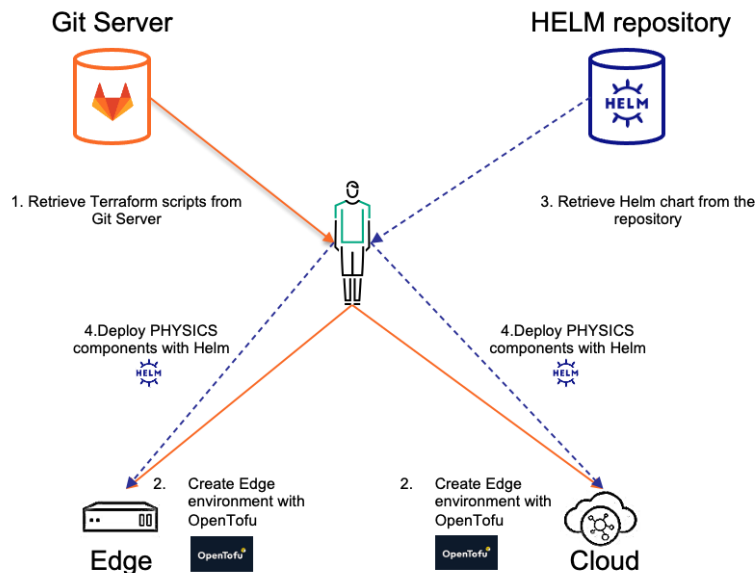


Figure 67 - Deployment flow

## 5.2 Cross infrastructure components

The partner HPE in the second phase of PHYSICS project, worked to provide some components that are cross within the PHYSICS platform and could be used to facilitate the integration of the various services.

In particular, two main issues often required within the PHYSICS architecture were analyzed:

1. Data historicization
2. Concurrent request handling

To address the first request, it was agreed to provide two methodologies:

- a. One based on No-SQL DB that would allow unstructured data to be stored.
- b. One that would provide S3-compliant storage.

As a No-SQL DB it was decided to use a single instance of MongoDB<sup>63</sup> deployed in the *dev* namespace. The deployment as well as the reachability methodology used for that component are the same as those chosen for the components successively described in this section, and are respectively Helm for deployment, while the ClusterIP<sup>64</sup> service was adopted for exposure. For use by the services, a dedicated user was set up for each Workpackage that could work on its own collection. Regarding S3-compliant storage, MinIO<sup>65</sup> software was chosen. In this case to simplify management by all users the GUI was also exposed externally via endpoints:

<https://minio.apps.ocphub.physics-faas.eu>

A specific bucket, with associated access policies, was created per specific component.

<sup>63</sup> <https://www.mongodb.com/atlas/database>

<sup>64</sup> <https://kubernetes.io/docs/concepts/services-networking/service/>

<sup>65</sup> <https://min.io/product/kubernetes>

For handling multiple requests to a service, a broker message based on publisher/consumer logic in particular the Advanced Message Queuing Protocol (AMQP)<sup>66</sup> was chosen, so the RabbitMQ<sup>67</sup> has been selected. As in the case of MinIO, it was preferred to expose the GUI externally via the endpoint <https://rabbitmq.apps.ocphub.physics-faas.eu>, so that we could simplify both the creation of exchanges and queues

### 5.3 Visual Workflow component

The partner 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, 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:

- A first one for managing the transformation of functions in Docker
- A 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 was based as described in 5.2 paragraph on RabbitMQ server as a backend and integrating its use inside Jenkins through the MQ Notifier<sup>68</sup> plugin.

The last component configured in this architecture is MongoDB. It has been used to store 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.

### 5.4 Visual Workflow on cloud

In the first part of the project HPE worked closely with the Work Package 3 team for the development of the Visual Workflow (aka Design Environment).

In the initial version the Visual Workflow was designed to be composed of two macro blocks:

1. A block running on the edge (front-end) i.e., the customers' PCs/environments, from where to develop the workflows of the FaaS functions
2. A cloud block with backend components

The edge was based on the use of 3 microservices (containers)

1. A microservice for managing the UI

---

<sup>66</sup> <https://www.rabbitmq.com/tutorials/amqp-concepts.html>

<sup>67</sup> <https://www.rabbitmq.com/documentation.html>

<sup>68</sup> <https://plugins.jenkins.io/mq-notifier/>

2. A microservice as backend based on SFG
3. A microservice based on NodeRED

Their execution relied on the docker compose<sup>69</sup> methodology, which was easier to install and use than using Kubernetes as an orchestrator.

However, during the project, this approach revealed several problems which can be summarized as follows:

1. Difficulty pulling images from the PHYSICS registry from clients behind a reverse proxy or firewall.
2. Problems with correct cloning/pushing towards GIT server in a Windows Subsystem for Linux (WSL) environment.
3. Basic knowledge of docker/docker-compose required for correct use.

To overcome these problems, it was decided to move the edge part to the cloud, maintaining the macro-logic between the front-end and the backend unchanged. This approach allows the user to take advantage of the same features he previously had locally, simply by opening any browser without having to install anything. This modification involved the development and modification of several components.

The first was to transform the first two edge microservices, the UI and the SFG, in a standalone application, with a full reengineering, where the SFG is evolved in a compliance GraphQL application and the UI have a redesigned UI and integrate user and environment provisioning. Both the application was released as standalone application in CLOUD, accessible on the route:

<https://control-ui.apps.ocphub.physics-faas.eu/>

For Node-RED and the application that needs a direct access to it, the approach was different because, for the nature of Node-RED, each user needs a separate environment. To achieve this requirement, it will be creating a POD for each registered user.

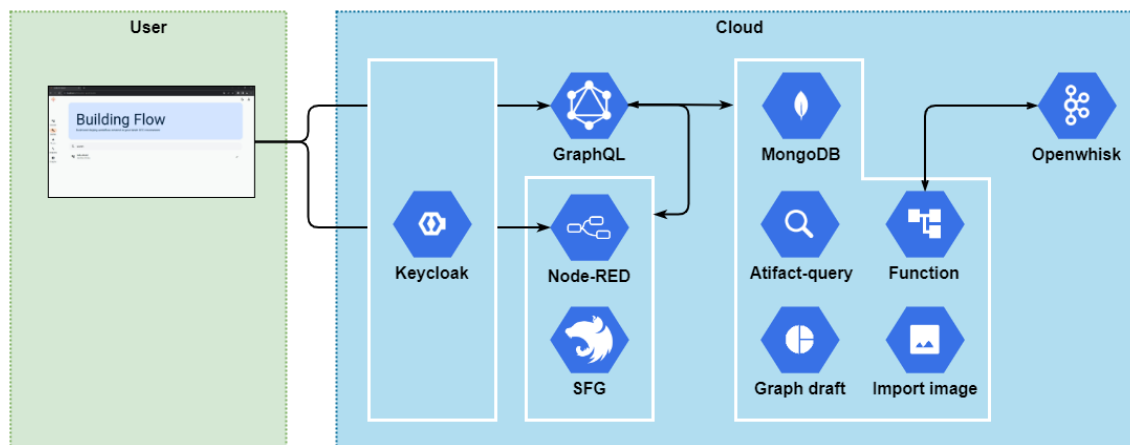


Figure 68 - Visual Workflow CLOUD

The detail of the Visual Workflow CLOUD architecture structure is shown in **Error! Reference source not found.**, the SFG is evolved in GraphQL that interacts with the user Node-RED POD and the already present microservices and a POD for each user with Node-RED and a light way of the actual SFG.

<sup>69</sup> <https://docs.docker.com/compose/>

Each user POD is identified during deployment on OKD by the name *de-pod-<username>* to which a service named *de-<username>* is associated and by a route identified with the nomenclature *de-route-<username>*

The reachability of the POD from the outside is guaranteed by a specific route from the structure:

*<https://de-<username>.apps.ocphub.physics-faas.eu>*

Having created this system, the aim was to optimize the use of cloud resources, giving the user the possibility of running the environment only when necessary and stopping it when not required. For this purpose, the Visual Workflow UI integrates all the needed feature for provisioning user and POD. For this purpose, the UI as SPA has been divided into three main blocks:

- **User Creation:** accessible when in login phase is providing a general user; in this section the user can register to the UI. If the process is complete successfully the user is redirected to the Environment Creation section.
- **Environment Provisioning:** is the home section of the UI, each user access is redirected first on this page. This section is responsible for checking if the user POD is already created and if it is started. The creation of POD is made only on the first user access; vice versa the start of the POD is needed to make at each user login, because for a CLOUD resource optimization the POD will be shut down at user logout. If everything is up and running, the user is redirected to the Design Environment section.
- **Design Environment:** the last section, where all the actual UI features are integrated, such as build, test and import image.

The setup of the customer's infrastructure was delegated to Jenkins through the design of a specific pipeline *DEPLOY-DE-CLOUD-USER* (Figure 69) in which the following steps are carried out:

- Creation of a specific named de-cloud namespace if it does not exist
- Creation of secrets in this namespace for historicizing user credentials
- Creation of the configmap for the deployment configuration
- Preparation of PVCs for the stateful part of the deployment
- Creation of a specific branch within the GIT server, used to version the workflow code
- Creating the deployment
- Creation of associated services
- Creation of routes for external reachability

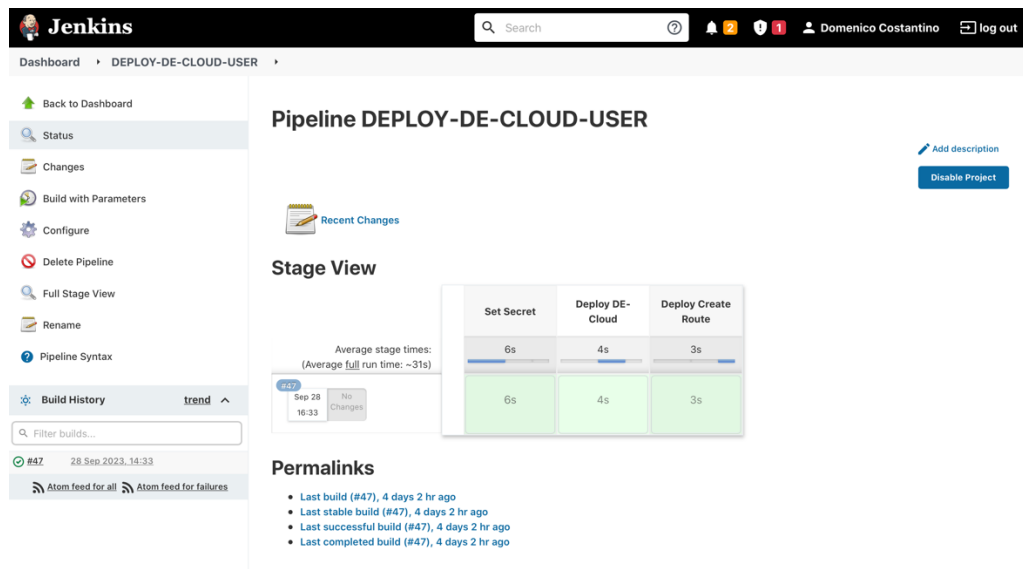


Figure 69 - Jenkins Pipeline for Visual Workflow

The security aspects have been also considered in the redesign: in fact, the whole environment was accessible only with user credentials, not only to access the UI page exposed via HTTPS, but also for making Node-RED safe. To create a homogeneous integration between UI and Node-RED, HPE integrated Node-RED with Keycloak <sup>70</sup>. We used it alongside UI, GraphQL and NodeRED to implement OIDC authorization Flow.

The configuration used to secure NodeRED is presented in the code below:

```
adminAuth: {
  type: "strategy",
  strategy: {
    name: "keycloak",
    label: 'Sign in',
    icon: "fa-lock",
    strategy: require("passport-keycloak-oauth2-oidc").Strategy,
    options: {
      clientID: "node-red",
      realm: 'master',
      publicClient: "false",
      clientSecret: "*****",
      sslRequired: "external",
      authServerURL: "https://auth.apps.ocphub.physics-faas.eu/auth",
      callbackURL: "https://node-{{USERNAME}}.apps.ocphub.physics-faas.eu/auth/strategy/callback",
      verify: function(accessToken, refreshToken, extraParams, profile, done) {
        return done(null, profile);
      }
    }
  },
},
users: [
  { username: "{{USERNAME}}", permissions: ["*"]}
]
```

<sup>70</sup> <https://www.keycloak.org/>

```
    ]  
},
```

## 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 second and final phase of the project. The document is the accompanying textual specification of the major result of the deliverable and the task: the second 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 7th major WP6 milestone (MS12 – PHYSICS 2nd integrated platform release – foreseen for M34 of the project), and provide the second and final 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 has been 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 upcoming 2nd iteration of the PHYSICS Pilots and Use Cases Operations and Stakeholders’ Evaluation of the proposed solution framework. The delivered integrated PHYSICS solution framework and RAMP marketplace are fundamental inputs and drivers for WP7 (Exploitation, Dissemination and Impact Creation), with special emphasis on T7.2 (Business Innovation Development & Exploitation).

The WP6 work done on Tasks 6.1 nevertheless puts the basis for possible future (exploitation) evolutions and enhancements of the proposed solution framework and marketplace with additional capabilities and features, even after the project official end. Such enhancements could take into account the latest evolutions of relevant technologies occurring after the project timeframe, and also the expected feedback coming from the final project wave of the Pilot Operations and Stakeholders Evaluation tasks.

## 7. REFERENCES

- [1] F. D. & B. J., "Graph Databases Comparison: AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, and OrientDB," *In Data*, pp. 373-380, 2018 July.
- [2] S. University, "Pocket," [Online]. Available: <https://github.com/stanford-mast/pocket>.
- [3] 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.
- [4] PHYSICS Consortium, "D2.5 - PHYSICS REFERENCE ARCHITECTURE SPECIFICATION V2," 2022.



## DISCLAIMER

---

The sole responsibility for the content of this publication lies with the authors. It does not necessarily reflect the opinion of the European Union. Neither the EASME nor the European Commission is responsible for any use that may be made of the information contained therein.

---

## COPYRIGHT MESSAGE

---

This report, if not confidential, is licensed under a Creative Commons Attribution 4.0 International License (CC BY 4.0); a copy is available here: <https://creativecommons.org/licenses/by/4.0/>. You are free to share (copy and redistribute the material in any medium or format) and adapt (remix, transform, and build upon the material for any purpose, even commercially) under the following terms: (i) attribution (you must give appropriate credit, provide a link to the license, and indicate if changes were made; you may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use); (ii) no additional restrictions (you may not apply legal terms or technological measures that legally restrict others from doing anything the license permits).

---