

PHYSICS

OPTIMIZED HYBRID SPACE-TIME SERVICE CONTINUUM IN FAAS

D2.5 – PHYSICS REFERENCE ARCHITECTURE SPECIFICATION V2

Lead Beneficiary	UPM
Work Package Ref.	WP2 – Requirements, Architecture and Technical Coordination
Task Ref.	T2.3 – Reference Architecture Specification
Deliverable Title	D2.5 – PHYSICS REFERENCE ARCHITECTURE SPECIFICATION V2
Due Date	2022-11-30
Delivered Date	2022-12-31
Revision Number	3.0
Dissemination Level	Public (PU)
Type	Report (R)
Document Status	Final
Review Status	Internally Reviewed and Quality Assurance Reviewed
Document Acceptance	WP Leader Accepted and Coordinator Accepted
EC Project Officer	Mr. Stefano Foglietta

H2020 ICT 40 2020 Research and Innovation Action



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

CONTRIBUTING PARTNERS

Partner Acronym	Role ¹	Name Surname ²
UPM	Task leader	Marta Patiño, Ainhoa Azqueta, Luis Mengual, Tonghong Li, Tomas San Feliu, José Calvo Manzano
HUA	Contributor	George Kousiouris, Stylianos Tsarsitalidis, Evangelos Boutas, Teta Stamati, Chris Giannakos
RHT	Contributor	Josh Salomon, Luis Tomas
HPE	Contributor	Alessandro Mamelli, Domenico Costantino, Roberto Musso
RYAX	Contributor	Yiannis Georgiou
ATOS	Contributor	Antonio Castillo Carlos Sánchez
BYTE	Contributor	Yannis Poulakis
INNOV	Contributor	George Fatouros
GFT	Contributor	Lisa Erba
RHT	Reviewer	Luis Tomas Bolivar
HUA	Reviewer	Anastasia Dimitra
DFKI	Quality Assurance	Arnold Herget

REVISION HISTORY

Version	Date	Partner(s)	Description
0.1	2022-10-16	UPM	ToC Version, Initial contents
0.2	2022-10-23	INNOV	Updated content for the Reasoning Framework
0.3	2022-10-31	RHT	Updated sections 2.5, 2.13 and 3.3
0.4	2022-11-14	ATOS	Update section 2.10 and 3.2
0.5	2022-11-15	HPE/GFT	Update section 2.2
0.6	2022-11-28	UPM	Ready to review
1.0	2022-12-06	RHT	Reviewed version
1.1	2022-12-09	HUA	Reviewed version
1.2	2022-12-21	UPM	Integrated version
2.0	2022-12-27	DFKI	Quality Assurance version
3.0	2022-12-31	GFT	Version for the submission

¹ Lead Beneficiary, Contributor, Internal Reviewer, Quality Assurance

² Can be left void

LIST OF ABBREVIATIONS

FaaS	Function as a Service
DAG	Directed Acyclic Graph
DMS	Distributed Memory Service
DoA	Descriptions of Action
ETL	Extract, Transform and Load
UI	User Interface
DevOps	Development Operations
QoS	Quality of Service
QoE	Quality of Experience
OCM	Open Cluster Management
OW	OpenWhisk
OWL	Ontology Web Language
RA	Reference Architecture
MVP	Minimum Viable Platform
CI/CD	Continuous Integration / Continuous Delivery
ML	Machine Learning
IaC	Infrastructure as Code
SFG	Serverless Function Generator
QoS	Quality of Service
WP	Work package
PEF	Performance Evaluation Framework
JSON-LD	Javascript Object Notation – Linked Data
RDF	Resource Description Framework
REST	REpresentational State Transfer
SFG	Serverless Function Generator
UML	Universal Modelling Language

EXECUTIVE SUMMARY

The goal of this deliverable is the description of the PHYSICS project architecture. This is the second version of the PHYSICS framework architecture foreseen in the project. The PHYSICS architecture is developed as part of the work package WP2, Requirements, Architecture and Technical Coordination.

The architecture of PHYSICS consists of a set of software components that are developed in three technical work packages (W3 Functional and Semantic Continuum Services Framework, WP4 Cloud Platform Services for a Global Space-Time Continuum Interplay and WP5 Extended Infrastructure Services with Adaptable Algorithms) which correspond to the three foreseen layers of the PHYSICS platform: application level, platform level and infrastructure level. The PHYSICS architecture has been defined based on the study of state of the art and the requirements definition and updated after the first integrated version was produced and tested by the pilots. The architecture is described using a functional view in which the description of each software component is provided, as well as the interactions among them. This functional description presents for each component its definition, challenges the component has to deal with, input received and produced output. Hence, it provides the structuring principles that will drive the integration of the PHYSICS components in a unified platform. As such the PHYSICS architecture will drive integration activities towards producing the PHYSICS platform and integrating the use cases.

TABLE OF CONTENTS

1.	Introduction	7
1.1	Objectives of the Deliverable	8
1.2	Insights from other Tasks and Deliverables	8
1.3	Deliverable Structure	9
2.	PHYSICS Architecture	10
2.1	Architecture Overview	10
2.2	Visual Workflow/Design Environment	11
2.3	Semantic Extractor and Application Semantic Models	16
2.4	Design Patterns Repository	18
2.5	Elasticity Controllers	20
2.6	Reasoning Framework	21
2.7	Performance Evaluation Framework	23
2.8	Global Continuum Placement	25
2.9	Distributed Memory Service	28
2.10	Adaptive Platform Deployment, Operation & Orchestration	29
2.11	Service Semantic Models	33
2.12	Local Adaptive Scheduler	35
2.13	Resource Management Controllers	37
2.14	Co-allocation Strategies	38
3.	PHYSICS Components Interactions	41
3.1	Application Development Environment (WP3)	41
3.2	Continuum Deployment Layer (WP4)	42
3.3	Infrastructure Layer (WP5)	45
4.	PHYSICS Global View	47
5.	PHYSICS Development and Deployment Strategies	49
5.1	Development Strategy	49
5.2	Deployment Strategy	50
6.	Conclusions	53

TABLE OF FIGURES

Figure 1 PHYSICS design environment and toolkits	7
Figure 2 PHYSICS planning.....	8
Figure 3 PHYSICS deliverables dependencies.....	9
Figure 4 PHYSICS Software Components.....	10
Figure 5 Design Environment GUI	11
Figure 6 Create Application Developer Use Case.....	12
Figure 7 Test Application Developer Use Case	12
Figure 8 Deploy Application Developer Use Case.....	13
Figure 9 Design Environment Components and Interactions with other elements of the PHYSICS platform	13
Figure 10 Design Environment Node-RED implementation	14
Figure 11 Design Environment test deployed functions	15
Figure 12 Application Semantic Models components	17
Figure 13 General incorporation of a pattern concept in PHYSICS.....	19
Figure 14 Reasoning Framework architecture and interactions with other components.....	22
Figure 15 Performance Evaluation Framework Diagram and Interactions.....	24
Figure 16 Global Continuum Placement high-level view and relation to other components.....	26
Figure 17 Global Continuum Placement component internal architecture.....	27
Figure 18 Distributed Memory System Architecture.....	29
Figure 19 Translator service, part of Adaptive Platform, Deployment, Operation & Orchestration component	30
Figure 20 FaaS Proxy service, part of Adaptive Platform, Deployment, Operation & Orchestration component.....	30
Figure 21 Deployment pipeline of a PHYSICS application workflow	31
Figure 22 Global Runtime Adaptation flow.....	32
Figure 23 The two control planes	33
Figure 24 Service Semantic Models Architecture and semantics interactions.....	34
Figure 25 Local Adaptive Scheduling Algorithms and its relation to the Global Continuum Placement for the 2-level scheduling of the continuum.....	35
Figure 26 Co-allocation Strategies component internal architecture	39
Figure 27 WP3 internal and external interactions	42
Figure 28 PHYSICS Deployment Pipeline	43
Figure 29 WP5 internal and external interactions for Function Registration	45
Figure 30 WP5 internal interactions for Function Execution	46
Figure 31 Design, deployment, and execution of a function.....	47
Figure 32 CI/CD workflow example.....	50
Figure 33 PHYSICS components deployment flow	52

1. INTRODUCTION

The PHYSICS project aims at delivering a complete vertical solution that will offer (a) advanced cloud application design environments for Application Developers to create workflows of their applications, exploiting generalized Cloud design patterns for functionality enhancement with existing application components, easily designed and reused through intuitive visual flow programming tools (Cloud Design Environment); b) Platform-level functionalities to be easily incorporated by providers in order to translate the created application workflows into deployable functional sequences, based on the *Function as a Service* (FaaS) model, optimizing their placement across the Cloud computing domain and exploiting the computational space-time continuum as well as advanced semantics for the definition of a global service graph (Optimized Platform Level FaaS Services Toolkit); c) Provider-local resource management mechanisms that will enable providers to offer competitive and optimized services with extended interfaces offering local fine grained control of elasticity rules and policies, while applying a holistic set of provider-local strategies based on a wide set of controlling techniques and tackling key aspects of multitenancy (Backend Optimization Toolkit). The main features of each of these three toolkits are summarized in Figure 1.

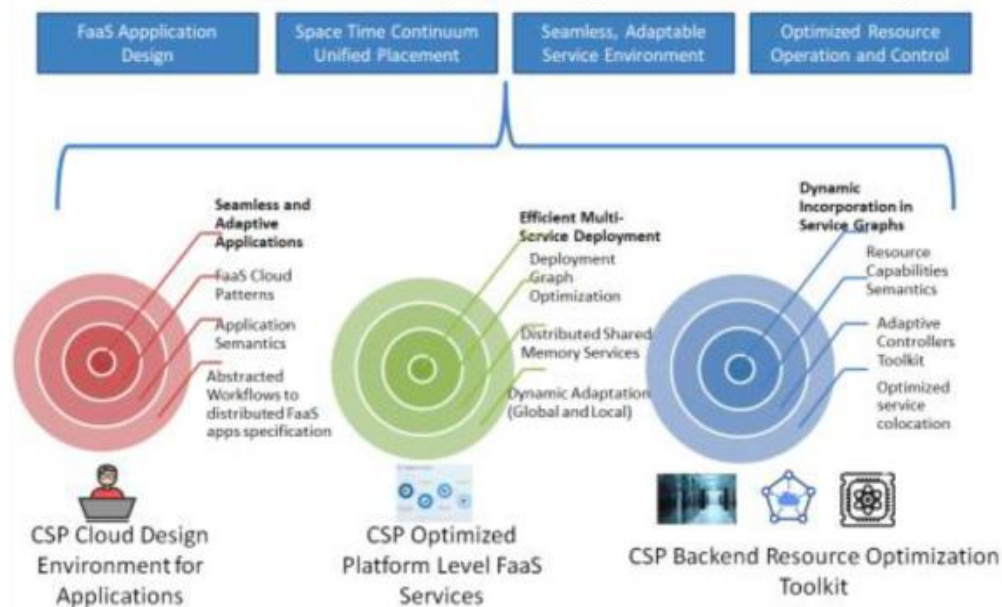


Figure 1 PHYSICS design environment and toolkits

In order to achieve these goals the PHYSICS project is structured in seven work packages, W1 Project Management and Administration, WP2 Requirements, Architecture and Technical Coordination, WP3 Functional and Semantic Continuum Services Design Framework, WP4 Cloud Platform Services for a Global Space-Time Continuum Interplay, WP5 Extended Infrastructure Services with Adaptable Algorithms, WP6 Use Cases Adaptation, Experimentation, Evaluation, and WP7 Exploitation, Dissemination and Impact Creation. Work packages WP3, WP4 and WP5 (technical work packages) are in charge of developing each of these toolkits/environments. WP2 main roles are the studying state of the art in each of the fields where PHYSICS is contributing to, gathering the requirements, and designing the PHYSICS architecture. This deliverable presents the second and last version of the PHYSICS architecture.

1.1 Objectives of the Deliverable

The goal of this deliverable is to define the final version of the architecture of the PHYSICS project. The architecture is defined as a set of views, namely functional view, information view and deployment view³. This deliverable mainly describes the functional view of the PHYSICS architecture, where the software components are identified as well as their interactions. For each component, a description of the main goals of the component is provided, as well as their main inputs and outputs and issues the component must deal with.

This document is relevant for the design of the technical components produced in work packages WP3 (Functional and Semantic Continuum Service Design Framework), WP4 (Cloud Platform Services for a Global Space-Time Continuum Interplay), and WP5 (Extended Infrastructure Services with Adaptable Algorithms) as well as for the design of pilots in (Use Cases Adaptation, Experimentation and Evaluation). The deliverable is also useful for future adopters of the PHYSICS platform either as a whole or the different toolkits to be developed during the lifetime of the project.

This deliverable presents the second version of the PHYSICS Architecture being part of Phase 3 of the project. Although this is the final version of the PHYSICS architecture, the deliverable is a living document that may be updated as the project progresses. This version of the PHYSICS architecture reflects the feedback received from the use cases after the end of the first iteration of the project in month 18 as shown in Figure 2.

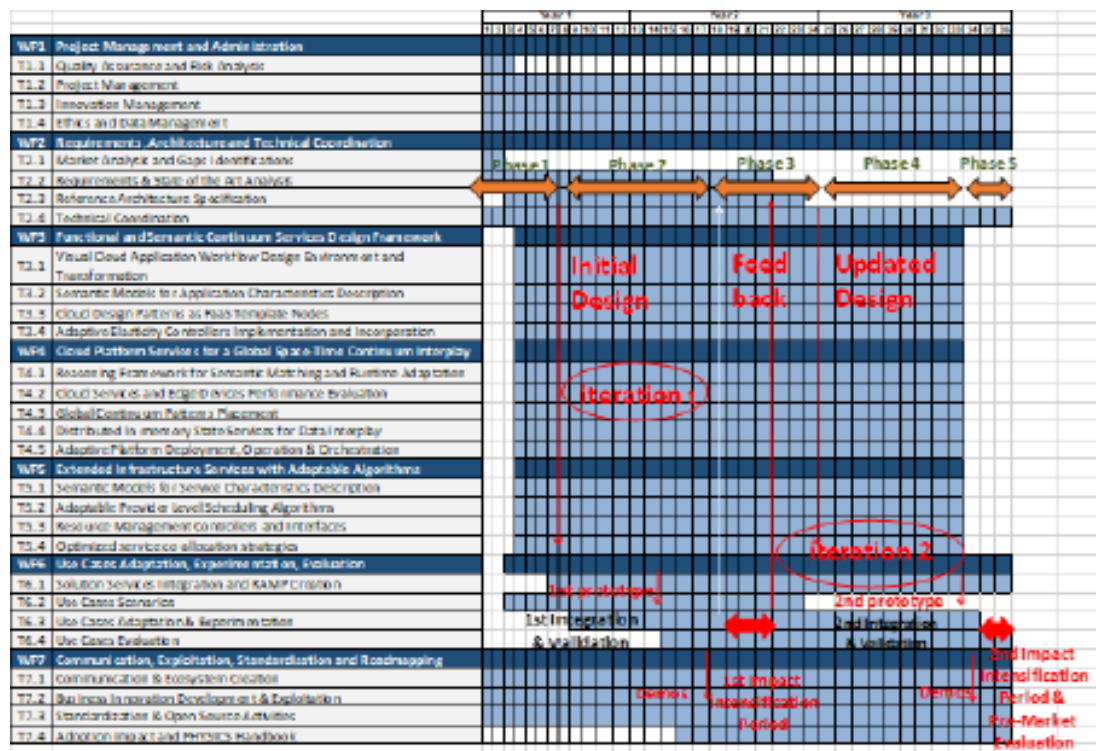


Figure 2 PHYSICS planning

1.2 Insights from other Tasks and Deliverables

The first version of the architecture of PHYSICS was designed using as input the study of state-of-the-art analysis and the requirements gathered in deliverable D2.3. State of the Art Analysis and Requirements

³ Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives. N. Rozanski, E. Woods. Addison-Wesley 2012

Definition v2. This version of the architecture also takes into consideration the input from the pilots defined in deliverable D6.4 Application Scenarios Definition v2. Although deliverable D6.4 and this deliverable are very close in time, they progressed in a coordinated manner. Several meetings were organized to define the scope and needs of PHYSICS pilots from the PHYSICS platform and define the requirements according to the pilots. These meetings provided very valuable information for the definition of the PHYSICS Architecture. Figure 3 shows the dependencies between this deliverable (D2.5) and other deliverables in the project. The timeline is represented at the top in months (M2 represents month 2) and the different phases of the project are shown at the bottom of the figure (Requirements, Development, Evaluation...). This deliverable (shown in a red circle) will provide input for the deliverables in charge of documenting the design of the toolkits to be developed in work packages WP3, WP4 and WP5, respectively, and the associated software prototypes, namely deliverables D3.2, D4.2 and D5.2. The second version of the integration of these prototypes will be documented in deliverables D6.2 and D6.6, while the final evaluation of the PHYSICS toolkits will be documented in deliverable D6.8, concluding the fourth phase of the project.

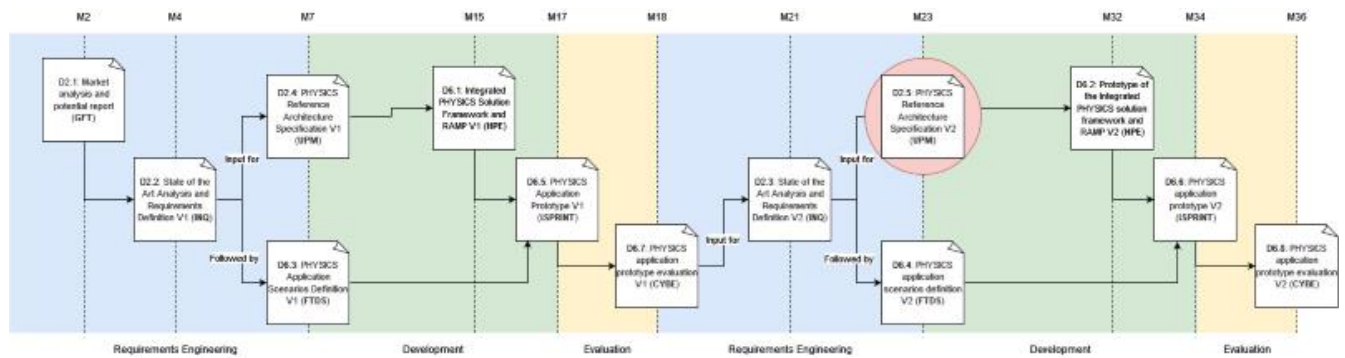


Figure 3 PHYSICS deliverables dependencies

1.3 Deliverable Structure

The rest of the deliverable is organized as follows. First, an overview of the different software components of PHYSICS architecture is presented in Section 2. Then, the functional view of the PHYSICS architecture is presented in the next section, Section 3. This view is organized in several subsections that correspond to each of the components to be developed. The interactions among the components of one toolkit are described in Section 4. Section 5 presents the global view of the PHYSICS platform. Conclusions are presented in the last section of the document, Section 6.

2. PHYSICS ARCHITECTURE

2.1 Architecture Overview

PHYSICS consists of three main layers with the goal of enabling seamless application creation, deployment and operation across distributed and dynamically managed service environments and infrastructures. These layers are depicted in Figure 4 from top to bottom and can be summarized as follows,

- A top-level *application developer layer* (*design environment*), that will enable abstracted design, reusability of code as well as implemented programming patterns in the FaaS model. Existing components will be wrapped around FaaS operators.
- A *continuum deployment* mid-level layer for the support, deployment and federated execution layer, including services and functionalities that enables component semantics, services benchmarking and evaluation, deployment optimization and definition, spanning across different and diverse providers and services and enabling a seamless execution across.
- A bottom level *infrastructure layer*, targeting at optimizing the provider-local strategies and resource management, for the benefit of both the local provider as well as the hosted application instances.

Each layer is developed in one of the respective technical workpackages (WP3, WP4 and WP5). The boxes in the figure represent components while the arrows represent dependencies among components. Most of the components are associated with a single task in the work plan. The task associated with a component is also depicted in Figure 4.

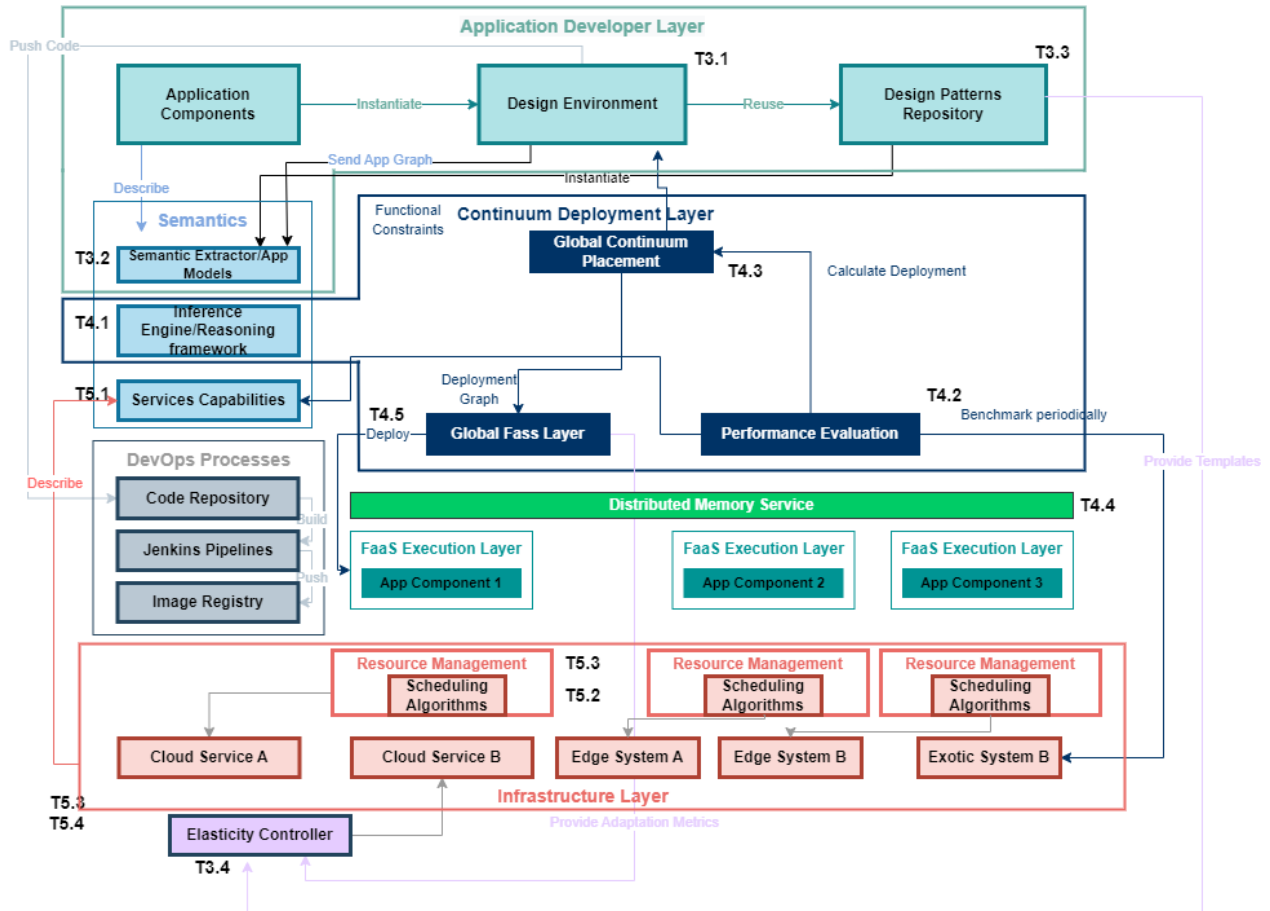


Figure 4 PHYSICS Software Components

2.2 Visual Workflow/Design Environment

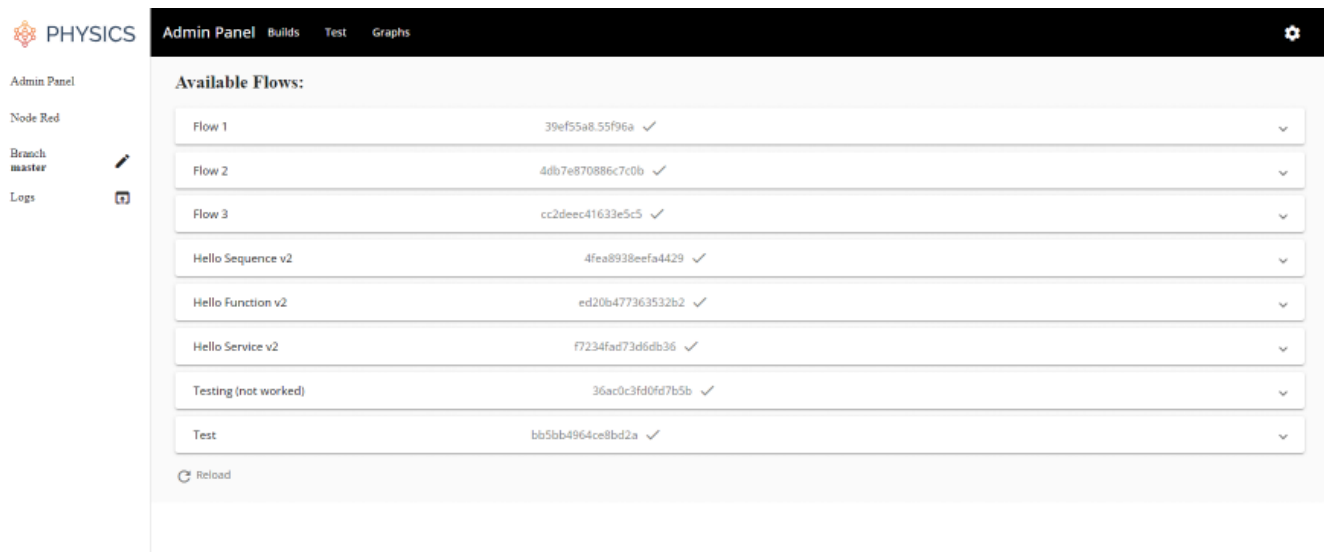


Figure 5 Design Environment GUI

Component Description

The PHYSICS Design environment (Figure 5) is the main entry point for the application developer when interacting with the PHYSICS platform. In this environment, the latter needs to visually design and implement their application, by creating new code segments, importing existing ones, or re-using generic, available implementations (in the form of patterns) available from the PHYSICS platform. A key element is the ability to dictate workflows of operations among these diverse components, that in the end will be implemented during runtime, so that different elements of the application can be deployed according to their envisioned operation (i.e., as microservices or as functions, or a combination of the two). The overall UML use case diagram appears in Figure 6, Figure 7 and Figure 8.

Except for the overall application creation testing, the developer will also need to test the individual elements of the flow, either locally (for small function segments) or as a whole (local flows). When the local integrated flow test is complete, they will also need to do a deployment test in order to ensure that the implementation is correctly transferred to the platform side. Once the tests are complete, the final application will need to be deployed in the production environment.

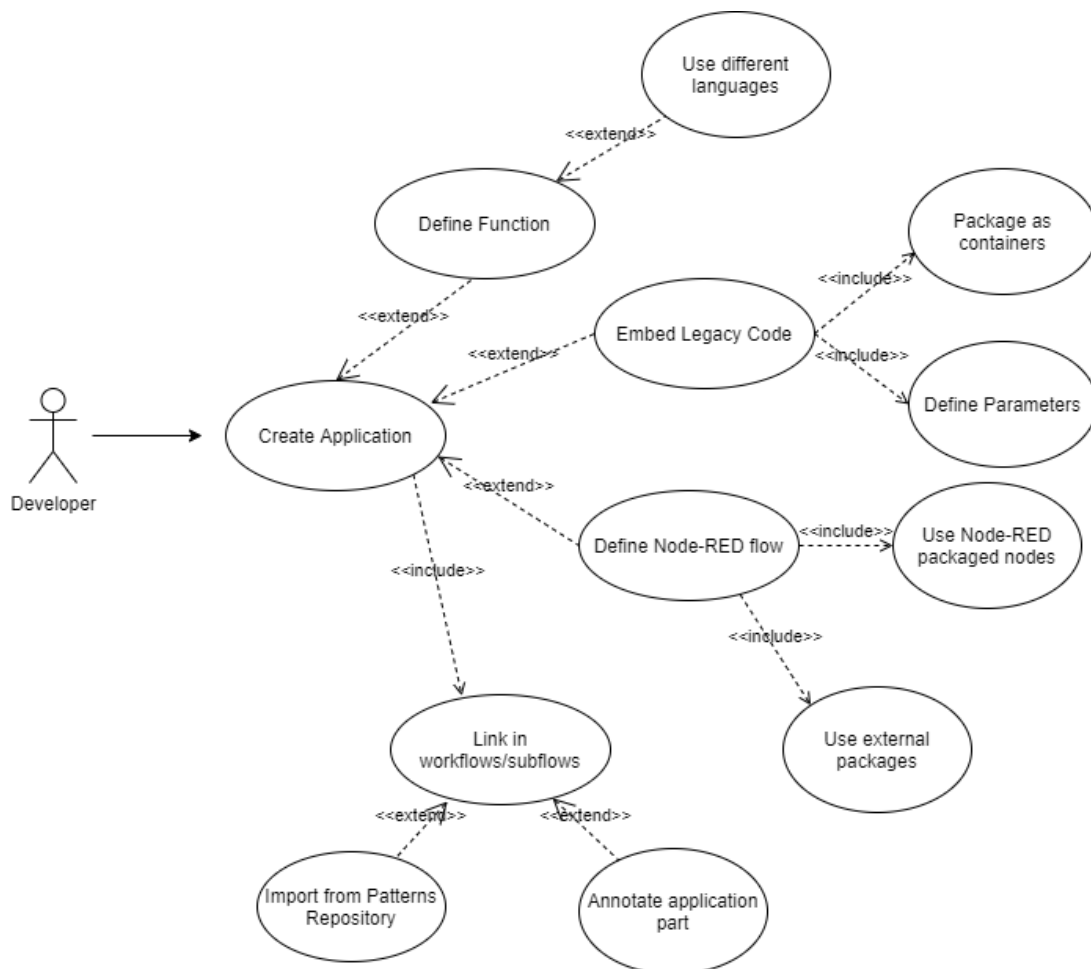


Figure 6 Create Application Developer Use Case

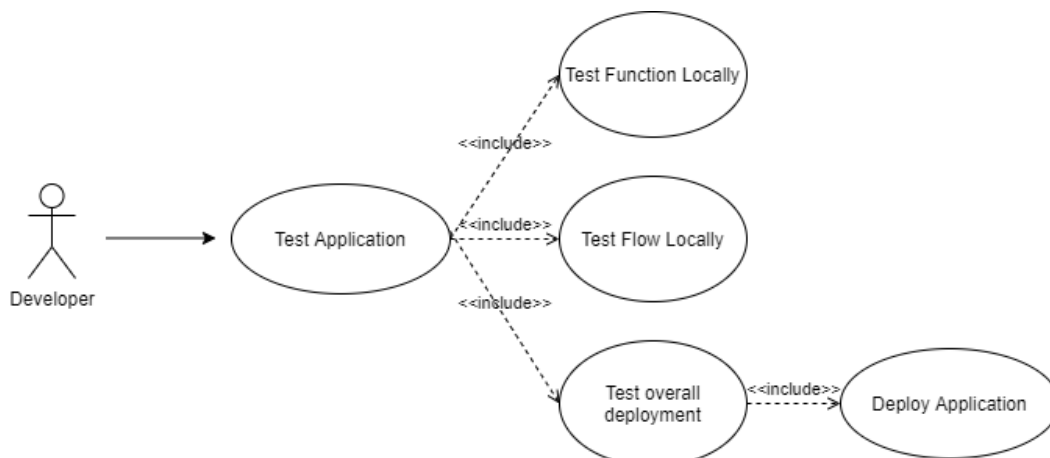


Figure 7 Test Application Developer Use Case

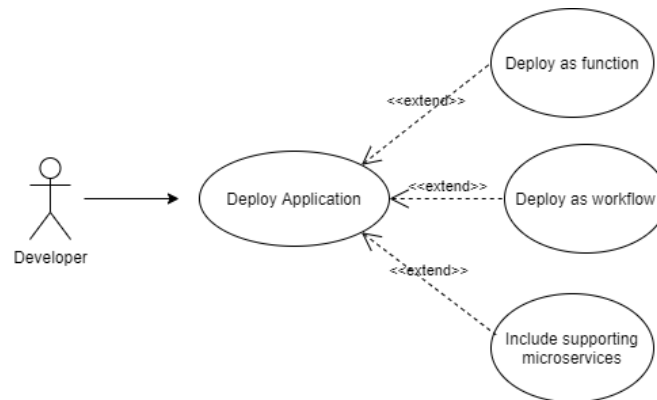


Figure 8 Deploy Application Developer Use Case

The overall architectural diagram of the WP3 entry point for the developer appears in Figure 9.

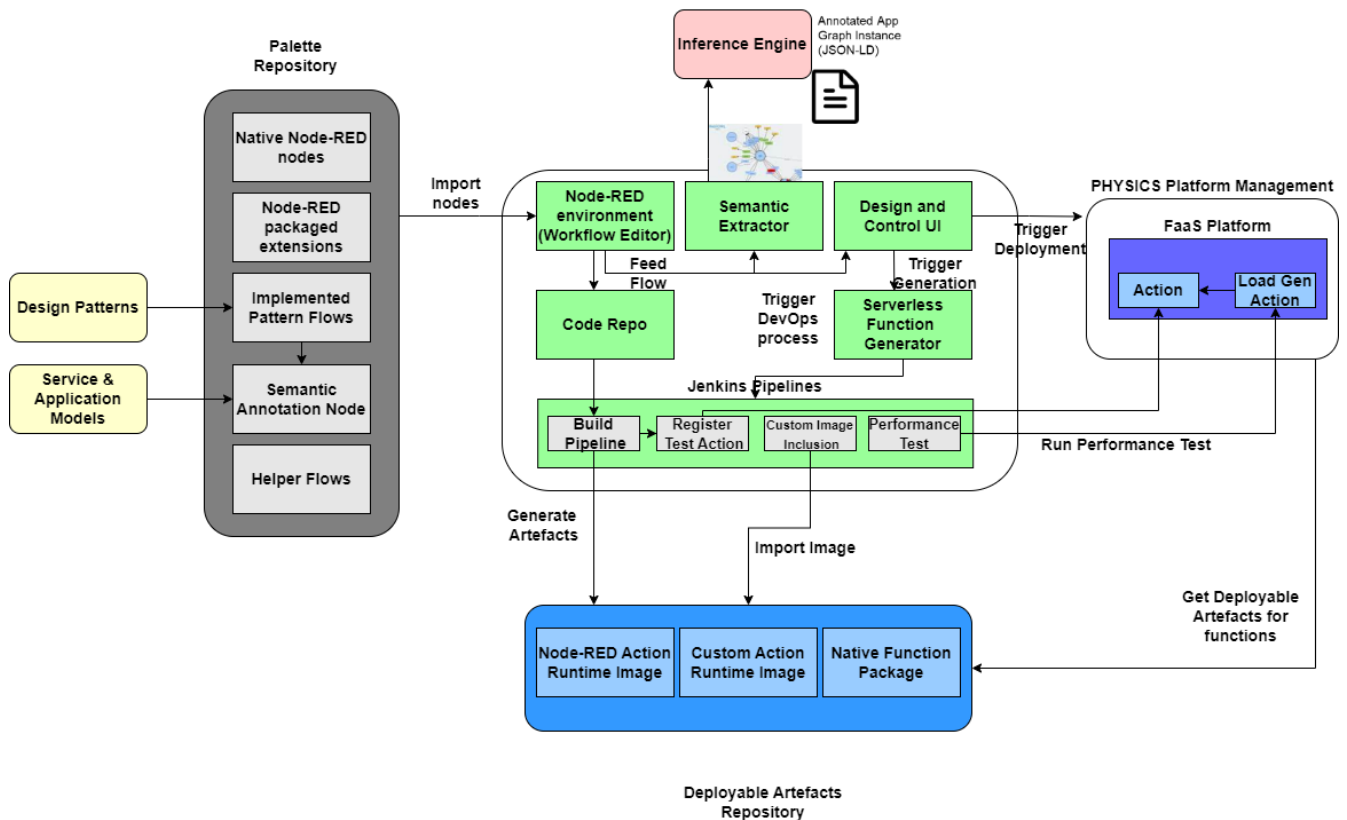


Figure 9 Design Environment Components and Interactions with other elements of the PHYSICS platform

This also includes interactions with the semantic block (Application Semantic Models (Section 2.3), Inference Engine (section 2.6), the Design Patterns Repository (Section 2.4) and the FaaS Platform Deployment (Section 2.10).

The Design Environment is a centralized UI application that includes and embeds other elements/tabs offering the aforementioned functionalities with a centralised login to guarantee the security and an isolated work environment for each user. The central element is the Node-RED environment (Figure 10), used to develop the application structure. In this, the developer can exploit a palette of existing nodes, that either offer functionality or a link/interface with an external system (e.g., interaction nodes for creating, registering and invoking functions on a FaaS platform). Collections of nodes linked in order to implement a specific functionality can be performed in the form of subflows and reused in different locations of the code or application. The relevant flows can be uploaded on the Node-RED repository as subflows, or they can also be packaged as regular Node-RED nodes.

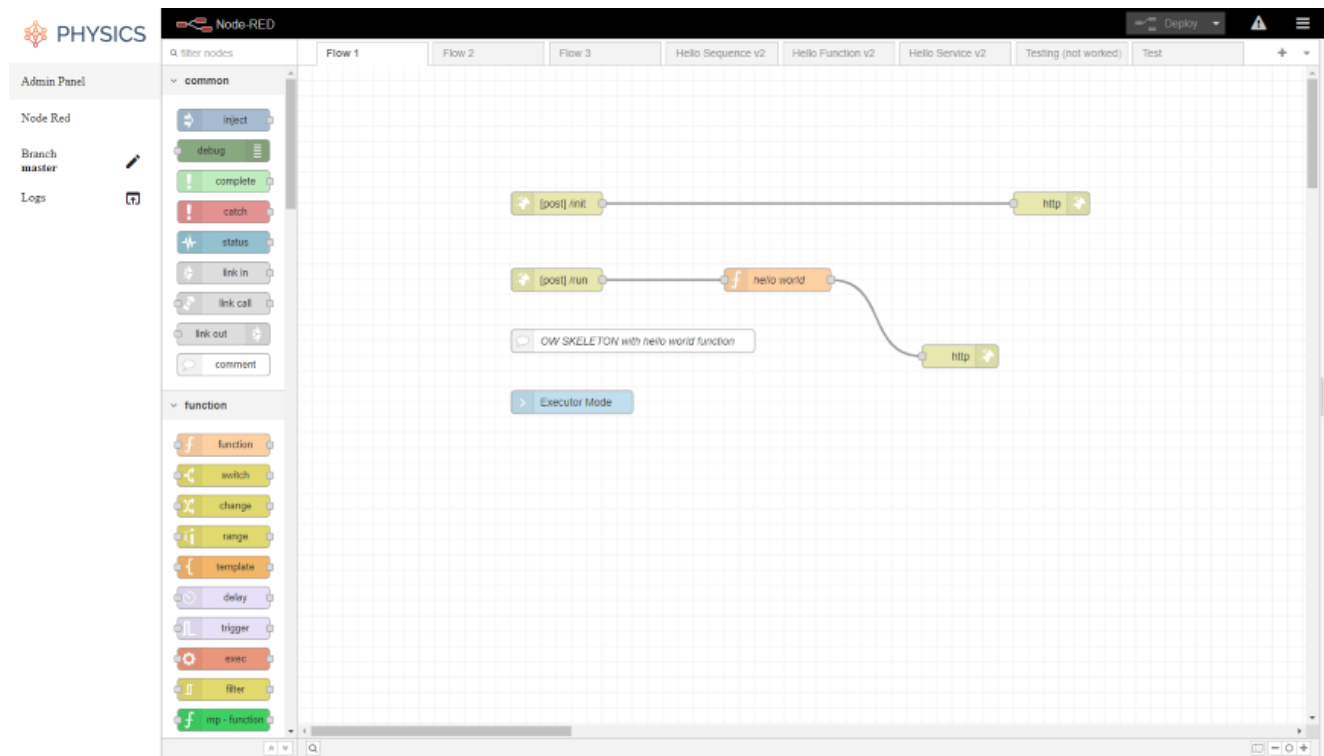


Figure 10 Design Environment Node-RED implementation

The environment foresees the need to aid in the support of three distinct execution modes. These include a) native functions created in the environment, b) legacy components imported in the application graph as well as c) arbitrary flows that are created within Node-RED, reutilizing its vast node repository, in order to offer functionalities, integration or application-level workflow orchestration abilities. Once the developer has created the relevant functions and workflows, testing of these operations and/or triggering of the corresponding DevOps processes that are needed to build the respective deployable artefacts are supported by the corresponding UI tabs and trigger a relevant build pipeline. During this process, the created flows are retrieved, and the overall necessary steps coordinated. As an example, functions developed within the environment need to be adapted/migrated to the FaaS platform runtime, through means of extracting their code and dependencies and injecting them into one of the available image templates. This process includes also the availability of baseline processes and skeleton flows needed to interact with the FaaS platform. For example, the latter in the case of OpenWhisk assumes that any registered function artefact exposes two endpoints (an /init method and a /run method) used to initialize and then execute the function logic. This in turn triggers a second pipeline, that undertakes the registration of the created action in the test environment. Following that, a manual test can be performed against the action from the relevant UI tab (Figure 11) or a third pipeline can be triggered for automated performance data collection by using the load generation functions stemming from WP4.

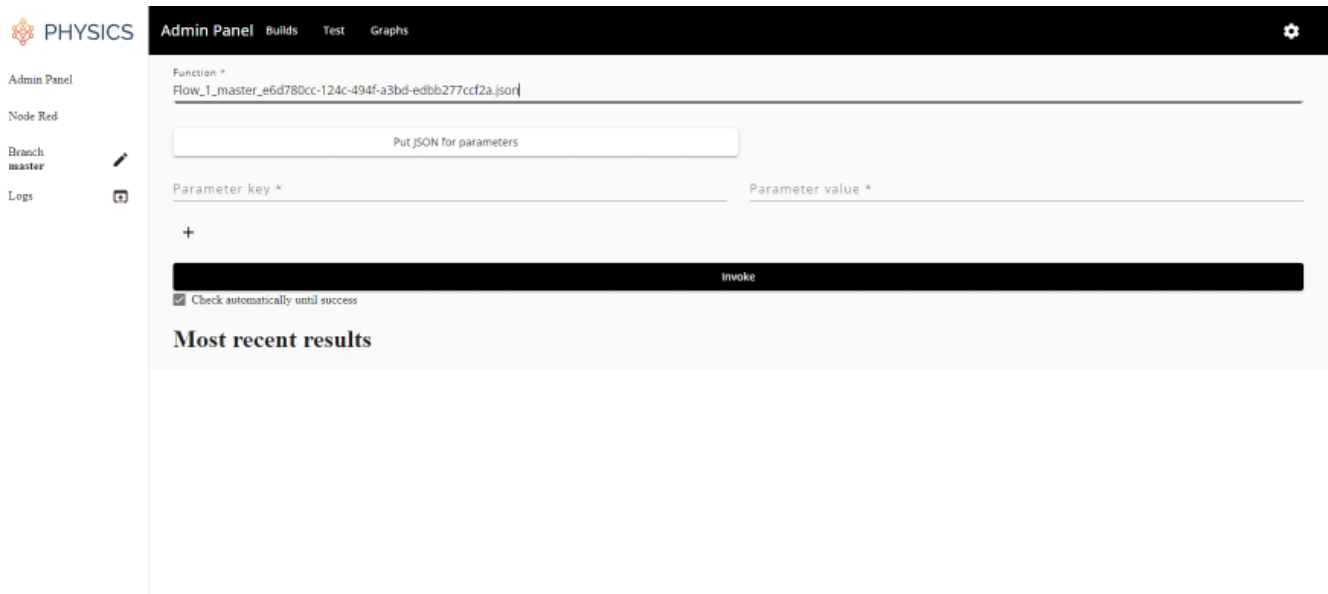


Figure 11 Design Environment test deployed functions

For the external actions' incorporation, specific tabs are needed through which a developer may declare external dockerized actions. For this reason, another pipeline is made available for importing custom images that the developers need to make available for action registration.

Further functionalities can be included in a flow, such as the need for functional annotations at the function level (e.g., inclusion of external library dependencies, sizing considerations etc.), ability to use semantic annotator nodes enriching the semantic descriptions of a flow. Upon finalization, the respective flows that consist of the application graph are passed through the Semantic Extractor subcomponent that transforms them into instance triples and stores them in the Reasoning Framework (inference engine) of WP4. The outcome of the process in the Design Environment is to have created and registered different application blocks on the FaaS platform. In the end, the overall application graph is forwarded towards WP4, enriched with a number of features such as annotations used further down the PHYSICS process for functional adaptation, preferable means of management, non-functional requirements etc.

Main issues to be handled by the component

- Ability to incorporate multiple diverse elements in the application graph (legacy code, function code, microservices, subflows etc.) and bundle them in a workflow style
- Inclusion of code dependencies in the function nodes
- Synchronization between development versions of the code and deployable artefacts, that involves direct DevOps processes in the context of the Design Environment, as well as registration of the according artefacts in the FaaS and Orchestration platform

Inputs

- **Expected Node-RED node inputs**
 - Readymade patterns from Pattern Repository
 - FaaS platform nodes to handle interactions and platform operators
 - Semantic annotation nodes
 - Any imported Node-RED node or flow from existing repositories (e.g. <https://flows.nodered.org/>)
- **Code segments of different types**

Existing microservices
 Legacy code to be embedded in function logic
 Arbitrary flows created in Node-RED to be executed alongside the application

Outputs

- Application design graph representation in JSON, annotated with information to be used by latter stages.
- Registered functions and sequences on the FaaS platform.
- Deployable artefacts (e.g. container runtimes) of the inputs transformed into functions (especially for legacy code and Node-RED flows).

2.3 Semantic Extractor and Application Semantic Models

Component Description

The main goal of the Semantic Extractor component is to express the characteristics of an application graph, as well as the constraints and requirements of application components, and describe them in a format that is widely understandable and can be utilized by other components. A metamodel (the PHYSICS Application Ontology) provides the types of entities and their relationships, i.e., workflows, functions, resource requirements, and locality constraints. The workflows defined in the Design Environment component (Section 2.2), are then fed to the Semantic Extractor and converted to individuals that belong to the classes of the ontology. A workflow is expressed as a dependency graph of functions (nodes), with each function node or collection of nodes having characteristics, like resource requirements and locality constraints, imprinted on them as attributes. The resulting Application Semantic Models are mainly used in the reasoning processes implemented in the Inference Engine component, with the workflows being a key type of entity.

The OWL-based ontology describes the overall domain of the application workflows seen in PHYSICS. The workflow nodes/steps themselves are either functions, in FaaS terms, or middleware dependencies for other functions. “Workflow pattern”, as well as related terms, such as “workload type” and generic requirements for them, like the need for a specific kind of device, are to be included in the ontology. Each pattern is essentially an example or template workflow that is targeted for some specific type of application, with some predefined requirements and characteristics like maximum distance/cost between function nodes. Workflows and Workflow Patterns are RDF individuals that adhere to the terms of the OWL ontology. All the information related to application workflows are imprinted in the ontology, so that it can be used by the Inference Engine in conjunction with the Service Semantic Models (Section 2.11). The most fundamental operation to be done, driven by the Semantic Extractor included in Section 2.2, using application models that adhere to the PHYSICS ontology, is to enrich the application graph with attributes related to the requirements and constraints, by means of reasoning using this ontology along with the ontology of the Service Semantic Models, and then match the application graph with the best resources, based on the imprinted attributes, by means of subgraph matching. The Inference Engine then implements more sophisticated operations based on this one.

The generation of semantic models via the application ontology/metamodel is based on the RDF.js libraries suite, in order to semantically enrich the workflow models exported by the visual workflow/design environment. The enrichment is essentially the transformation of the workflow models into a form that adheres to the PHYSICS application ontology. The transformation is composed of custom logic, with the traversal of inputs being done using the JSONata library. The data format used is JSON-LD, since it is an RDF representation in JSON, and each JSON-LD document/instance can directly reference the OWL ontologies that act as the domain of structures and attributes used in it and can be thought of as an advanced schema. In essence, the OWL ontology acts as a metamodel, and the JSON representation of the application is

transformed into JSON-LD that contains individuals of classes expressed in the ontology. The applications semantic models are linked with the Services Semantic Models (Section 2.11) through the resource constraints and requirements that workflow nodes have as attributes. The relationship of this component with other components is presented in Figure 12.

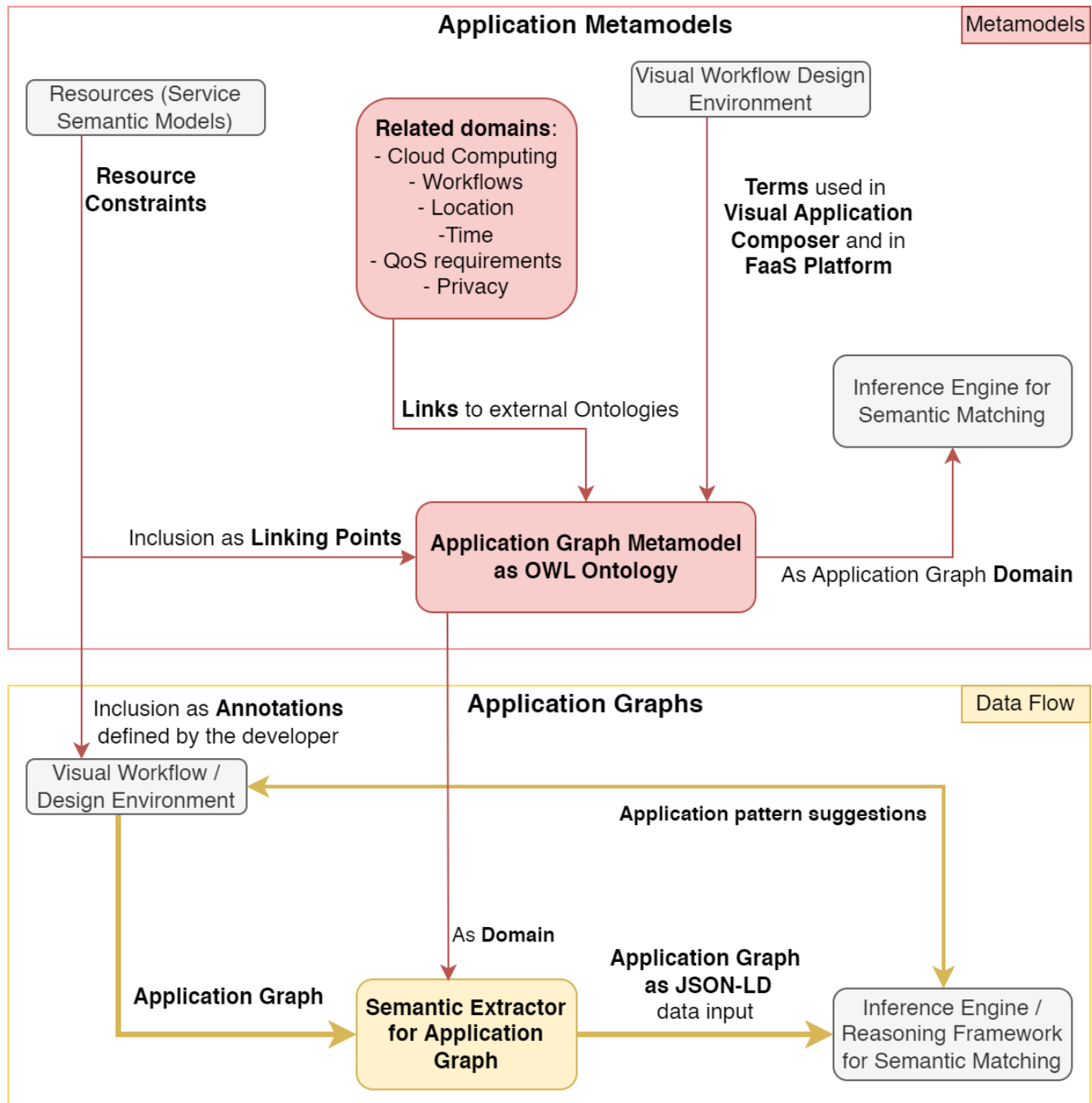


Figure 12 Application Semantic Models components

In the above schema, the composition and usage of the ontology is highlighted in red arrows and boxes, the semantic transformation / extraction of application graphs is presented in the yellow arrows and box, and the grey boxes represent components from other tasks. The operation of this component also helps prepare the suggestions of application patterns that are created by the reasoning framework and shown to the developer via the visual workflow environment.

Main issues handled by the component

- Annotating mechanisms at the function level
Through specialized annotation nodes in node-RED or code level annotations in the visual design environment
- Expression of the application workflows of PHYSICS as a linked data domain
The Application Graph Metamodel is an inference-capable OWL ontology
Creation of a domain that contains links to external ontologies and related tasks
- Preparation of application workflows for the processes of the reasoning engine
Transformation/Enrichment into JSON-LD, using the application workflows domain

Inputs

- Application description from the design environment
- Resource requirements
- Graph representation that is eventually forwarded to the resource optimizer

Outputs

- Transformed application model/graph that is inference-capable. It is based on the metamodel (a static ontology) that describes application models
- Application and function requirements and constraints are included as attributes in the application graph



2.4 Design Patterns Repository

Component Description

The main goal of the Design Patterns Repository is to provide the ability to developers that will use the PHYSICS platform to use common, already implemented, and compatible with FaaS paradigm, design, and algorithmic patterns in their applications. These patterns can be either functional, which means they provide an actual functionality (for example, a Request Aggregator or a Node-RED-flow-as-function executor), or they can be design patterns that help with application development and provide support artefacts.

These patterns are primarily implemented as Node-RED flows, a way which provides the flexibility to maximize their optimization and re-usability in the FaaS platform or in more general contexts through the Node-RED repository. However, any other language or framework can be used, as long as a relevant interface node is provided at the Node-RED level. Alternative packaging may include the use of a Docker image, so that the relevant component can be deployed alongside the FaaS application and used by it.

The implemented patterns have and will be published in two manners:

- As a subflow structure, directly at the Node-RED PHYSICS collection (<https://flows.nodered.org/collection/HXSkA2jILcGA>)
 This means of publication requires a manual installation of dependencies (other node-red nodes needed) however it ensures that the users can afterwards adapt the provided subflows based on their own needs and wishes
- As a packaged node-red node (example at: <https://www.npmjs.com/package/node-red-contrib-owmonitor>)
 This means of publication ensures better packaging, with included dependencies and installation, however it also means that the developers can not change the internal workings of a pattern

The baseline Node-RED image provided by PHYSICS includes the produced patterns as well as their dependencies out of the box in any case. A pattern may also need the existence of other services from the

platform level, such as object storage services, messaging and event management services. In this case, the existence of adequate interface nodes in the Design environment will aid the creation of flows that use such services.

Main outputs of this component are the actual implementation of the identified patterns in an executable manner, documentation of each pattern which will be available to the Visual Design Environment and, finally, UI components which will be necessary to the Visual Environment so it can represent each pattern. These elements are included in the Design Environment in Section 2.2. Where relevant, pattern documentation includes the instantiation of the pattern semantic description, that may incorporate various characteristics such as typical pattern applicability use cases, what functional and non-functional aspects it enhances (e.g. performance, reliability, cost), configuration parameters for the pattern, as well as other linked patterns. In many cases in pattern-based development, patterns can act in a complementary or competitive manner.

Main issues to be handled by the component

- Means of a pattern implementation and incorporation in an application graph
Should follow the specification of the design environment and of the various execution modes.
- Ability to launch patterns with one of the deployable means identified in Section 2.2.
- Pattern parameter description and configuration

In many cases patterns come with a parameter set that needs to be configured by the developer. This may be case specific and may influence the applicability or effectiveness of the pattern. Examples of such parameters, in the case of a Retry Pattern, include the number of retries performed, potential back-off intervals, selection of the option with relation to what happens if the call finally fails etc.

In cases of patterns that involve some form of self-adaptation, through a rule or an AI model, parameters would include the location and version of the model to be used or the configuration of the rule. Examples of such patterns are the Request Aggregator for handling set batch size for release as well as the Split Join pattern for the granularity of the split of the incoming payload.

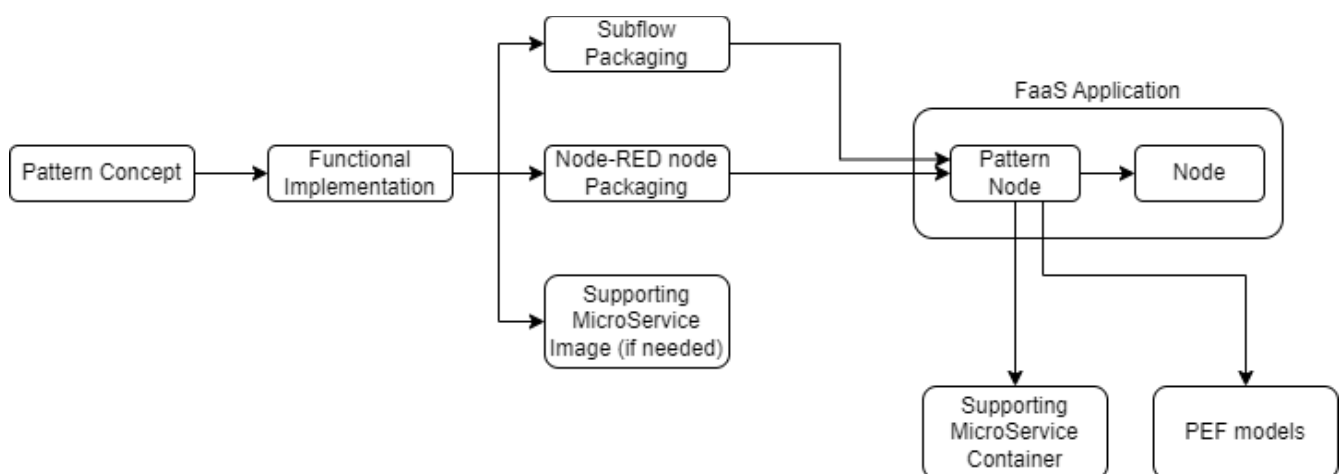


Figure 13 General incorporation of a pattern concept in PHYSICS

Inputs:

- Prototype flows (in Node-RED), function sequences or microservices of pattern implementations.

Outputs

- Documentation of patterns with relation to various aspects such as parameter definition, configuration, runtime adaptation.
- Reusable and deployable pattern artefacts.
- Collaboration with deployed models for pattern regulation/configuration from PEF component

2.5 Elasticity Controllers

Component Description

The main goal of this component is to horizontally and vertically scale a PHYSICS deployment based on various static (e.g., workflows, semantic description) and dynamic (e.g. the load and load prediction, the system performance metrics) inputs in a performant and economical way. The main idea is not only to use basic metrics to make the decisions, but also to include application metrics into consideration (e.g., response time or queue length instead of just CPU or memory). This component decides what is the number of pods per replica set, and how these should change when some events occur, in order to meet the user requirements (e.g., latency and bandwidth) or how much CPU or memory needs to be allocated to them over time (vertical scaling). The recommendations of the controllers are later realised by the Co-allocation Strategies component (Section 2.14) calling the APIs provided by the Resource Management Controllers component (Section 2.13).

The work in this component is based on upstream Kubernetes features for Horizontal⁴ and Vertical⁵ pod autoscalers (HPA and VPA). PHYSICS work focuses on detecting the right application metrics for scaling decisions and making them available through Prometheus, as well as creating its own recommender with focus on FaaS differentiated features (such as warm containers). To achieve this the KEDA project⁶ has been evaluated for the horizontal use case and a specific scaler will be implemented/enhanced to better account for the PHYSICS needs. For the vertical scaling, a customised Vertical Pod Autoscaler will be developed to better account for PHYSICS (FaaS) needs⁷.

Main issues to be handled by the component

- Access to application specific metrics by integration with the metrics monitoring system (i.e., Prometheus)
- Adapt the deployments over time to meet their performance requirements
- Identify the key performance indicators to actuate on them
- Integration of new controllers with HPA and VPA, as well as with KEDA
- Make them easily deployable/usable through proper K8s APIs

Inputs

- Application performance metrics and limits
- Minimal required performance goals
- Type of application and scaling type/engine to use

⁴ <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>

⁵ <https://docs.openshift.com/container-platform/4.11/nodes/pods/nodes-pods-vertical-autoscaler.html>

⁶ <https://keda.sh/>

⁷ <https://cloud.redhat.com/blog/how-to-enable-a-customized-vpa-recommender-on-openshift>

- System workload and performance metrics needed for prediction, including low level metric related to pods

Outputs

- Minimal (or almost minimal) scaled configuration (pods and replica sets) that meets the performance requirements and uses.

2.6 Reasoning Framework

Component Description

The Reasoning Framework (RF) lies between the three layers of PHYSICS (i.e., application, platform, and infrastructure), enabling semantic interoperability between these different contexts. It could be perceived as an interface between the PHYSICS platform's layers, serving as a central repository for application and resource metadata while interacting with various platform components that provide or request data. The RF interprets the latter as graphs and applies semantic inference to create relevant connections between them. In this way, the RF contributes to the automated, timely, and optimised deployment of the input applications.

To achieve that, the RF leverages the ontologies developed by the Application Semantic Models (T3.2) and the Service Semantics component (T5.1) that provide a common language for the various data types of the platform (e.g., application's functions and workflows, developer annotations/requirements, QoS (Quality of Service), performance evaluations, Kubernetes cluster descriptions). Specifically, the input data to the RF have been translated into triples according to the relationships defined in the ontologies. At the same time, the RF provides appropriate endpoints for injecting the individual application and service data. To this end, the RF consists of two components, (i) a server that implements the REST endpoints and (ii) a knowledge base (KB) that facilitates the storage, processing, and reasoning of the input triples.

The Reasoning Framework relies on the open-source AllegroGraph⁸, a horizontally distributed, multi-model (document and graph), entity-event knowledge graph technology that enables the extraction of sophisticated decision insights and predictive analytics from highly complex, distributed data that cannot be answered with conventional databases. AllegroGraph provides an architecture through the REST protocol, while there are APIs for various programming languages, including Python (Graph databases comparison: Allegrograph, Arangodb, Infinitegraph, Neo4j and Orientdb). This facilitates the enhancement of machine learning models, typically served by Python-based applications, with features retrieved from the KB.

Furthermore, RF consists of a Flask-based backend service (Design an MVC model using python for flask framework development, 2019) that is responsible for exposing specific REST endpoints so that other platform components (i) ingest application and resource data, (ii) retrieve required information, and (iii) inferred insights from the AllegroGraph for optimising application design and deployment in terms of cost, latency, performance and more. Specifically, the design environment posts the application graph to the Flask service that forwards it to the KB. In a similar way, utilising the resource semantics component, each cluster registered in the platform sends its description to the RF. Depending on the type of input data, Flask guides AllegroGraph to create further relationships at both individual and default graph levels. This facilitates the timely retrieval of specific data needed by the platform as well as provides inference on the possible function allocations.

The architecture follows the microservices approach as both components (i.e., KG and Flask) are containerized (using Docker⁹) and integrated as a single service allowing additional services to be added

⁸ <https://allegrograph.com/products/allegrograph/>

⁹ <https://www.docker.com>

without affecting the existing component. The interactions mentioned above, along with the internal architecture of the RF, are depicted in Figure 14.

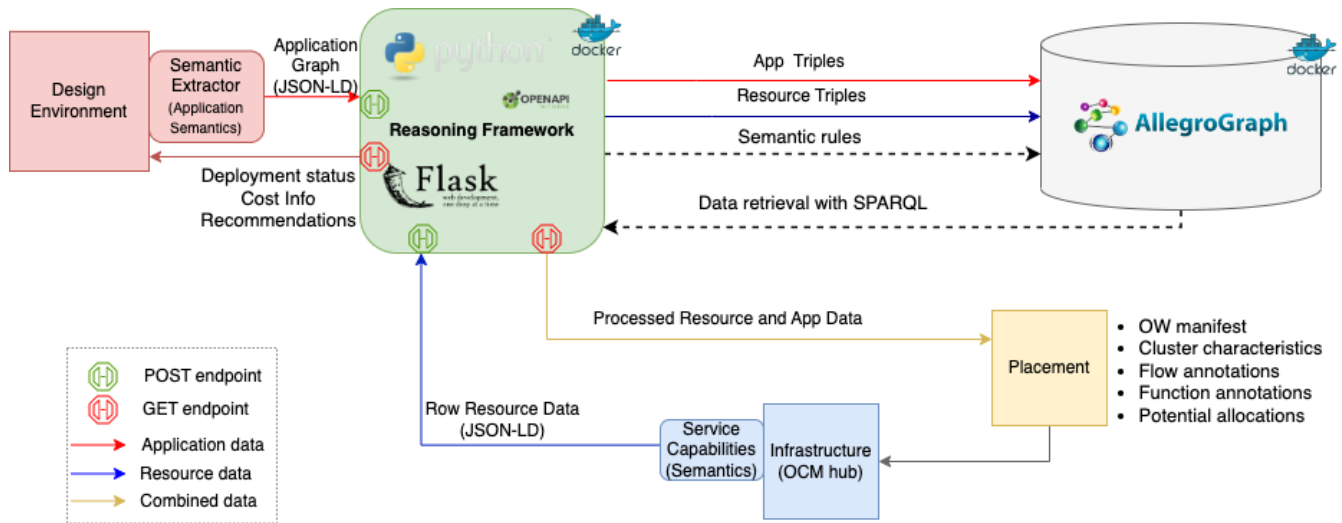


Figure 14 Reasoning Framework architecture and interactions with other components

Main issues handled by the component

RF is responsible for completing the following needs of the Physics platform:

1. Offer reasoning capabilities and semantic inference over the application and resource semantic descriptions.
2. Filter the number of candidate services that may be used for the deployment of a given application graph, enhancing the optimization process of Global Continuum Placement (T4.3).
3. Provide abstracted querying nodes to:
 - a. Retrieve application graphs from the Semantic Extractor (T3.2).
 - b. Retrieve information about the available resources from the Orchestrator (T4.5) Hub.
 - c. Query the semantic descriptions of the available resources provided by the Service Semantics component (T5.1).
 - d. Retrieve performance metrics from the PEF (T4.2) that will be considered during the placement and facilitate runtime adaptation.
 - e. Provide relevant information (i.e., list of application graphs, graph name, owner etc.) to the Design Environment (T3.1) that can be included during the application graph specification.
 - f. Provide the candidate resources that could be used to deploy the given application to the Optimizer (T4.3).
 - g. Provide the necessary options and executables that need to be defined in the deployment configuration to the Orchestrator (T4.5).

Inputs

- Individual application descriptions (e.g., function requirements, functions sequence, location constraints etc.) provided by the Semantic Extractor in JSON-LD format.
- Individual Cloud/Edge service descriptions (e.g., type, CPU, RAM, location, etc.) provided by Resource Semantics also in JSON-LD format.
- Performance evaluation of the available services for an individual application provided by the Performance evaluation Framework in JSON format. RF will translate these data into triples and then create the relevant connections between the graph nodes in the KB (e.g., <evaluationTest1, hasPerformanceScore, "85%"> <evaluationTest1, appliedIn, InferenceFlow>, <azure, hasPerformance, evaluationTest1>).

Information related to new resources registered in PHYSICS (i.e., Cluster name, IP, OW credentials) provided by the Orchestrator in JSON.

Outputs

- Flow/function potential allocations (i.e., the resources which can host each flow/function of the given application) to the Global Continuum Placement Optimizer.
- Flow/function user-specific annotations to the Global Continuum Placement Optimizer (i.e., optimization goal).
- Characteristics of the available resources or Resource Graph (i.e., CPU cores, RAM, architecture, locality, performance scores)
- Application Graph to the Global Continuum Placement Optimizer that is passed directly to the Orchestrator along with the final placement decision (DeploymentGraph).
- Deployment status, design recommendations and relevant information to the Design Environment.

2.7 Performance Evaluation Framework

Component Description

The main focus of this component is to enable informed decision making on various aspects of the platform and application execution based on retrieved performance evaluation data from the execution of designated workloads towards target functions.

To this end it needs to be able to trigger relevant executions on demand towards target endpoints (e.g. function invocation APIs) based on diverse scenarios needed for evaluation. Such scenarios may originate from the nature of the FaaS platform, including for example the effect of cold/warm/hot container start consideration, the limitation on function concurrency factors and the relation to burst or trace driven requests, investigation of scheduling strategies that aim at maximizing context reuse in functions etc. Other needs for investigation may include the analysis and prediction of function execution time and memory usage (which could also be used for cost estimation) as well as tailored performance analysis of the reusable patterns and their parameters available in the Design Environment. As an example, the size of a Node-RED flow may influence its performance in relation to the way it is executed (as a function or as a service), along with other parameters such as hot/cold function execution. Finally, the evaluation of the ability of available services/resources on typical workloads (e.g., benchmarks or candidate functions) is another goal that may aid in more informed resource selection during deployment and runtime management.

Following the above, the component functionalities need to be made available through relevant API endpoints, whether this relates to load generation triggering or result retrieval, so that they can be tailored to arbitrary experiments needed. Following the data collection, the relevant QoS descriptions of the Service Resource or Application model may be populated, therefore this component should also participate in the definition of the relevant semantic structures and produce results for their population. Given that at any given point in time it is very difficult to acquire all relevant performance metrics for all possible combinations of relevant parameters (resource, application, environment etc), this component needs also to be able to create performance models from a limited number of experiments, so that it can reply to requests for performance data from configurations it has not actually benchmarked. This for example might be predictions regarding the function execution time of a given configuration and/or other relevant metrics. In patterns that need a form of self-adaptation, in order to adapt to varying conditions of execution, relevant models may be created in order to support this process, linking the pattern configuration parameters with aspects such as the anticipated traffic and the predicted QoS.

The capabilities of this component may either be triggered by the Global Continuum Placement in the quest for an optimized deployment trade-off. Alternative usages may also include the invocation from the Design Environment in order to get information on pattern configuration, or by the pattern implementation itself

during runtime (e.g., for getting the predicted parameters based on the current conditions of execution). Other uses include the application developer being able to benchmark their created functions or evaluate the performance implications of a changed function or workflow structure/implementation.

Given that the component lives and breathes in the FaaS domain, it was considered to change the initial architecture of the main load generation from a cluster-based, Jmeter-containerized execution to a function based one, based on an implemented Node-RED flow. This alternative design gives a much more modular approach, enhanced packaging and scalability as well as portability to any available FaaS platform, without the need to setup and operate separate load generation clusters. This aids in having much less complicated test orchestration, with the main task of the PEF being to launch the function load generators and concatenate the results in case more than one of them are used.

Furthermore, in order to have a more targeted function performance analysis, it was considered best to focus directly on the produced functions from the application development and not generic benchmarks like FunctionBench. This process may be integrated with the main design process of a function. With the packaging as a function, it is also easier for any platform service to directly invoke the load generation, in order to test the change in a setup parameter, scheduling used etc.

The subcomponents of this component and the interaction with other components are depicted in the Performance Evaluation Framework diagram of Figure 15, which has been updated in order to highlight the new advancements in the design.

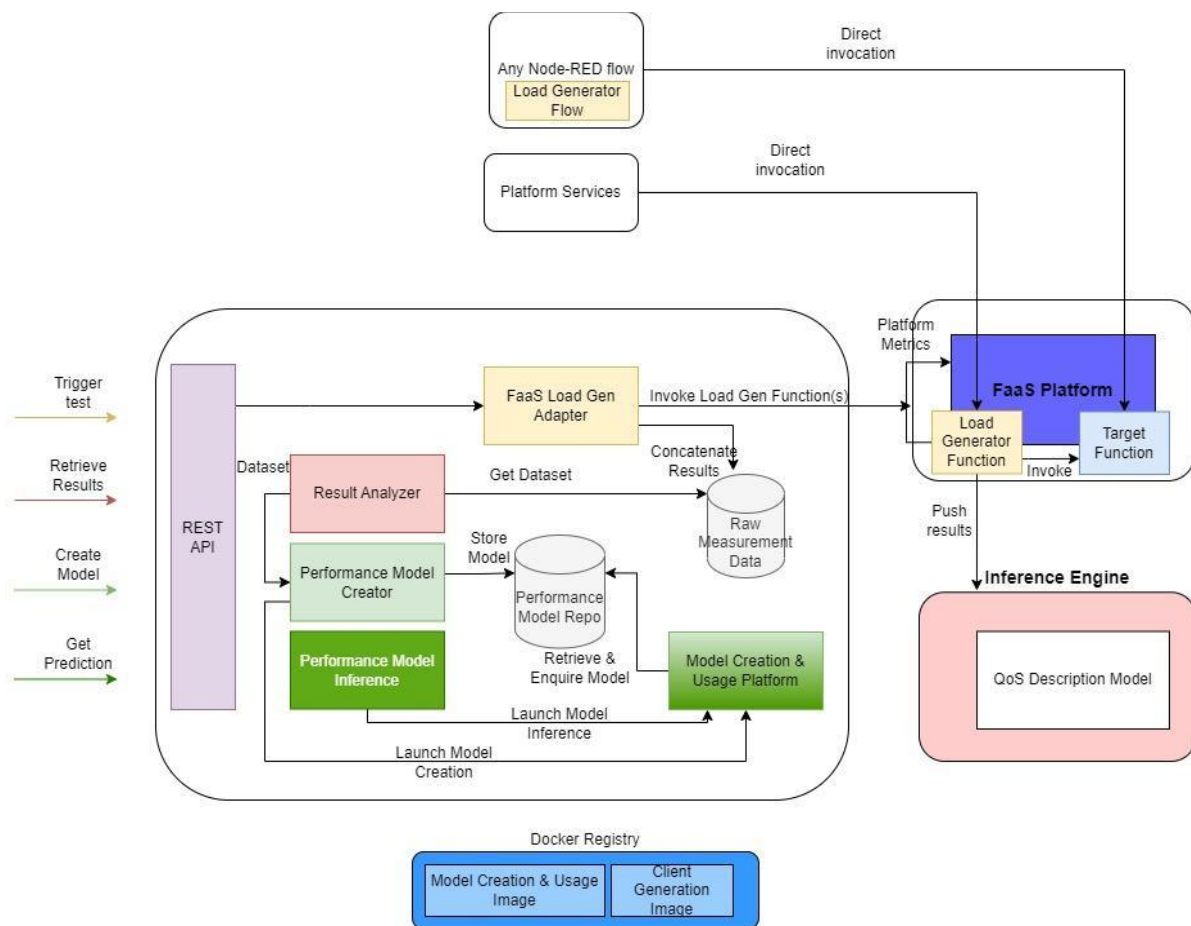


Figure 15 Performance Evaluation Framework Diagram and Interactions

Initially, a REST API layer exposes the main functionalities of the component (test triggering, result retrieval, model creation and model usage/inference). Results are retrieved from the clients, for client-side response times, as well as the FaaS platform (for platform related metrics) and stored in an internal repository. Following external requests, these data are queried by the Result Analyzer and returned to the caller. The Result Analyzer can also be queried by internal components, i.e. the Model Creator, in order to retrieve the necessary dataset for model creation, following an external according call. Once the model has been validated and finalized, it is stored in a model repository, where it is accessible by the final operation of the Model Inference. In this case the external call provides input arguments for the model and needs the prediction of the output, based on the model structure.

Given the transformation of the design from a Jmeter based load generator to a Node-RED subflow one, the new implementation also gives the ability to use the load generation subflow in any NODE-Red environment against a target OW platform. The availability of this subflow as an invocable function already deployed on the FaaS platform enables also its usage more easily in any testing scenario from the platform services.

Main issues to be handled by the component

- Integrate performance evaluation test towards a target function in the typical function development lifecycle of an application
- Define and design workloads that are representative of testing scenarios, use case needs or anticipated usage
- Analyse the execution instances of user functions, providing insights with relation to their predicted execution time, thus aiding in aspects such as scheduling decisions, placement etc.
- Include and enable the evaluation information to be used by other components in the context of service selection
- Enable the on-demand execution of stress tests from various components of the PHYSICS platform in order to evaluate different strategies in deployment and runtime management

Inputs

- Target functions for benchmarking
- FaaS platform runtime statistics.
- Triggers for launching tests or other requests.

Outputs

- QoS metrics model definition and metrics.
- Resource and application models QoS instances population with the results from the measurements
- Performance models and predictors for various aspects such as function runtime prediction, co-allocation performance degradation, hot/cold/warm start execution, performance of Node-RED flow function versus service, pattern parameters definition etc.

2.8 Global Continuum Placement

Component Description

The main goal of the Global Continuum Placement component is to perform the decision making to efficiently select the right compute resources for the placement of the different tasks of the applications to be executed on a hybrid edge-cloud infrastructure.

The latest version of the specific component has the ability to schedule application workflows and proposes a resource allocation and deployment schema for each workflow selecting resources across an infrastructure composed by different public cloud, on-premises, edge and even HPC clusters of computing resources, in an optimal and timely manner. Each application workflow will come as a graph of tasks - functions- (in a FaaS programming model) with particular requests in resources (CPU, Memory, Bandwidth, GPUs, etc), possible constraints (execution only upon one type of infrastructure: edge to satisfy data sensitivity/locality obligations, etc) and scheduling objectives (energy, latency, data movement minimization, etc) based on particular scoring techniques. The component considers the computing resources characteristics (number of total CPUs, amount of available bandwidth, energy, cost, etc) and availability (remaining amount of memory available for allocation, etc) and matches this with the application graph needs and specific objective (performance, energy, etc) weights. Based on these inputs, the latest version of Global Continuum Placement component performs placement respecting the constraints while aligning with the multi-objective dimension and uses the basic best-fit scheduling policy. We are currently working on an improved version of this algorithm using Linear Programming to calculate optimal placement when considering multiple objectives. Eventually, the user will have the possibility to select the scheduling algorithm of its choice, but the component will be able to automatically set the most adapted algorithm based on the context. Simple policies (such as First Fit or Round Robin) that consider workflows requests, constraints and single objectives will provide faster but non-optimum results whereas more complex algorithms (based on Linear Programming or genetic) that consider multiple objectives and various QoS to address will return optimal or sub-optimal results in a less timely manner. Figure 16 provides the high-level view of the GCP component along with the direct interactions to other Physics components.

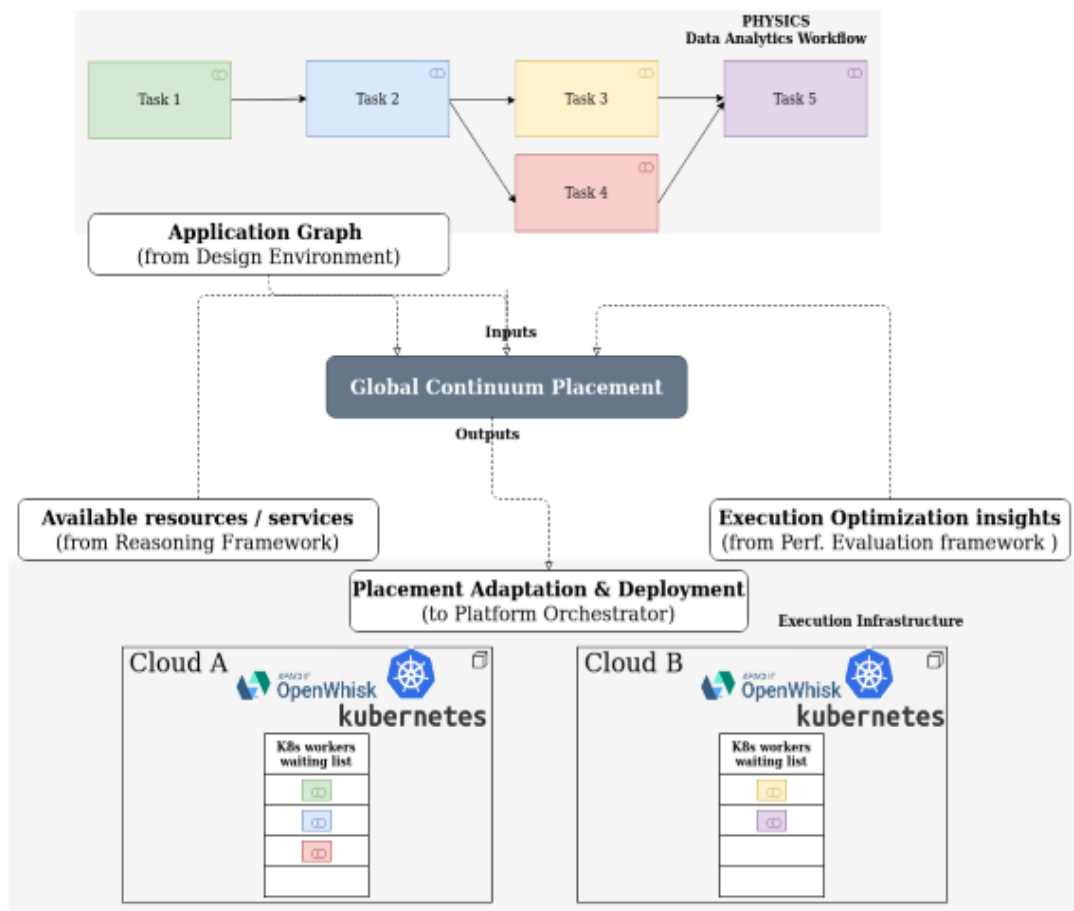


Figure 16 Global Continuum Placement high-level view and relation to other components

Internally, this component is composed by: 1) a subcomponent that consumes inputs related to the application graph expressing the need of resources and constraints coming from the Services semantic models along with cluster resources availabilities coming from the Reasoning Framework and the deployment optimization possibilities coming from the performance evaluation framework; 2) the scheduling algorithm is expressed as a separate module built within a wrapper with the ability to be programmed in different programming languages and to be extracted into a simulator in order to experiment and evaluate its performance, 3) the output subcomponent which provides the scheduling decision is pushed as a YAML file to the centralised orchestration component. Figure 17 provides the internal architecture of the GCP component showing the details of its internal subcomponents along with the external tools and inputs & outputs.

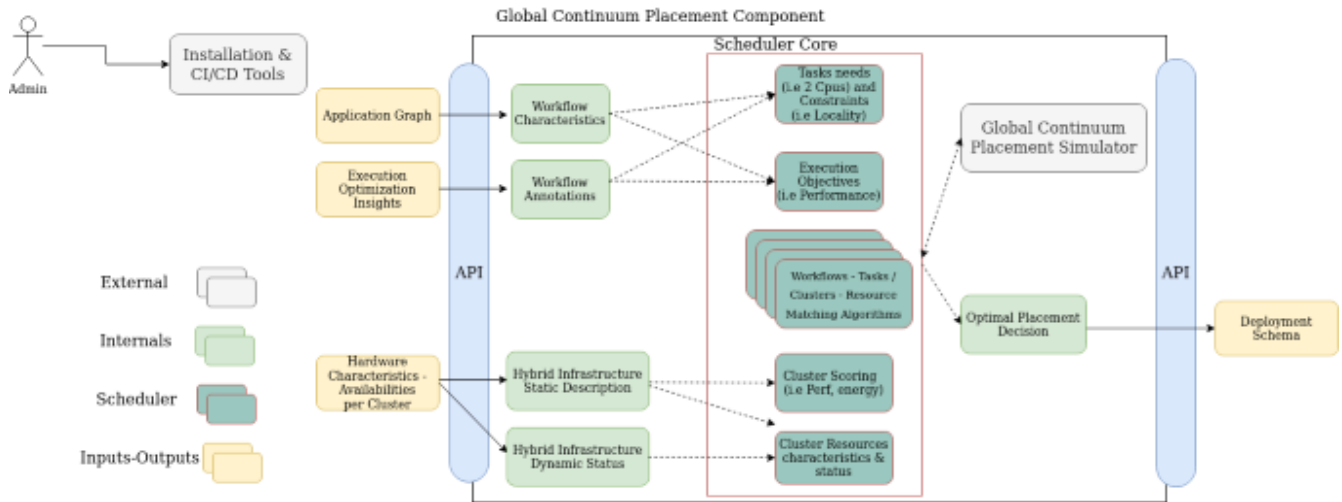


Figure 17 Global Continuum Placement component internal architecture

Main issues to be handled by the component

- High-level task placement of applications to the compute resources (or services) of the Global Continuum.
- Optimal matching of application functions' needs to the underlying compute resources availability.
- Efficient scheduling of multiple applications on the Global Continuum in a timely manner, considering different constraints and various solutions for optimizations.
- Dynamic adaptation of task placement decisions based on new parameters, such as performance, energy, etc.

Inputs

- Application graph decomposed in functions including the resource needs and constraints coming from the applications semantic model (semantic extractor component) in a YAML format
- Deployment graph containing the available and adequate resources (or services) of the hybrid edge-cloud continuum coming from the Inference Engine (or reasoning framework) in a YAML format
- Deployment optimization possibilities related to the execution of specific tasks coming from the performance evaluation framework

Outputs

- Deployment decision schema featuring the global continuum resources selection and task placement decisions to be transferred to the adaptive platform deployment, operation and orchestration component in the form of YAML file.
- Deployment decision schema details to be transferred directly to the local cluster scheduler component as a YAML file (this is not yet supported from the local cluster scheduler component).

2.9 Distributed Memory Service

Component Description

The Distributed Memory Service (DMS) allows sharing data between functions invocations. Functions in FaaS frameworks are stateless and any data sharing must be done through a remote data store which is expensive in terms of latency. The DMS component provides an in-memory distributed state service that allows functions to store objects out of main memory and share them among other functions efficiently. Several issues must be considered in order to achieve good performance in a FaaS scenario. The DMS should run collocated within the nodes where functions are executed in order to avoid access to remote data. Since the same function can run in several nodes, data should be replicated. The consistency of data should be preserved so that, even if functions running on different nodes update the same data, data will converge (eventual consistency). The DMS provides a simple interface to access the data: *get* and *put* operations

Figure 18 (top) shows the internal architecture of the DMS. The DMS design is based on *Pocket*¹⁰, however, the storage system used by *Pocket*, Apache Crail¹¹, is not supported any more (from June 2022). At this point we are evaluating other alternatives among them KeyDB¹² is the most promising for its performance and built-in replication features. The DMS has three sub-components: one *controller*, one or more *metadata servers* and one or more *storage servers*. The *controller* is a subcomponent that allocates storage resources and decides the data placement and scales the metadata and storage servers. Moreover, it deploys a *resource monitoring daemon* on each node where the DMS runs. This process sends CPU and network statistics to the controller frequently. The controller uses these metrics to decide which subcomponent must be scaled up or down. The *metadata server* redirects clients' requests to the storage server allocated by the controller. It also sends storage servers capacity utilization statistics to the controller. The metadata server was built on top of Apache Crail in the previous version of the component and it will be replaced soon. The *storage servers* are in charge of storing the data. They can be used with different storage media such as: DRAM, NVMe, SSD or HDD. The next version of this component will not offer different storage servers and data will be mainly kept in main memory.

Figure 18 (bottom) shows how a workflow that consists of a sequence of three functions (actions) deployed in the PHYSICS Platform accesses the DMS. The DMS provides a library with the basic functions for accessing data (*get/put*) and other functions required to interact with the DMS. Solid arrows in Figure 18 represent the operations for accessing the data from functions, while dashed arrows represent other operations needed for accessing the data in the DMS. Blue arrows (steps i,ii,iii) represent the control functions (register, allocate and assign resources and de-register), and black arrows represent *get/put* interactions between the actions and the Distributed Memory System.

Initially, when a workflow (sequence in this case) is invoked the *register* function is executed (step i). The controller registers the sequence with the metadata server (step ii). The controller returns a *sequenceID* and a reference to the metadata server(s). The first time a function issues a *get/put* operation, the metadata server is accessed to obtain the location (IP) of the assigned storage server and a connection (steps 1 and

¹⁰ <https://www.usenix.org/conference/osdi18/presentation/klimovic>

¹¹ <https://crail.incubator.apache.org/>

¹² <https://docs.keydb.dev>

2). This IP is stored for future access to the storage server. Next, data is written and read from the storage server (step 3). When the last function of the sequence (workflow) completes the *de-register* function is executed and the resources allocated by the DMS to the workflow are released.

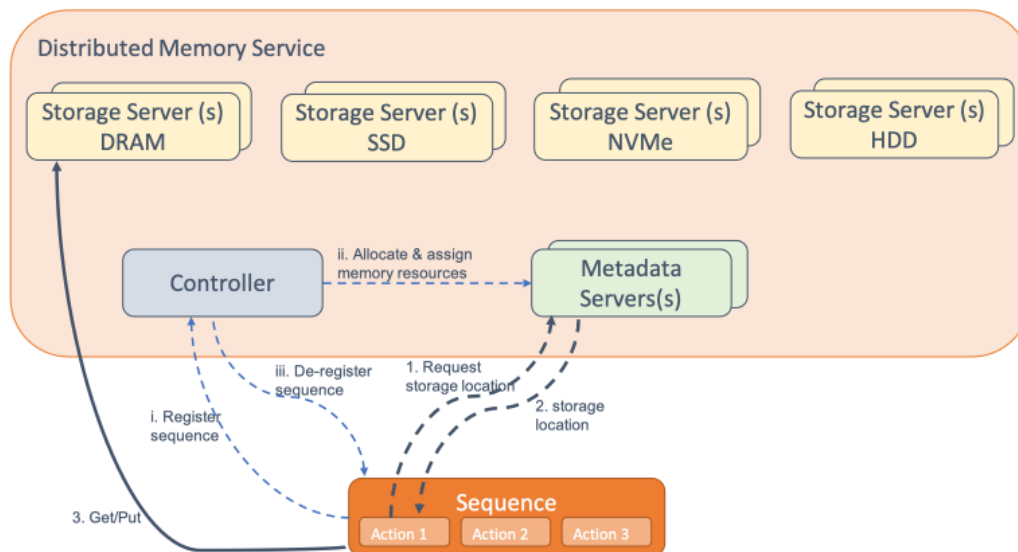


Figure 18 Distributed Memory System Architecture

Main issues to be handled by the component

The main requirements of the Distributed Memory Service are:

- Data consistency
- Support for different cloud providers (AWS, Google Cloud, ...)
- Provide fast data access to functions

Inputs

- The DMS uses the description of the cluster (number and location of nodes, type and size of available storage...) where a workflow will be executed.
- Workflow definition: functions part of the workflow in order to share connections.

Outputs

- The information stored in the DMS.

2.10 Adaptive Platform Deployment, Operation & Orchestration

Component Description

The objective of the Adaptive Platform Deployment, Operation & Orchestration component is to enable easy dynamic deployment orchestration, reconfiguration & adaptation of the applications defined in the deployment graph (aka Global Service Graph). That deployment graph is produced by the Global Continuum Placement component with all the information present in the application graph previously obtained from the Reasoning Framework and the optimal placement infrastructure selected from the candidate set proposed by the Reasoning Framework. Based on the deployment graph definition, the orchestrator operates the deployment on the different clusters managed by PHYSICS. To achieve this deployment, the

orchestrator uses a translator component (see Figure 19) that parses the information from the deployment graph into a definition that is consumed by the Resource Management Controllers (WP5 component described later in this document).

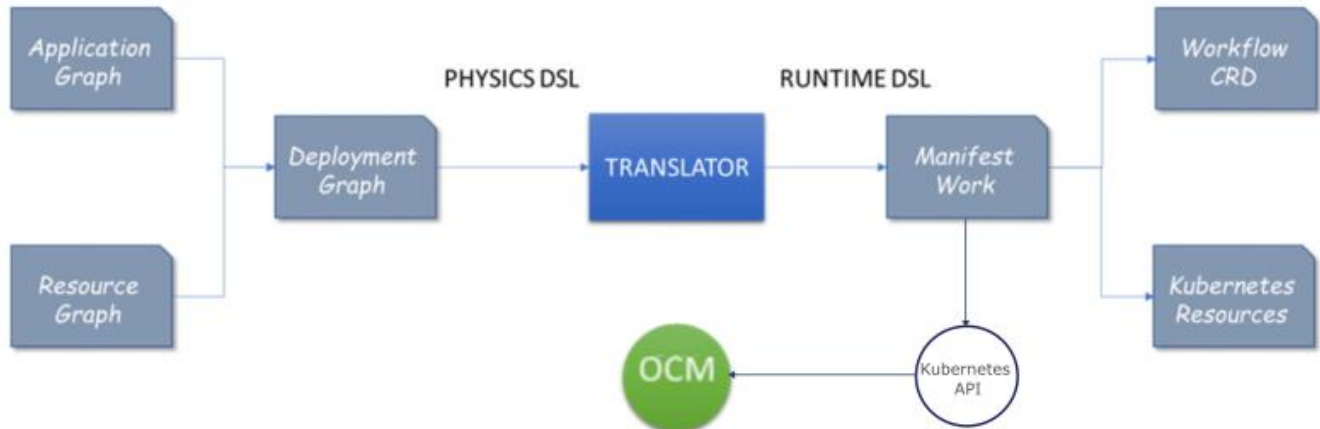


Figure 19 Translator service, part of Adaptive Platform, Deployment, Operation & Orchestration component

This translator service acts as a bridge between the semantic representation language of PHYSICS and the domain specific language (DSL) of the Resource Manager API (OCM) and cluster native resources (manifests). The Resource Manager Controllers use a ManifestWork object as an envelope to wrap cluster native resources and the custom resource definition (CRD) of a workflow in PHYSICS. This PHYSICS Workflow contains the necessary information to deploy the application functions in the target FaaS platform. The translator service will be deployed in the PHYSICS hub cluster together with the rest of the core components of PHYSICS. This translator was named "semantics & placement schema parser" in the first version of this Reference Architecture.

An additional service named "OW-proxy" will oversee registering the functions in the FaaS platform at the target "managed" cluster. This service will be deployed in all the target clusters in the catalogue of resources offered by the PHYSICS platform. This service will deal with the specific API of the FaaS platform to create the functions (or actions) included in the application workflow and optionally the linear sequence of functions execution order (Figure 20).

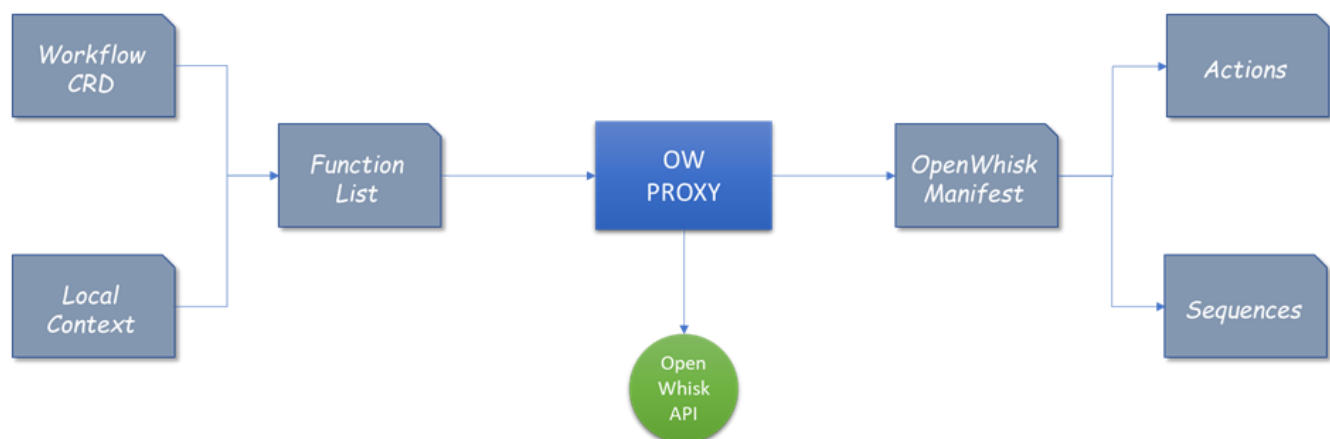


Figure 20 FaaS Proxy service, part of Adaptive Platform, Deployment, Operation & Orchestration component

Figure 21 shows the above two services of the component (in red) integrated with the rest of the PHYSICS components that are involved in the deployment pipeline of a PHYSICS application workflow.

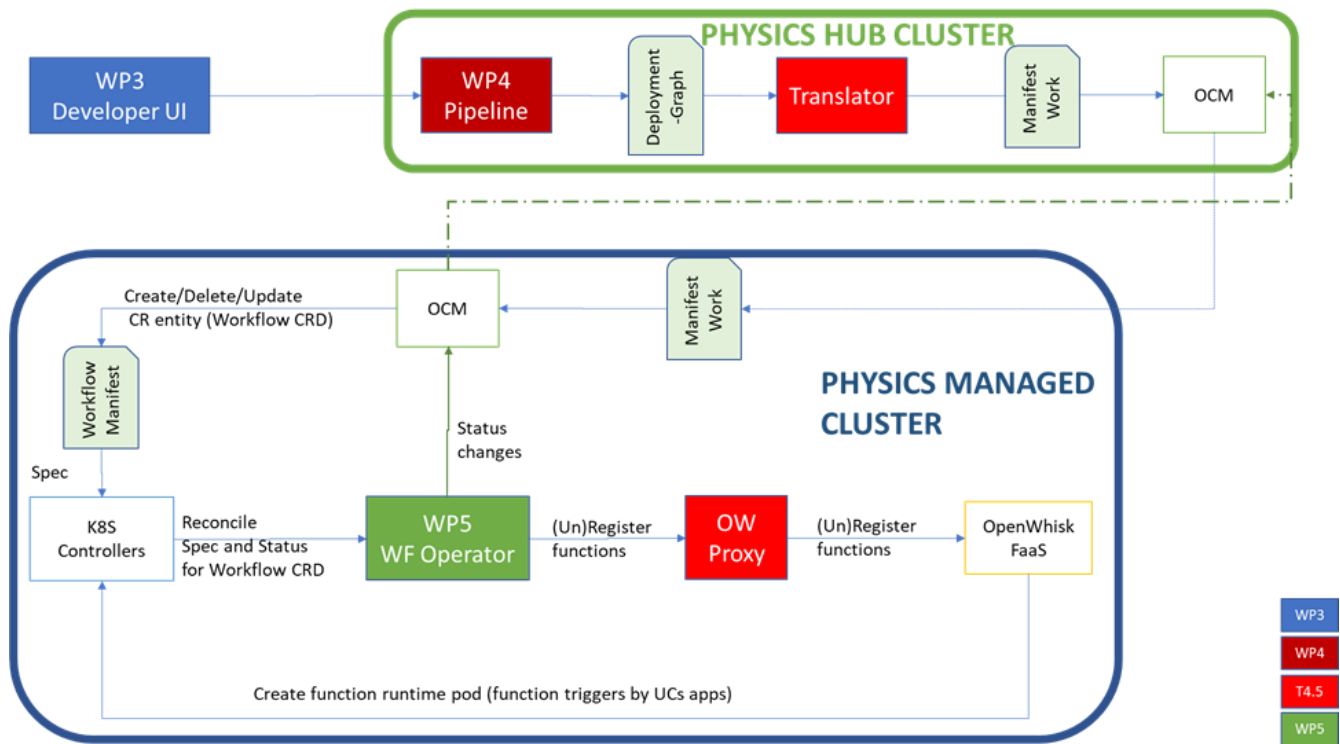


Figure 21 Deployment pipeline of a PHYSICS application workflow

This component will also implement a service to monitor the QoS (Quality of Service) expected by the application owner by means of semantic annotations in the application graph (translated into the final deployment graph). We name this component as “QoS Evaluator / Alert System.” This service implements two functionalities: an evaluation loop to periodically check the current metric value with respect to the defined threshold and an alert system to notify the PHYSICS Hub about any violation of the expected QoS. This service was named “QoS & QoE Runtime” in the first version of this Reference Architecture. The evaluation loop functionality will need to connect to the metric time series database storage (Prometheus) of every target or managed cluster used in PHYSICS to periodically check the current QoS. This time series database and the collection process of metric values (samples) will be implemented by the metric system installed in each managed cluster using some open-source tool like Prometheus. Figure 22 shows the flow of the global runtime adaptation loop with the interoperability between several components of the PHYSICS core platform.

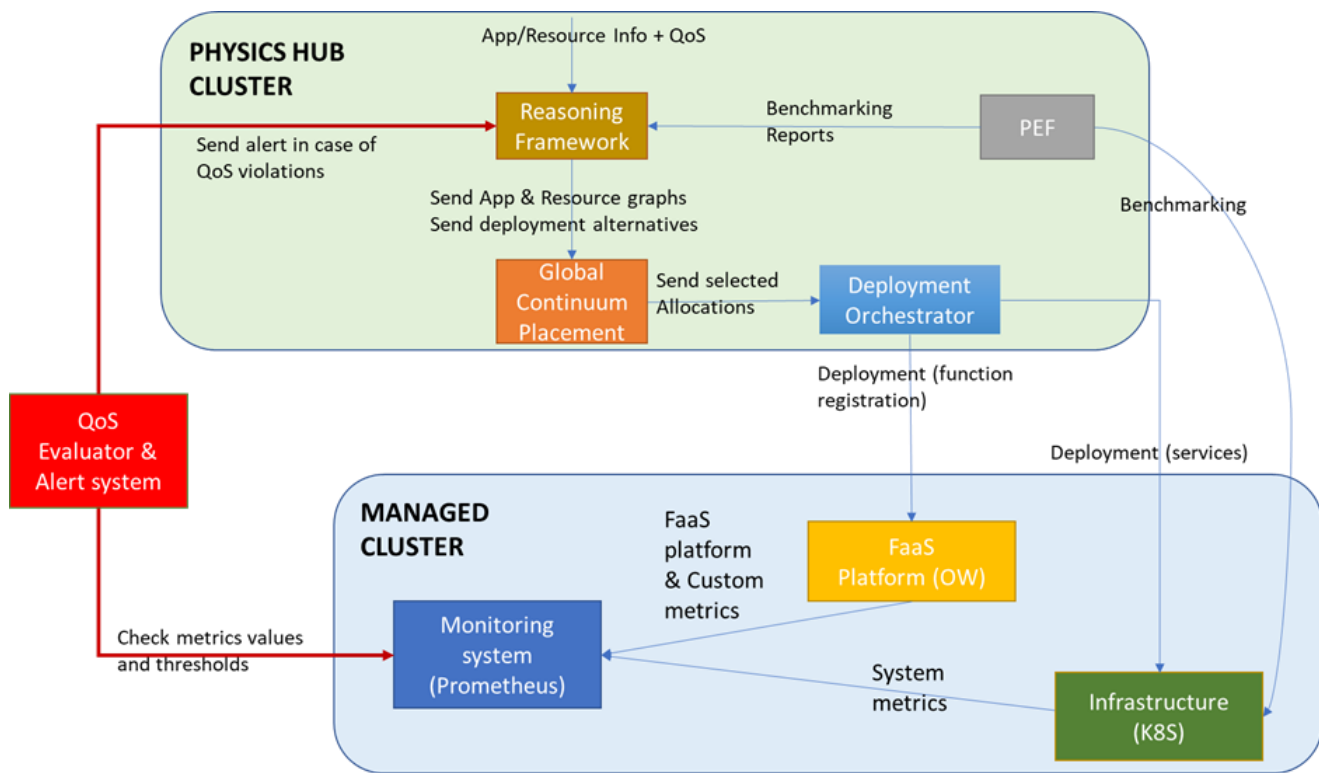
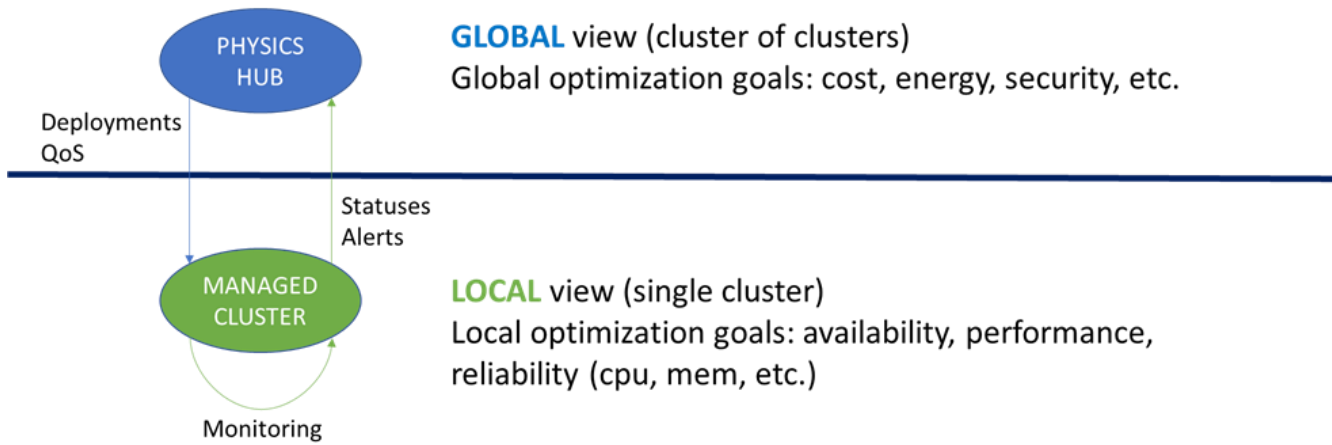


Figure 22 Global Runtime Adaptation flow

All these services that comprise this Adaptive Platform Deployment, Operation & Orchestration component described here are the executor arm of PHYSICS core platform and they are commanded by other components of PHYSICS that implement/make the smart decision part of the process. In this way this component decouples the technical details to interoperate with different Resource Manager APIs from the smart decision support maker components of PHYSICS.

The “Global” runtime adaptation is a concept to separate the two different control planes that can interact in the runtime adaptation process. There is a global control plane implemented by the PHYSICS Hub cluster with a global view of what is going on in the deployed application and a local control plane implemented by the resource/container orchestrator in the target or “managed” cluster (Figure 23). The last one is implemented by some Kubernetes flavours.



QoS: Quality of Service or KPIs (Key Performance Indicators)

Figure 23 The two control planes

These local cluster control planes already implement out of the box monitoring and reactive runtime adaptations functionalities for the application/workloads deployed in the cluster like redeployment in case of workload failed or horizontal scaling. We customise these capabilities for local runtime adaptation using PHYSICS components like the Scheduling Algorithms, Co-allocation Strategies and Resource Manager Controllers.

All the services implemented by this component will expose their functionality using a REST API interface following the Open API standard.

2.11 Service Semantic Models

Component Description

The main goal of the Service Semantics Models component is to capture information on the functional and non-functional properties of the available cloud resources of a cluster and transform this information to comply with the semantic rules and relationships formed in the designed ontology. The ontology is a crucial part of the component. It comprises the semantic relationships formed between resource individuals at various levels (*Cluster, Cluster Node, Serverless Platform, GPU/CPU/, etc.*) and the characteristics of them (*allocatable values, endpoints, versions, location, etc.*). This collection of semantics allows the description of cloud schemata whether they are on premises, either as cloud or edge clusters, or make use of cloud vendor offerings.

Other than the ontology design, information gathering is another crucial functionality that the component includes. In order to alleviate time consuming semantic annotation of resources by domain experts the component incorporates methods to directly draw information about a cluster in an automated fashion, when that is possible. The respective methods are designed to be Kubernetes API compatible, being the most mainstream cluster management software, which also comes with various distributions some of which are well suited to form clusters on the edge. Finally, the process of information transformation to semantics, from the raw cluster information to the ontology, is targeted with the use of a Python service that includes various relevant libraries such as Flask and Owlready2.

In the PHYSICS platform semantics are mainly utilised to enable the application deployment and resource management/optimization processes by the relevant components. To that end, populated ontologies that come from the Application Semantic Models component (T3.2), hold information about the requirements and constraints for application deployment and are compared with the resource properties and capabilities presented in the populated ontology provided by this component to the Reasoning Framework (T4.1). Finally, further optimizations are made to select the most efficient deployment schema by the Global Continuum Placement (T4.2). Essentially, centralised information about all the clusters resides within the AllegroGraph database that is part of the Reasoning Framework, and this is where other components reach out for the ingestion endpoints (Figure 24).

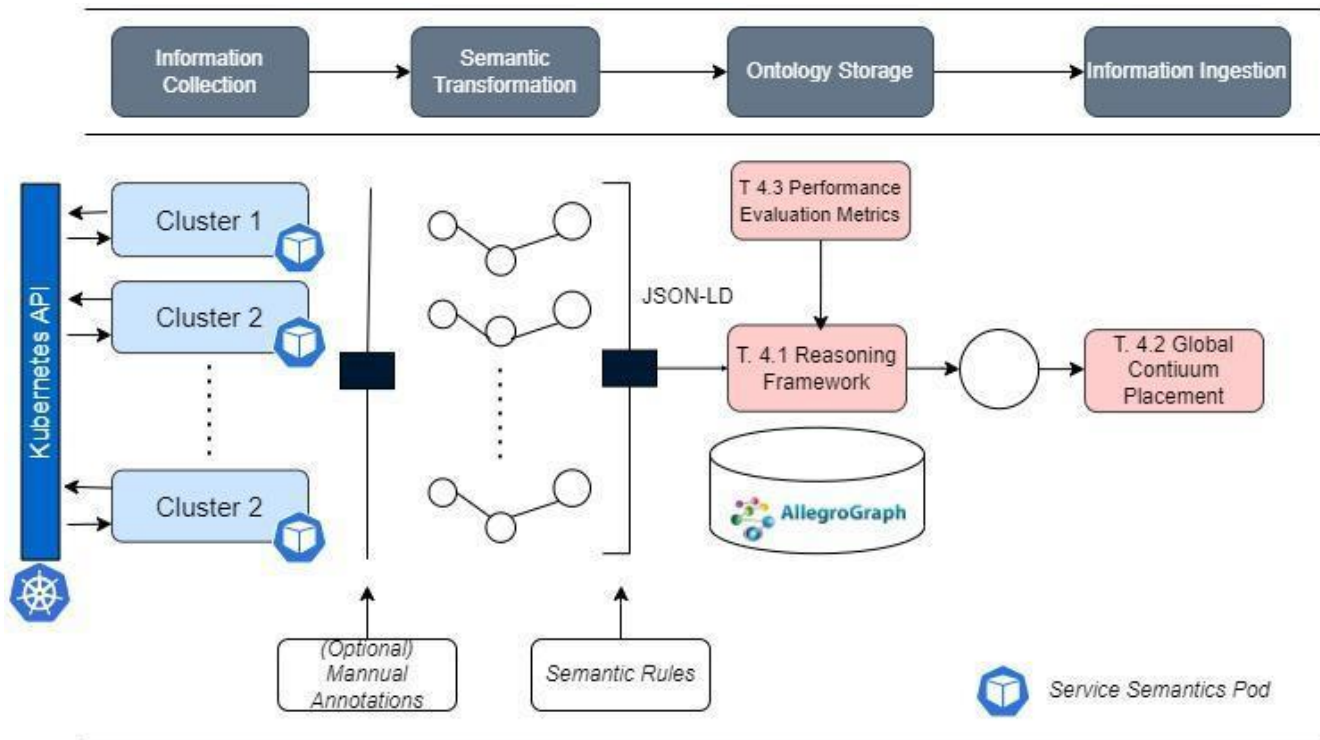


Figure 24 Service Semantic Models Architecture and semantics interactions

Main issues to be handled by the component

The service semantics component is responsible for tackling the following within the PHYSICS environment:

1. Modelling of a schema that captures the service semantics, an OWL ontology capable of inference.
2. Automating information gathering, when possible, a service to call and manage information that comes from Kubernetes API.
3. Interfaces to easily provide information when no automated method is available.
4. Information exchange endpoints to communicate with all the relevant components.
5. Transformation of the gathered information to ontology individuals and relationships.
6. Creation of various semantic rules that produce inferred knowledge automatically from the available information.

Inputs

- The OWL resource ontology modelled within the scope of this task.
- Information on resource performance provided from the Performance Evaluation Framework (T 4.3) and/or aggregated metrics from application deployment monitoring.
- (OPTIONAL) Manual ontology annotations for a cluster by accessing the endpoint of the respective service

Outputs

- A populated OWL ontology describing the properties of a cluster, depicted as JSON-LD and to be ingested by the Reasoning Framework.

2.12 Local Adaptive Scheduler

Component Description

The main goal of the Scheduling Algorithms component is to provide the local cluster scheduling capabilities to enable the execution of functions as parts of FaaS applications. In this context, the component will take into account specific characteristics and challenges of FaaS applications and will try to perform efficient sharing of computational resources (CPU, memory, storage, network) taking into account aspects such as functions' priorities, dynamic load-balancing and energy efficiency. The local scheduling algorithms are represented by the 2nd level scheduler taking place locally on each cluster and allows the selection of the most adapted resources on each cluster as shown in the high-level example of Figure 25.

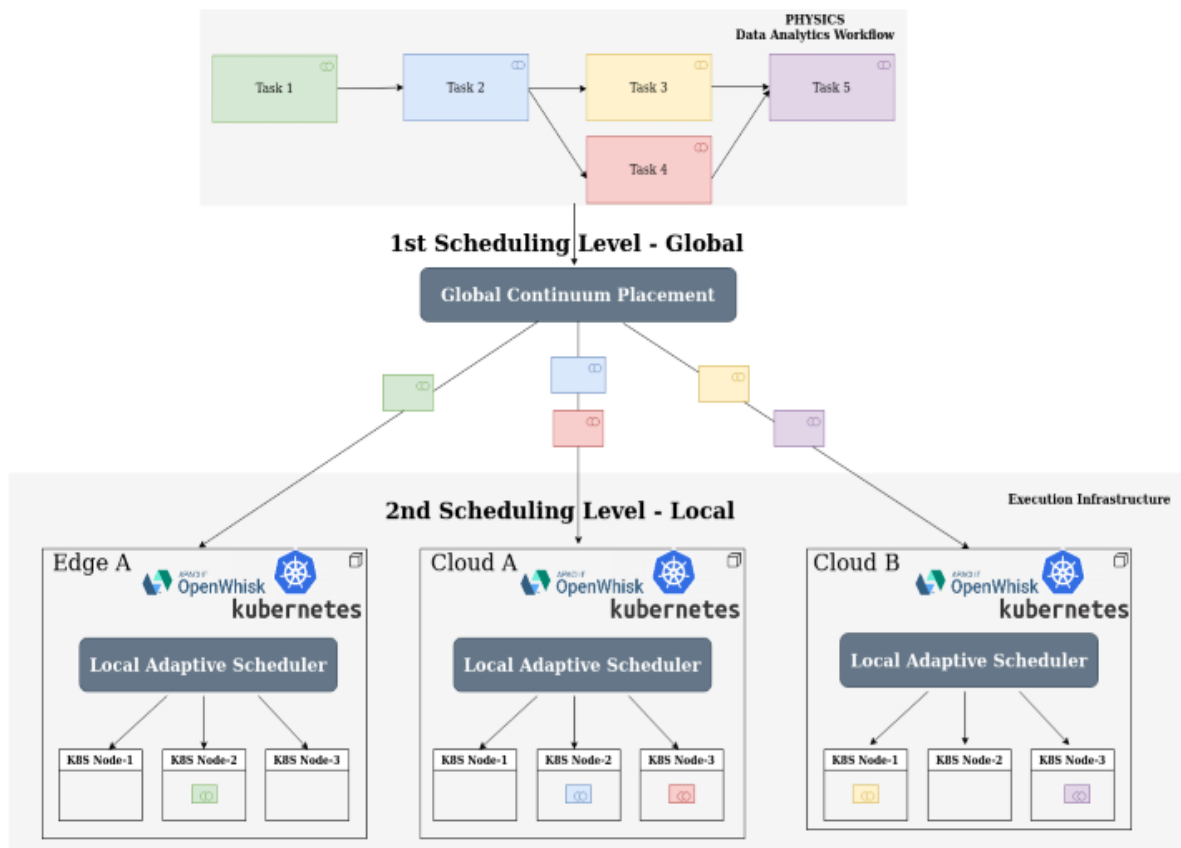


Figure 25 Local Adaptive Scheduling Algorithms and its relation to the Global Continuum Placement for the 2-level scheduling of the continuum

The current version of this component is mainly a new Kubernetes scheduling policy which minimises the cold starts of functions. This is particularly important for FaaS applications because their execution time can be quite small while the download time of the environment (container image) may be large and even larger than the execution time. So, reducing the download time of containers is important for the FaaS paradigm. The new proposed Kubernetes scheduler takes into account the existence of Containers' Layers and tries to favour the execution of functions on nodes where layers of the containers to be deployed already exist. For this the particular algorithm needs to get on one side the available layers on each node (name and size) and on the other side for each new pod compute a score per node considering the cumulative size of already available layers. Hence this particular component currently makes use of details coming from the node container runtime interface and brings this info in the level of Kubernetes in order to enable a scheduling decision through a new scheduling policy. The new scheduling policy can be selected by a webhook defined by the resource management controllers.

Besides the interaction with the resource management controllers, the current version of the local scheduler does not interact with other components, but this will change in the upcoming versions where the component will communicate directly with other components to get details related to monitoring, high-level scheduling propositions, co-allocation, adaptivity, etc.

Each local cluster has the ability to select a scheduler among a group of different available schedulers defined by a combination of Kubernetes profiles, policies and algorithms. Different schedulers may be adopted or implemented to cope with issues such as the cold start of functions by selecting resources where the function's container has been previously downloaded either completely or at least some layers of it. We are currently investigating the usage of energy related heuristics in the policies of the local scheduler. For this, we consider ways to monitor the energy consumption of the executions and try to study techniques on how to minimise it. Another example of scheduler may enable functions to be deployed upon already deployed containers which implies the need of further isolation among the different functions that may be deployed on the same container. Other possible scheduling algorithm is the one that considers the collocation of CPU-bound functions with Memory-bound functions to optimally pack functions and utilise computing resources on each cluster. Another possibility may be the automated setting of requested resources and limits of each function based on previous allocations. This can be automatically adapted on-the-fly and can even use Machine Learning for optimal adaptation.

Furthermore, since we adopt Kubernetes as the default PHYSICS cluster manager, each cluster will have the ability to deploy different scheduling algorithms per deployed function and even allow the simultaneous usage of multiple algorithms within the cluster at each moment.

The upcoming version of local scheduler component will be composed by 1) a subcomponent that will consume the annotations expressed in the YAML file of the orchestrator including higher level scheduling preferences and constraints coming from the Global Continuum Placement and the Application semantic models along with cluster resources availabilities; 2) the scheduling algorithm which will be expressed as a Kubernetes scheduler packaged in containerized form. The scheduling algorithm may also be expressed as an OpenWhisk scheduling policy to enforce specific aspects related to OpenWhisk FaaS execution. The latest may allow to keep track of the subflow related dependencies among the functions and possible subflow constraints, and 3) the output subcomponent which will provide the scheduling decision to be pushed through the relevant API to the Resource Management component.

Main issues to be handled by the component

- Efficient scheduling of FaaS applications' functions upon the computational resources of local cluster considering resource availability and tasks' needs.
- Reducing the cold start of functions execution (container download time)

- Dynamic adaptation of resource allocations based on possible tasks' needs change (autoscaling is not handled in the current version of this component).

Inputs

- Local cluster resource availability status coming from the Resource Management Controllers through the internal Kubernetes API calls
- Containers Layers names and size per node
- Usage of containers layers for the environment of each function to be deployed on the nodes
- Global continuum scheduling decisions forwarded from the Adaptive Platform Deployment, Operation & Orchestration based on the decision of the Global Continuum Placement in the form of YAML (this is not yet supported but it is under development)

Outputs

- The resulting scheduling decisions favouring nodes that already have layers of the containers to be deployed. The scheduler which takes this decision is forwarded to Resource Management Controllers for deployment in the form of Kubernetes scheduling policy.

2.13 Resource Management Controllers

Component Description

The Resource Management controllers are a set of controllers and their respective APIs at the infrastructure layer that 1) manage and enhance different parts of the heterogeneous, multi-cloud infrastructure; and 2) provide the needed APIs to the upper layers.

The new Resource Management functionalities are implemented by extending the Kubernetes API by using Kubernetes Custom Resource Definition (CRDs) Objects with associated controllers that react to the information stored on them, following the declarative model established in Kubernetes.

The controllers are working at different layers on the infrastructure, from top to bottom:

- **OCM - Multi-cluster Management and Orchestration:** The Open Cluster Management (<https://open-cluster-management.io>) is a community-driven project which focuses on multicluster management for Kubernetes applications. It offers APIs for cluster registration, application distribution across them, as well as dynamic placement across the multiple clusters.
- **Submariner - Multi-cluster Networking:** Work is focused on enhancing the Submariner upstream project (<https://submariner.io>) so that it can work with different Kubernetes CNIs, as well as its integration with the multi-cluster manager (OCM) so that it can be easily installed, configured and used. The Submariner project allows application components in one cluster to reach other applications (or other components of the same application) located in remote clusters.
- **Scheduler webhook:** Provides the needed hook to select different scheduler algorithms per pod (see section 2.12), so that different applications can use different schedulers depending on their needs. In this case reducing the time for the pods to be started in a given node by using the knowledge about the existing container image layers.
- **Coallocation webhook:** Together with the previous one, it ensures the collocation engine gets executed before the pod is created, so that the proper hints about affinities/anti-affinities can be added to the pod spec before K8s starts processing it.
- **WorkflowCRD:** New operator in charge of defining the API (by extending Kubernetes API using Custom Resource Definitions) to be used by the upper layers (WP4 components) for registering the Functions in a given cluster. It is used through OCM for multicluster purposes. Besides defining the

API, it also defines the logic (i.e., controller) for processing the actual registration of the function, which in turns calls the OpenWhisk proxy to do so. Finally, it also reports the status of the applied actions so that they can be consumed both at the cluster level, as well as in the main (hub) cluster through OCM. In addition, the information stored in this Workflow CRD object is leveraged by the collocation engine to take its decisions.

- **uShift -- Low footprint Kubernetes deployment:** There is a need for low-footprint Kubernetes distributions, specially at the edges, where the computational capacity can be limited and the CPU architecture can be different (e.g., ARM). However, it is not only about being able to create a low-footprint single node Kubernetes node (e.g., KIND for developers, or k3s), but also about being able to control them in a centralised way (install, configure, manage) as the number of edges can grow fast. To tackle this problem, we are working on a new OpenShift/Kubernetes flavour optimised for edge devices named uShift (<https://next.redhat.com/project/ushift>). This will be integrated into the multicluster management (OCM), as well as the networking (Submariner).

Main issues to be handled by the component

- Install, configure and manage a distributed set of Kubernetes clusters of varying sizes (from central clouds to small edges)
- Deploy applications in a simple, descriptive way on the set of Kubernetes clusters
- Provide the needed APIs for the upper layers to:
 - Extend K8s API to support PHYSICS functions management (registration and execution)
 - Get monitoring information about the cluster and application (i.e., functions) status
 - Manage the clusters in a declarative way
 - Deploy the applications in a declarative way
 - Allow specific configurations of the applications and/or clusters, such as the scheduler to use, the isolation techniques through collocation preferences.

Inputs

- Cluster information needed to install/configure a new cluster (subnets, IPs, size, provider, ...)
- Set of YAMLs defining the application (workflow CRD) and its extra configurations such as the specific scheduler to use, location hints, resources needed, ...

Outputs

- Kubernetes clusters installed and managed from a central plain of glass (Open Cluster Management UI).
- Applications deployed on the selected clusters/nodes with the appropriated/optimized configuration.
- Specific scheduler to be used by pods set
- Collocation engine executed to get the set of affinity/anti-affinity rules defined

2.14 Co-allocation Strategies

Component Description

The Co-allocation Strategies component analyses resource consumption information of the cluster and applications, application function dependencies and their computational requirements and produces a function co-allocation strategy in order to improve the performance of functions.

Main issues to be handled by the component

This component must represent efficiently dynamic information such as resource utilization of the different nodes and functions and more static information such as the topology of the cluster, resources provided by each node, dependencies between functions in a workflow, requirements of functions and provide a set of

rules to co-allocate the functions in a given cluster. Another challenge for this component is to process this information in a timely manner and keep an updated representation of the information.

Figure 26 shows the different components of the Co-allocation Strategies component. There are processes that run periodically (in grey in the figure) to obtain information about the status of the cluster or the execution of the functions, those components are: Cluster information, Cluster status, Function metrics and Function performance aggregator. The Co-allocation database (DB) stores all the information gathered by the periodic process. Last, the data collector and the rule generator components (in orange in the figure) are activated by the Coallocation webhook component provided by the Resource Management Controllers.

The pod YAML file intercepted by the Co-allocation webhook is sent to the data collector component. This component is in charge of analysing the requirements of the pod (function) to be deployed in terms of resources, or regarding co-allocation with other functions in the same flow. This information is in the WorkflowCRD file obtained through the Kubernetes API. The current status of the workflow in terms of functions running at each node and its impact in latency regarding previous executions is read from the Co-allocation database. Then, the rules generator component defines pod or node, affinity or anti-affinity rules, modifies the pod YAML file and sends the file back to the Co-allocation webhook component.

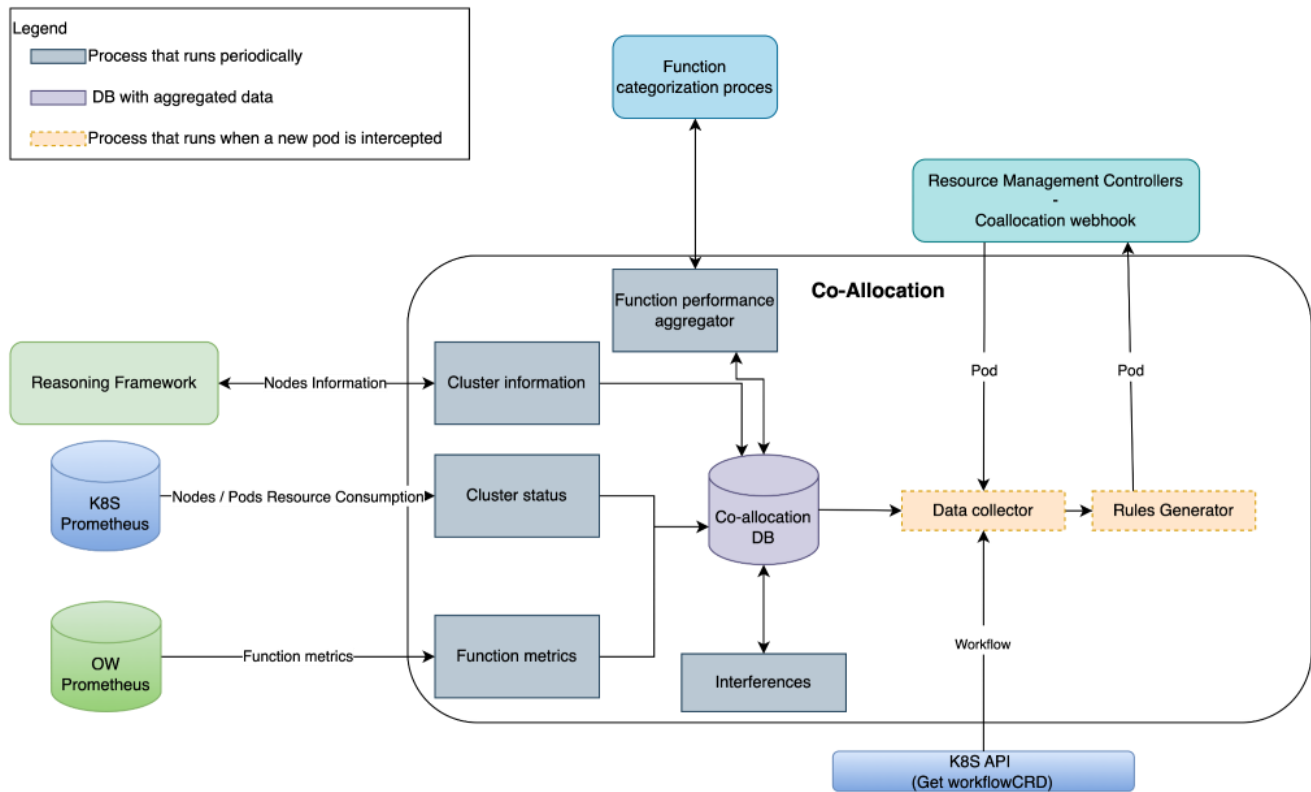


Figure 26 Co-allocation Strategies component internal architecture

Inputs

- Application constraints/preferences provided from the Visual Workflow and the Application Semantic Model components. Both the workflow, and its constraints and requirements will be considered to co-allocate functions. The dependencies between them in a YAML-based representation.

- Computations resources requirements defined by the Service Semantic Model component. The semantics will indicate CPU, memory, network consumption requirements of the different functions to deploy.
- Cluster architecture will provide the architecture of the PHYSICS cluster from the Semantics Model, the Global Continuum Placement and the Resource Management components. The placement of the various application components in the different cloud services available in the Physics cluster.
- Resource consumption statistics. The metrics are stored in Prometheus. These metrics include the CPU, memory, execution time, other co-allocated functions and network usage of the different functions and nodes available in the cluster.

Outputs

- Affinity and anti-affinity rules on the pod of a function.

3. PHYSICS COMPONENTS INTERACTIONS

3.1 Application Development Environment (WP3)

Figure 27 describes the interactions between the components of the visual design environment and external ones in PHYSICS and beyond. In this figure only the direct foreseen communications between components are displayed. Cases of indirect communication (e.g. the consumption of the annotations inserted in the application workflow during the design phase by elements in work packages WP4 and WP5) are not portrayed. Based on the figure the interactions are the following:

1. All envisioned areas/tasks that will offer some functionality, i.e. the Distributed Memory Service, the Elasticity Controllers etc. need to provide Node-RED nodes that will be embedded in the Node-RED editor used by the Design Environment. Through these client nodes, the developer will be able to utilize the interfaces of these components or embed the implemented functionalities (in the case of the patterns). Furthermore, means of inserting annotations in the created graph should be provided (in the form of descriptor nodes or in-code annotations), in order for these to be either used locally (in WP3) or forwarded by attachment in the application graph in order to be utilized downstream (for placement or management decisions)
2. Implementations of patterns may include the existence and/or usage of external services (such as Cloud storage, notifications, supporting micro services etc.). These implementations will also need to be embedded in the Design environment, either as sub flows or as supporting services through relevant descriptors.
3. The developer utilizes the Design Environment in order to create and annotate the application graph, exploiting the aforementioned client nodes and describing the application logic. They may also use the according tabs and functionality in order to test the application locally. Once they are ready, they will trigger the deployment process of the next step.
4. Once the application graph, including functions as well as micro services, has been finalized, it will be forwarded to the Global Continuum Placement component. The latter will decide on their placement and forward the decision to the Platform Orchestrator in order to be enacted. The Orchestrator will initialize calls for the various deployment artefacts (e.g., calls to handle the function registration process in the target OpenWhisk instances). However, given that different Openwhisk platform instances may be run and operated in different locations, it is not necessary that there is a single, high level Openwhisk platform that spans across different container platforms. Therefore, these OpenWhisk platforms need to be handled like separate instances. Then the Orchestrator needs to either act as a proxy, forwarding requests to external OpenWhisk instances or it needs to populate the environment of the action containers with suitable environment variables through which the action nodes will dynamically retrieve the relevant openwhisk instance details (e.g. endpoint, credentials etc) for that action invocation. In any case, WP3 provides a relevant Invoker node that can be dynamically configured through environment variables for the target OpenWhisk endpoint.

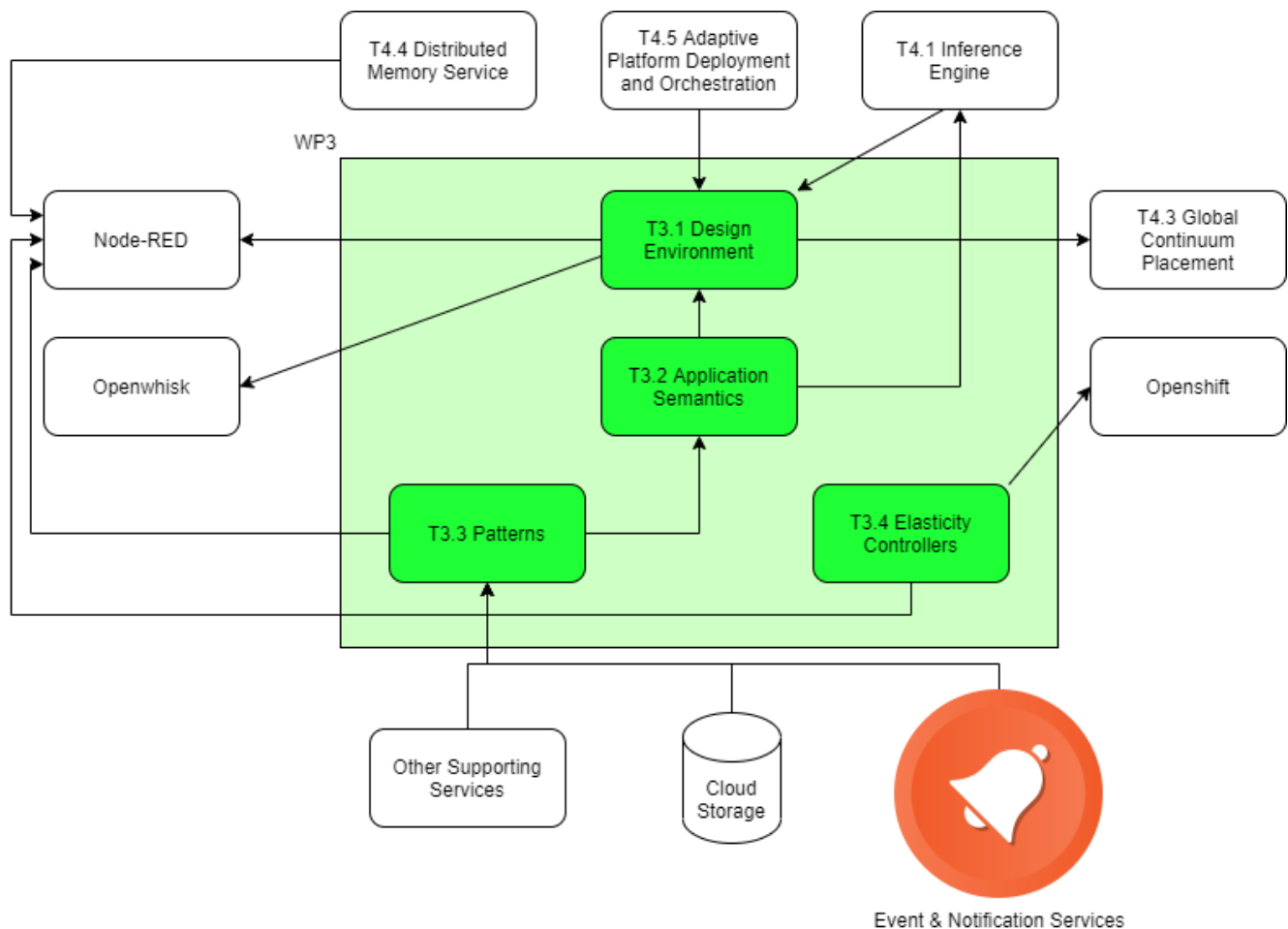


Figure 27 WP3 internal and external interactions

More details and the complete interactions between WP3 during the design and deployment process appear in the PHYSICS Global View (Chapter 5 of this document).

3.2 Continuum Deployment Layer (WP4)

The Continuum Deployment Layer consists of the different interrelation between WP4 components allowing the deployment of applications in different managed/remote clusters, based on the user annotations and application description from work package WP3. Figure 28 shows the interaction among WP4 components and how the deployment takes place within the PHYSICS platform.

Based on the previous figure, the interactions between the components are the following:

1. The Reasoning Framework will collect and process the input from the Design environment component. This Component will aggregate all the semantics into the application description together with the rest of the detailed configurations for the initialization. All the dependencies required for the application functionality will be included and sent to the next component, the Global Continuum Placement.
2. The Global Continuum Placement component will aggregate the placement information of the application components to the underlying infrastructure. To achieve this, Global Continuum Placement subcomponents will obtain all the information required from the infrastructure and take a decision based on function annotation. The Application Graph sent by the Reasoning Framework will get the

location decision to build the Deployment Graph and then sent to the Adaptive Platform Deployment, Operation & Orchestration component.

3. The Adaptive Platform Deployment, Operation & Orchestration component will collect the Deployment Graph and will process this information in order to parse the information to the cluster software infrastructure. If there is some infrastructure demand according to the Deployment Graph that is not in place it will request it through the Resource Management component and get it ready before the application is deployed.

4. The Performance Evaluation Component will consume the performance metrics data supplied by the Monitoring System of each managed cluster. These data are kept and analysed, used also for the creation of relevant performance models. Thus, it can and will consult other components such as the Global Continuum Placement, the Scheduling Algorithms, and the Co-allocation Strategies components regarding the anticipated performance of strategies.”

Figure 28 illustrates the flow of information between PHYSICS components to deploy an application (workflow) that is made from independently generated functions. It shows the connections between the initial workflow creation and the different components required for the registration (Reasoning Framework, Continuum Placement and Orchestrator) and the feedback to the user interface.

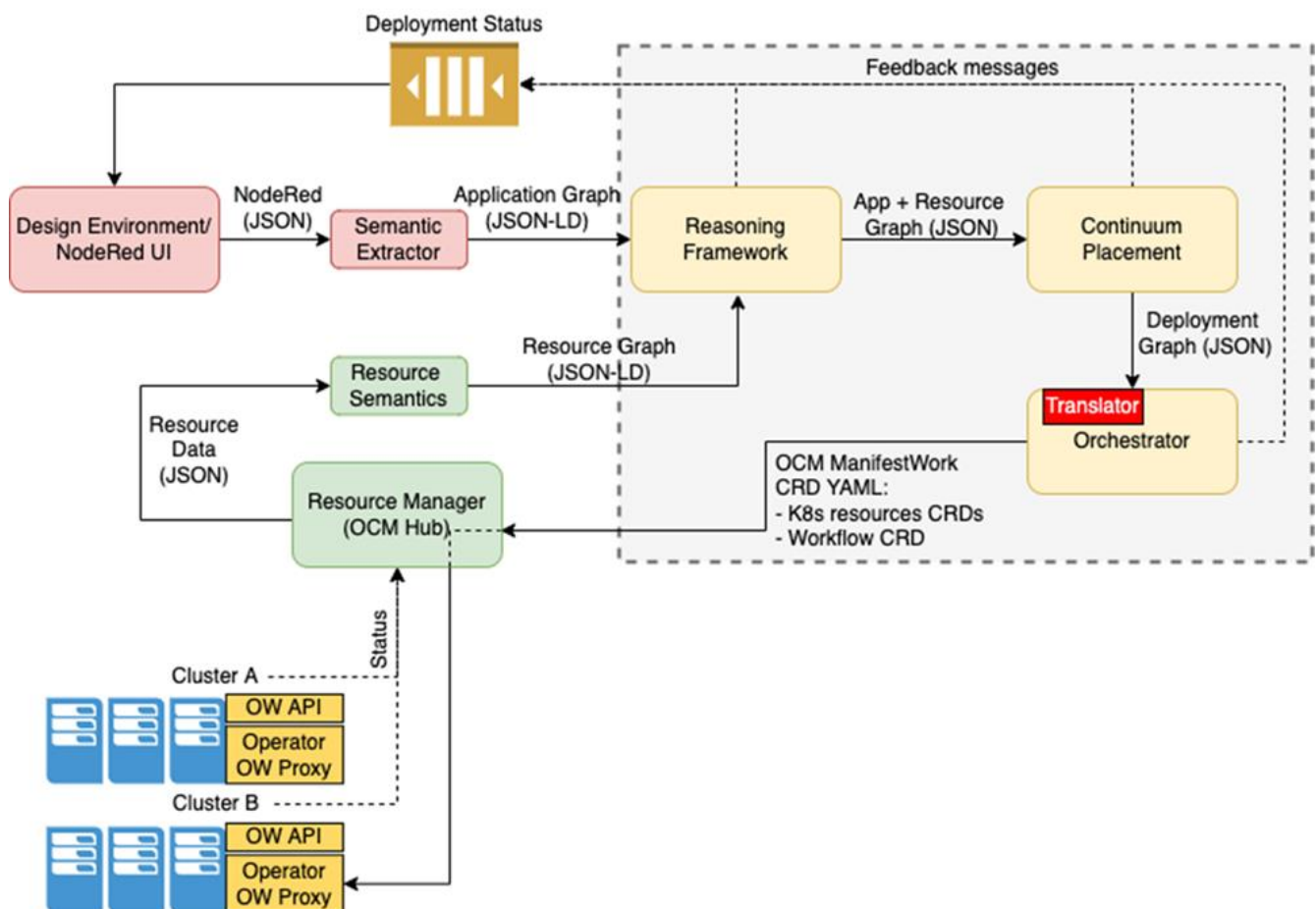


Figure 28 PHYSICS Deployment Pipeline

WP4 in the PHYSICS platform consists of different components in charge of the deployment of the application allowing the platform to run remote functions as flows (workflows) in a graph (application).

DESIGN ENVIRONMENT (DE)

This component is part of WP3, but it is the starting point for a new deployment. The user has a set of parts available in the Node-RED interface to interact with an application graph: the annotations, the endpoint definition, and the flow for the function invocation. The creation of a graph/app starts by dragging the flows/functions. Each of these may be deployed in a different cluster based on the user annotations.

SEMANTIC EXTRACTOR

The Semantic Extractor service receives the flows description from the Design Environment, including information on container image location (package format for the business logic of the function), and extracts information based on the included annotations as well as other fields in the Node-RED JSON object description.

The mapping is performed against the fields defined in the application ontology, creating, and forwarding the application graph to the Reasoning Framework.

REASONING FRAMEWORK (RF)

The Reasoning Framework acts when the user deploys the application and forwards the necessary information for the placement by creating the Application graph. All that necessary information involving both the application and the resource descriptions have been previously received from WP3 and WP4 components and stored in the Reasoning Framework knowledge base.

The Reasoning Framework API offers all the required REST endpoints for querying specific information from the knowledge base. For example, the Design Environment requires retrieving relevant information during the application graph specification. On the other hand, the clusters API endpoint will return specific information of the registered clusters, needed by the Global Continuum Placement component. The use of the RF user interface allows the interpretation of the available data as graphs and understand what happens behind the scenes.

Every registered cluster has properties like name, memory, locality, and performance scores to be processed by the Global Continuum Placement component. Properties such as memory and optimization goal, will be defined by the developer in the Design Environment and used to match each application graph with the available clusters. The "allocatable" property between flows and clusters is inferred when a new application graph is stored in the knowledge base indicating the possible targets for a specified flow.

GLOBAL CONTINUUM PLACEMENT

The Global Continuum Placement represents the decision maker for the final deployment of the different flows of an application graph in the target clusters. Based on a set of annotations given by the user and the information processed by the Reasoning Framework, an algorithm decides which cluster should receive a flow. The algorithm will take that decision based on a set of values for each function requirement to result in the optimization of the overall workflow.

ADAPTIVE PLATFORM DEPLOYMENT, OPERATION & ORCHESTRATION

The Translator service of the Orchestration component collects the information forwarded by the Global Continuum Placement component and maps the fields supplied in a complete deployment graph to a Resource Manager API domain specific language.

The ManifestWork message payload connects with Resource Management Controllers. The ManifestWork message is received by the Resource Manager Controllers of the target cluster, implemented with Open Cluster Management API (OCM), based on the namespace supplied. It will deploy a workflow object to each different target/managed cluster available to the PHYSICS platform.

The OpenWhisk proxy service of the Orchestration component will translate the workflow specification from the PHYSICS Workflow CRD (Custom Resource Definition), the domain specific language of PHYSICS Workflow entity, into the FaaS platform API (OpenWhisk FaaS API) to deploy/register the functions in the FaaS platform.

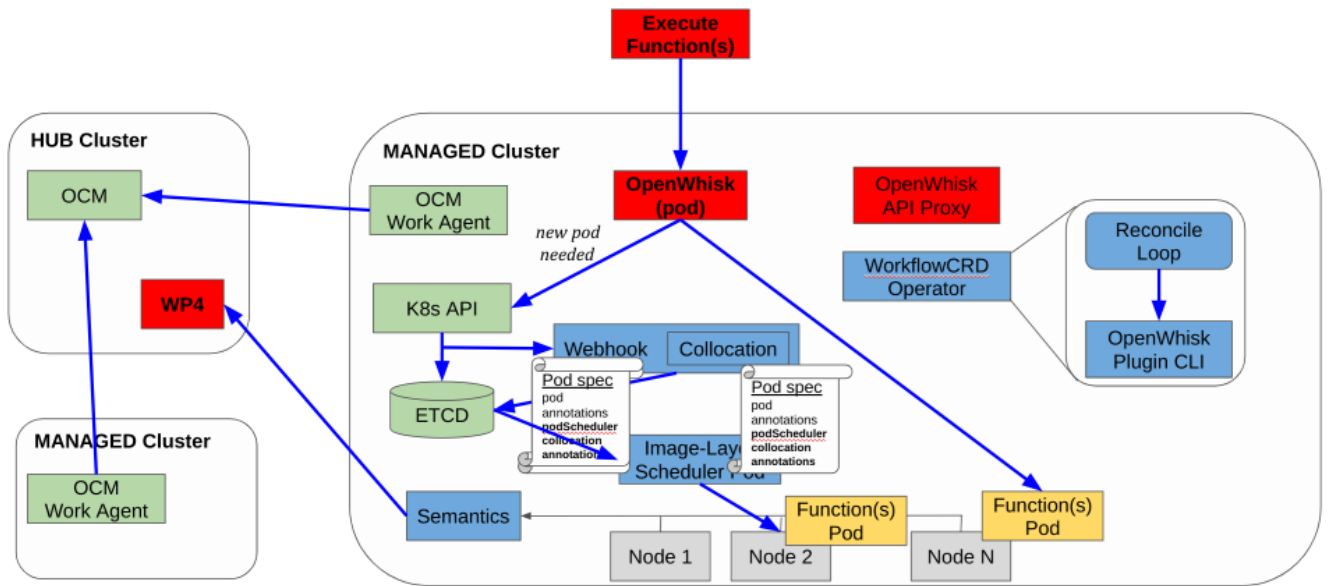


Figure 30 WP5 internal interactions for Function Execution

The interactions between the components are the next:

1. When the OpenWhisk API is called to execute a function 2 options are possible:
 - a. There is already a suitable pod (i.e., warm/hot container) ready to execute them. In that case there is nothing extra to be done
 - b. There is a need for creating a new pod for executing that function and the Kubernetes API is called for that
2. In the second case, once the Kubernetes API receives the call, but before storing it on the ETCD database, the Webhook gets invoked
 - a. First the scheduler related annotations are used, and the pod spec gets updated with the right scheduler (e.g., the image layer one) to be used for that pod/function.
 - b. Then, the co-allocation engine gets executed and obtains the right affinities/anti-affinities by checking the information about the function (in the WorkflowCRD), about the cluster (Prometheus) and about previous executions.
 - c. The output of the co-allocation component is applied to the pod spec as a set of affinity and anti-affinity rules.
 - d. The webhook execution finalises and the modified pod object is returned and stored in the ETCD database
3. At this point the normal process from Kubernetes happens. In this case the *image-layer locality scheduler* (running as a pod) detects there is a new pod associated with it that has no node selected and starts the process to obtain the best node, fulfilling the co-allocation hints and considering the node with more layers in common for the given pod container image.
4. After the node is selected the node is in charge of creating the pod locally as in the normal Kubernetes process.

4. PHYSICS GLOBAL VIEW

Figure 31 presents the steps and components of the PHYSICS framework platform involved in the design, deployment, and execution of a function. The function 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.

The starting point is the description of the application to be deployed including the other necessary options and annotations of the developer, while alerting the latter about the status in each step.

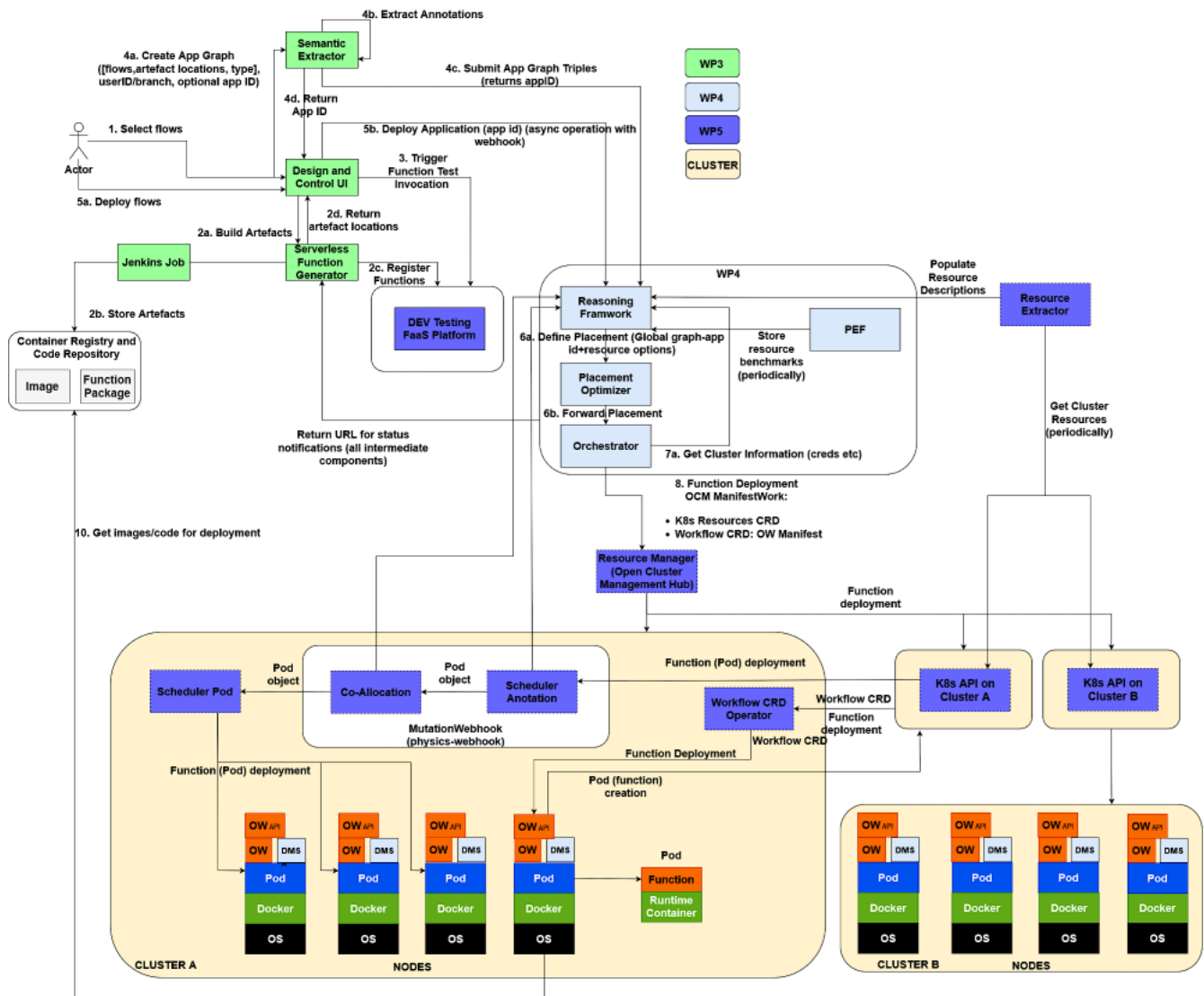


Figure 31 Design, deployment, and execution of a function

The design process can be summarized as follows:

1. The user uses the Design and Control User Interface (UI) to:
 - a. Create flows and functions. Local testing of these inside Node-RED can filter out common minor errors that can take up much time if each time an error is fixed, the function is deployed in the formal testing or production environment
 - b. Add annotations at the function or flow level for desired aspects (e.g. resource selection aspects, deployment options, QoS features etc.)
 - c. Select flows made out of functions (actions) to be included the application
2. Function generations are generated:
 - a. Building the corresponding artifacts (Serverless Function Generator)
 - b. Storing the artifacts (Docker image) in the Container registry
 - c. The functions can be registered in a local deployment of the FaaS platform for testing purposes. This aids in further reducing the time needed for testing, since the optimization processes and selection are skipped
 - d. The artifact location is returned to the design environment in order to be locally deployed and invoked
3. The function execution in the local deployment of the FaaS platform is triggered through the Control UI.
4. Deployment after testing: Once testing is done, the developer wants to deploy the application on the production environment.
 - a. The Semantic Extractor receives the request for creating the application graph
 - b. The Semantic Extractor extracts the annotations of functions
 - c. The Semantic Extractor generates a graph in the form of triples which is sent to the Reasoning Framework, which returns the application id.
 - d. The app id is returned to the UI

The process of deployment and execution of functions can be summarized as:

5. The user decides to deploy the application (deploy flows)
6. The Reasoning Framework is invoked.
 - a. It enriches the app graph with candidate resources
 - b. The Placement Optimizer selects the most suitable resources for the deployment. This information is sent to the Orchestrator
7. Function deployment
 - a. The orchestrator generates the OCM ManifestWork CRD YAML. with K8S resources CRDs and workflow CR
 - b. The Orchestrator sends this information to the Resource Manager (OCM Hub)
 - c. Each cluster receives this information through the K8s API on Cluster A

5. PHYSICS DEVELOPMENT AND DEPLOYMENT STRATEGIES

In alignment with the general PHYSICS Reference Architecture (RA) approach and to facilitate the PHYSICS framework development and deployment phases, 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 Kubernetes¹³ cluster.

This section contains an overview of the previously mentioned strategies, further details are provided in the deliverable D6.1 – “Prototype of the Integrated PHYSICS solution framework and RAMP V1”, any changes on this deliverable will be reported on D6.2.

5.1 Development Strategy

The PHYSICS RA design approach considers 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. In order to support the development and testing activities, a CI/CD approach leveraging DevOps methodologies are used. The CI/CD stands for the combined practices of Continuous Integration (CI) and Continuous Delivery (CD).

- *Continuous Integration* is a practice where development teams frequently commit (many times per day) application code changes to a shared repository. These changes automatically trigger new builds that are then validated by automated testing to ensure that they do not break any functionality.
- *Continuous Delivery* is an extension of the CI process. It is the automation of the release process so that new code is deployed to target environments, typically to test environments, in a repeatable and automated fashion.

The CI/CD processes are implemented in a blueprint reference testbed environment. The Continuous Integration tools are deployed on Kubernetes: it is an ideal choice for a Continuous Integration environment, since it allows easy updates of deployments when new application images are built, with manifests containing deployment configurations versioned like Git server alongside the application source code. Furthermore, it is easy to generate new test environments from scratch, which enables future scenarios including automated end-to-end integration testing. Build agents are also created on demand and removed when done, providing efficient resource utilization and clean environments to ensure build reproducibility.

On the target Kubernetes cluster, a namespace named *devops* has been created for hosting the DevOps tools, which are:

- **Gogs**¹⁴ is a simple, stable and extensible self-hosted Git service that lets each developer team collaborate on PHYSICS source code.

¹³ Kubernetes (<https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>)

¹⁴ Gogs (<https://gogs.io>)

- **Jenkins**¹⁵ is the de-facto standard open-source automation server for orchestrating CI/CD workflows.
- **Harbor**¹⁶ is a popular Docker registry CNCF compliant.
- **OpenLDAP**¹⁷ is used as the single user directory for all tools, centralizing authentication and simplifying management of developer accounts.
- **Helm**¹⁸ is a package manager that streamlines installing and managing Kubernetes applications.

Figure 32 shows a workflow describing how CI/CD works for a specific partner (e.g., Partner “A”). When a developer pushes new component code, Gogs (i.e. Gitlab in the picture) invokes a webhook on Jenkins, which starts any job affected by the code changes. The job builds the component, runs unit tests and, if everything has worked in a proper way, builds an updated Docker image and pushes it to Harbor. The following step is deploying the updated component in the specific partner namespace; in fact, we have as many namespaces as the partners in order to maintain the correct isolation between all PHYSICS partners. In order to deploy the component, where possible, Helm manager is used. At the end of the process, Jenkins sends a notification to a dedicated CI/CD channel on the PHYSICS *Slack*¹⁹ 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.

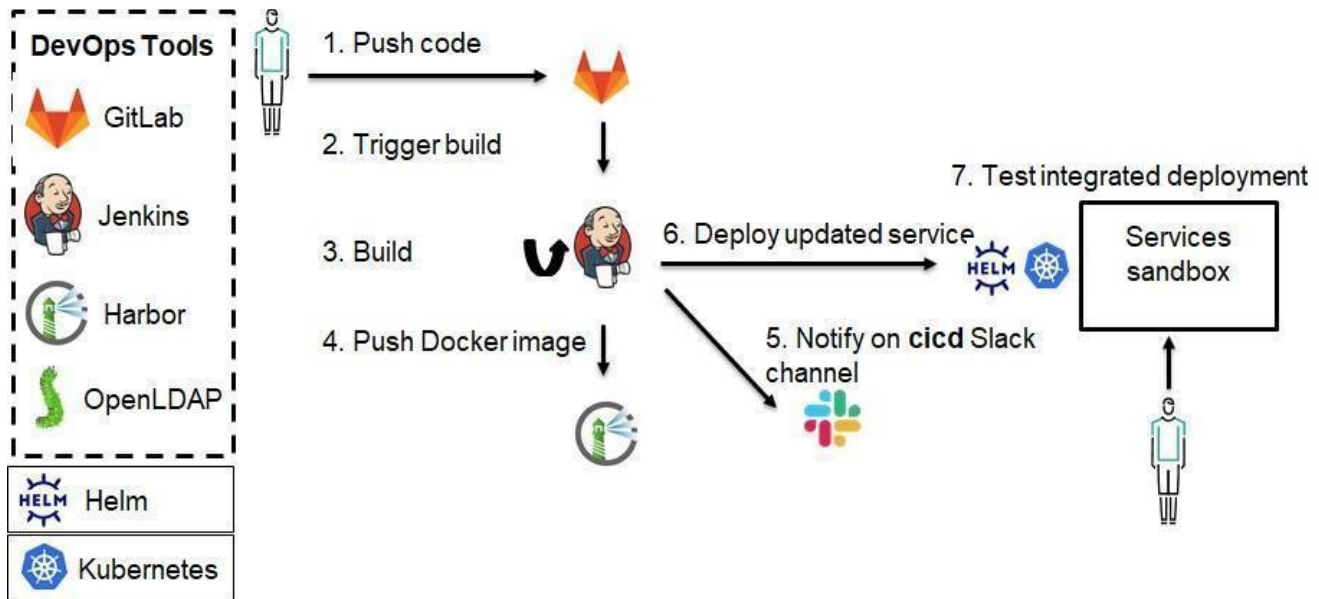


Figure 32 CI/CD workflow example

5.2 Deployment Strategy

The PHYSICS RA design approach provides to use the deployment strategy for the creation of a PHYSICS blueprint reference on a public cloud provider in order to have an easy reachable environment by anyone, with the possibility to scale on demand and to get the possibility of using Kubernetes as a managed service.

¹⁵ Jenkins (<https://www.jenkins.io/doc/>)

¹⁶ Harbor (<https://goharbor.io/docs/2.3.0/install-config/>)

¹⁷ OpenLDAP (<https://www.openldap.org/doc/admin25/>)

¹⁸ Helm (<https://helm.sh/docs/intro/>)

¹⁹ Slack (<https://slack.com/intl/en-pt/features>)

Kubernetes is the best choice being PHYSICS planned to be a framework based on microservices running into containers, so that an orchestrator is necessary. Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications, but it provides more features such as:

- **Service discovery and load balancing**
Kubernetes automatically routes the traffic to the pod creating a service assigned to it. This resolves the problem of knowing the pod IP, because the pod could die at any moment so its IP could change many times, and this would make it difficult to communicate with it.
- **Storage orchestration**
Kubernetes manages the storage for Stateful pods, the user has only to decide where the storage is located after that the Kubernetes automatically mounts and manages the storage consumption by the pod.
- **Automated rollouts and rollbacks**
Kubernetes allows the deployment of a new application without downtime, acting on the replicas that make up that application data.
For example, an application consisting of 2 pods with version 1.0 when the user decides to deploy version 2.0, Kubernetes will first create a new pod with version 2.0 when this is ready it will delete the pod with version 1.0, then it will create a second pod with version 2.0 and once active it will delete the last pod with version 1.0.
At the same time Kubernetes will keep track of this new release and any rollback can be done easily with a single command by recalling the previous release.
- **Resource Manager**
Kubernetes has an internal mechanism to manage in a fine-grained way the allocation of resources (RAM, CPU and Storage) to a specific pod, application or tenant.
- **Self-healing**
Kubernetes independently manages the health of the applications; the user only has to set how many pods a given application must be composed of and in case of a malfunction in one of them it will be solved by Kubernetes through the cancellation and creation of a new pod.

Moreover, Kubernetes gives the possibility to implement isolated resources accessible only by specific other resources or people. This functionality is very important during the development phase because it provides to all partners the benefit to have their own sandboxes in which to develop and test their components. In order to implement the sandboxes, we used two Kubernetes' concepts:

- **Namespace:** They are a logical grouping of a set of Kubernetes objects to which it is possible to apply some policies, in particular:
Quote sets the limits on how many hardware resources can be consumed by all objects
Network defines if the namespace can be accessed or can access to other namespace, in other word if the namespace is isolated or accessible
- **POD:** is the simplest unit in the Kubernetes object. A Pod encapsulates one container, but in some cases (when the application is complex) a POD can encapsulate more than one container. Each POD has its own storage resources, a unique network IP, access port and options related to how the container/s should run.

The deployment strategy makes use of IaC (Infrastructure as Code) tool like *Terraform*²⁰ to easily recreate on demand the blueprint environment. *Terraform* was selected because it is one of the best tools for IaC available on the market, which allows to recreate an infrastructure everywhere always in a predictable and

²⁰ Terraform (<https://www.terraform.io/intro/index.html>)

safe way; moreover, it is an open-source software with a very large community, and it is infrastructure agnostic.

Figure 33 presents the flow used to deploy PHYSICS components. This flow is made out of two macro phases, in the first phase the Terraform scripts are retrieved from PHYSICS general GIT repository; those scripts are used to create the environment that will accommodate the PHYSICS components in any location both cloud and edge. In the second phase the HELM charts are used to install the and configure the components into the environments created by Terraform. The only prerequisite that the customer, that is going to deploy PHYSICS, needs are the Terraform and HELM client.

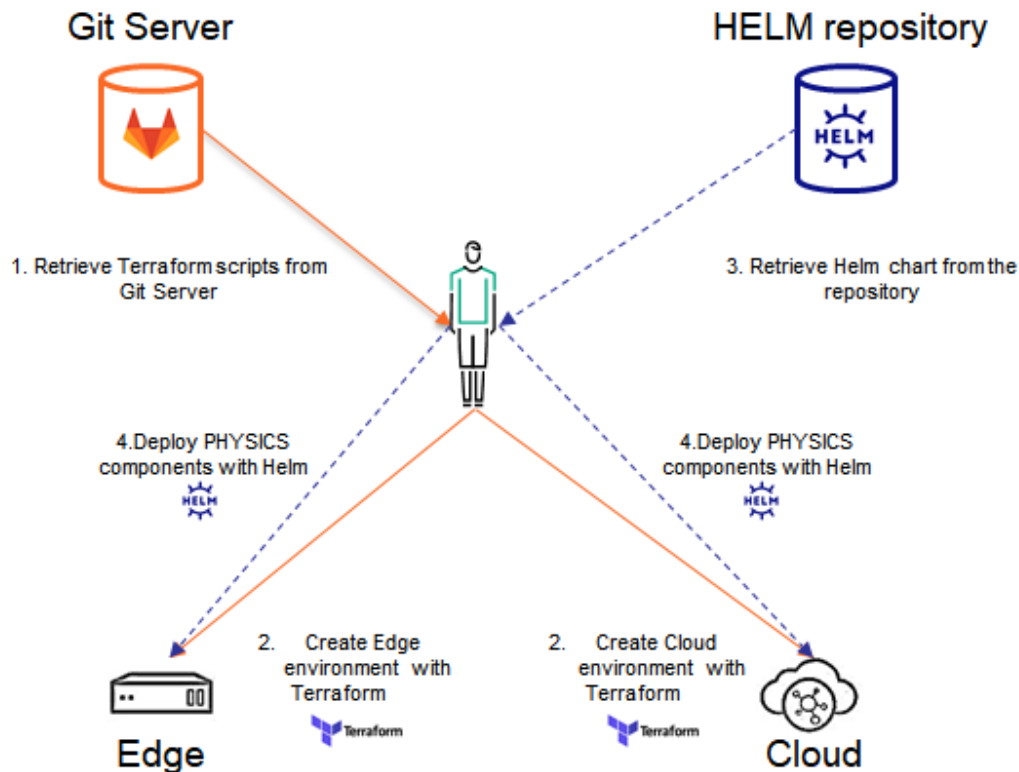


Figure 33 PHYSICS components deployment flow

6. CONCLUSIONS

This deliverable presents the second and last version of the PHYSICS Reference Architecture. This deliverable is an updated version of the PHYSICS Reference Architecture described in deliverable D2.4 PHYSICS Reference Architecture Specification V1 delivered in month 7 of the project. These updates include a more detailed description of the component's functionality, internal architecture and interfaces (input and outputs) and the information flow among components. These updates are derived from the implementation and integration of the components in the first integrated PHYSICS framework due in month 15 and its evaluation by the pilots in month 18. These activities triggered also the update of the PHYSICS requirements in month 19.

The architecture presents the functional description of the components (functional view of the architecture). Describing the challenges each component will face during the second phase of the development of the component, the input and output of each component. The interactions among components are also described. The deliverable also presents the dynamic aspects of the PHYSICS architecture by describing the interactions between components and the information flow from the creation of an application in the Design Environment to its registration and execution in one or more clusters. The deliverable also presents the process for the CI/CD during PHYSICS development and also the deployment process of the PHYSICS platform itself.

This deliverable will guide the second phase of the design and development of the three layers of PHYSICS, which will be documented in upcoming deliverables in work packages WP3, W4 and WP5, respectively.

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).