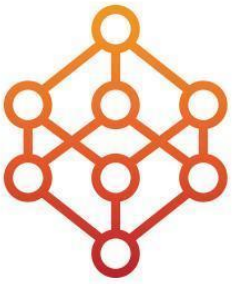

PHYSICS

oPTIMIZED HYBRID SPACE-TIME SERVICE CONTINUUM IN FAAS

D2.4 – PHYSICS REFERENCE ARCHITECTURE SPECIFICATION V1

Lead Beneficiary	UPM
Work Package Ref.	WP2 – Reference Work package Name
Task Ref.	T2.3 – Reference Architecture Specification
Deliverable Title	D2.4 – Reference Architecture Specification V1
Due Date	2021-07-31
Delivered Date	2021-08-02
Revision Number	3.0
Dissemination Level	Public (PU)
Type	Report (R)
Document Status	Release
Review Status	Internally Reviewed and Quality Assurance Reviewed
Document Acceptance	WP Leader Accepted and Coordinator Accepted
EC Project Officer	Mr. Stefano Foglietta

H2020 ICT 40 2020 Research and Innovation Action

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

CONTRIBUTING PARTNERS

Partner Acronym	Role[1]	Name Surname[2]
UPM	Task leader	Marta Patiño, Ainhoa Azqueta, Luis Mengual, Tonghong Li
HUA	Contributor	George Kousiouris, Stylianos Tsarsitalidis, Evangelos Boutas, Teta Stamati, Chris Giannakos
RHT	Contributor	Josh Salomon, Luis Tomas
HPE	Contributor	Alessandro Mamelli, Domenico Costantino
RYAX	Contributor	Yiannis Georgiou
ATOS	Contributor	Lucas Pelegrin
BYTE	Contributor	Yannis Poulakis
INNOV	Contributor	George Fatouros
RHT	Reviewer	Luis Tomás
HUA	Reviewer	George Kouiouris
INNOV	Quality Assurance	John Soldatos, Marios Touloupou
GFT	Revision	Vittorio Monferrino

REVISION HISTORY

Version	Date	Partner(s)	Description
0.1	2021-04-30	UPM	ToC Version
0.2	2021-05-15	UPM	Initial architectural overview definition
0.3	2021-05-31	UPM. WP3-4-5 leaders	1st round of Software components description
0.4	2021-06-10	UPM. WP3-4-5 leaders	Feedback and interactions on 1st version of components
0.5	2021-06-25	UPM. WP3-4-5 leaders	2nd round of Software components description
0.6	2021-07-10	UPM. WP3-4-5 leaders	Contributions on interactions across WPs
0.4	2021-07-20	HPE	Development and integration section
1.0	2021-07-20	UPM	First version
2.0	2021-07-29	RHT	Review
2.1	2021-07-30	HUA	Review
2.2	2021-07-30	INNOV	John Soldatos
3.0	2021-08-02	UPM	Version for Submission

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

LIST OF ABBREVIATIONS

FaaS	Function as a Service
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
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
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 first version of the architecture foreseen in the project. This document will be a live document that will be enriched as the project progresses till the second and last version of the architecture is delivered in month 23. 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 levels 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. The architecture is described using a functional view in which 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. As already outlines, future releases of the architecture will incorporate feedback from the updated design of the various components, as well as for the actual use of the first version of the architecture in integration and use case activities.

TABLE OF CONTENTS

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 or environments are summarized in Figure 1

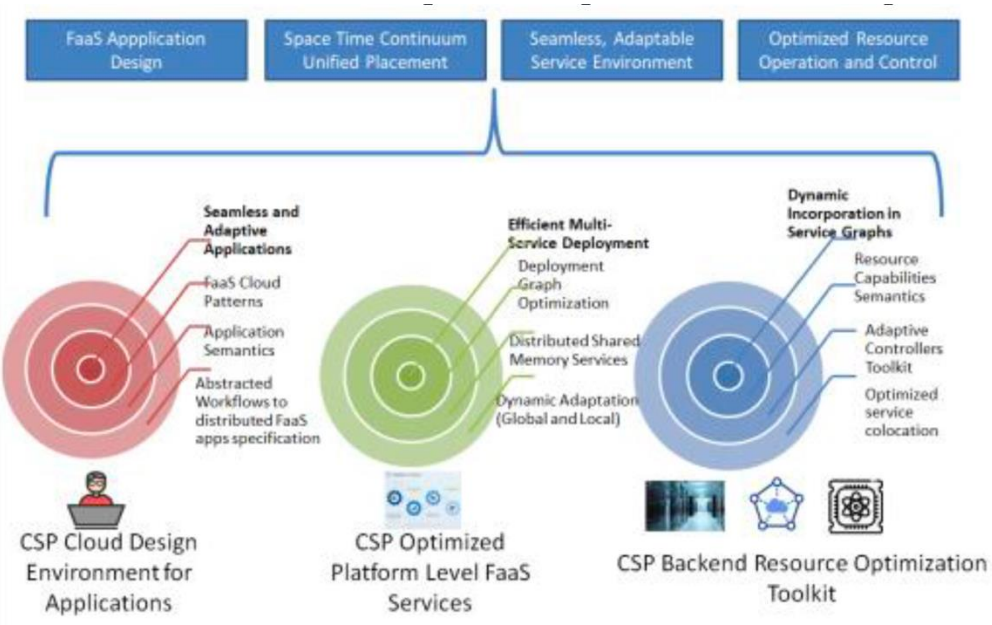

Figure 1 - PHYSICS design environment and toolkits

In order to achieve these goals 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 WP6 Exploitation, Dissemination and Impact Creation. Work packages W3, WP4 and WP5 (technical work packages) will be 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 first version of the PHYSICS architecture.

1.1 Objectives of the Deliverable

The goal of this deliverable is to define the initial version of the architecture of the PHYSICS project. The architecture is defined as a set of views, namely functional view, information view and deployment view[3] 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) and well as for the design of pilots (WP6 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 first version of the PHYSICS Architecture concluding Phase 1 of the project. The PHYSICS architecture will be updated as the project progresses. This deliverable can be considered a live document that will be enriched as the technical work packages progress and the software components are implemented and integrated (Phase 2). The architecture will also reflect the feedback received from the use cases after the end of the first iteration of the project in month 18. A final version of the deliverable with the updated architecture will be produced in month 23, at the end of the second year of the project (Phase 3), as shown in Figure 2

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

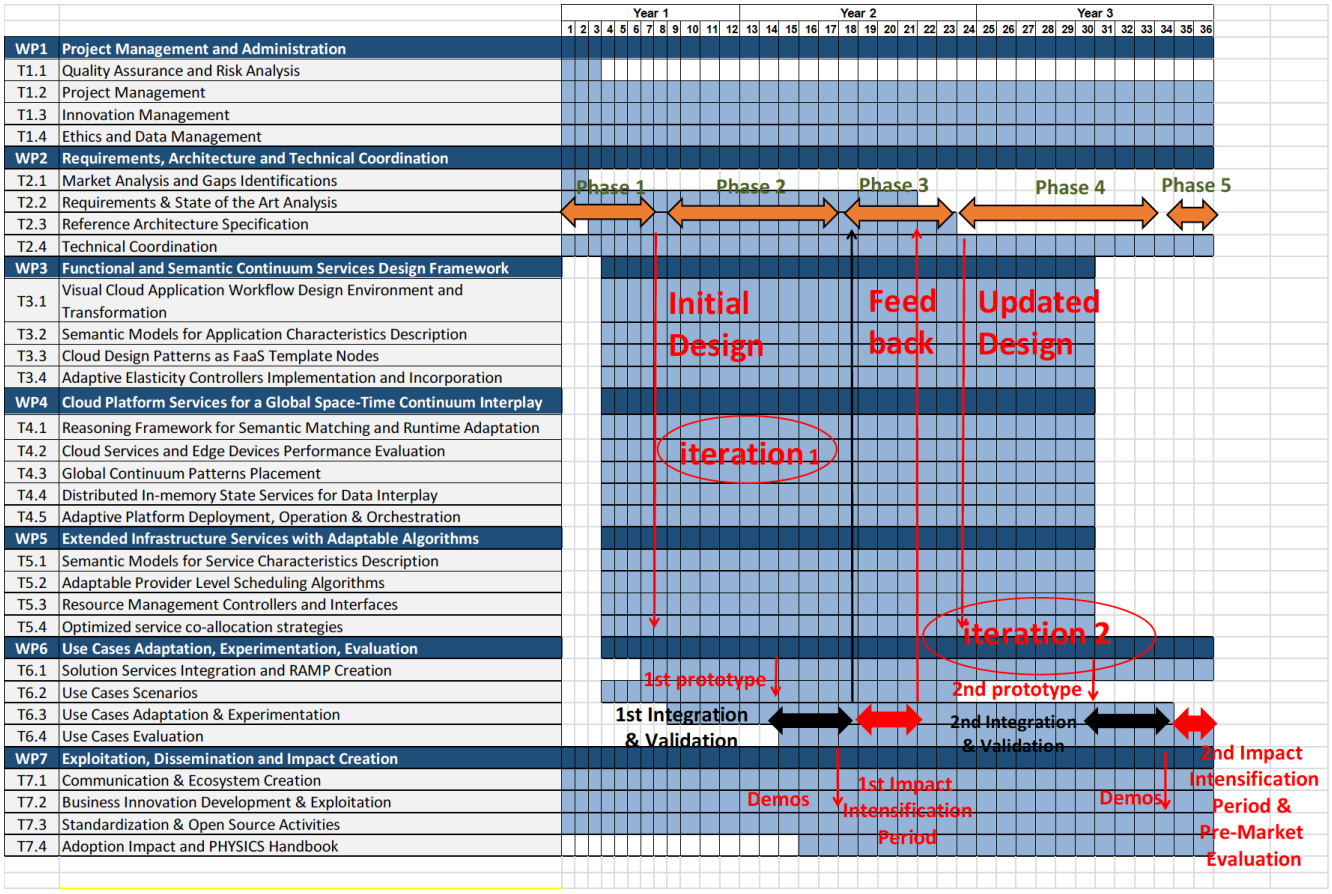

Figure 2 - PHYSICS planning

1.2 Insights from other Tasks and Deliverables

The Architecture of PHYSICS has been 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 Definition v1. The architecture also takes into consideration the input from the pilots defined in deliverable D6.3 Application Scenarios Definition v1. Although deliverable D6.3 and this deliverable are concurrent 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 (D2.4) 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 W3, WP4 and WP5, respectively, and the associated software prototypes, namely deliverables D3.1, D4.1 and D5.1. The integration of these prototypes will be documented in deliverables D6.1 and D6.5, while the first evaluation of the PHYSICS toolkits will be documented in deliverable D6.7, concluding the first phases of the project. Based on this feedback a second iteration of the project will be triggered in which the previous versions of all these deliverables will be updated (D2.5, D3.2, D4.2, D5.2, D6.2, D6.6 and D6.8).

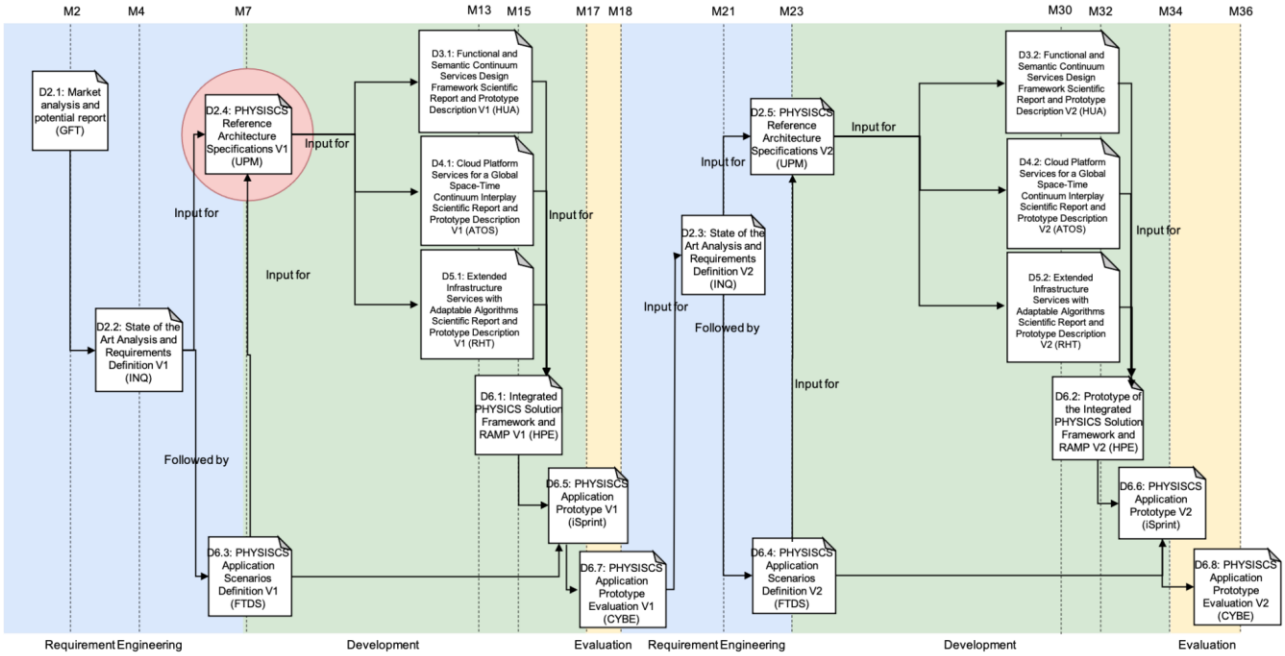

Figure 3 - PHYSICS deliverables dependencies

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. 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 PHYSICS development and deployment strategy. Conclusions are presented in the last section of the document.

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 development layer*, 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 work packages (W3, 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 the figure.

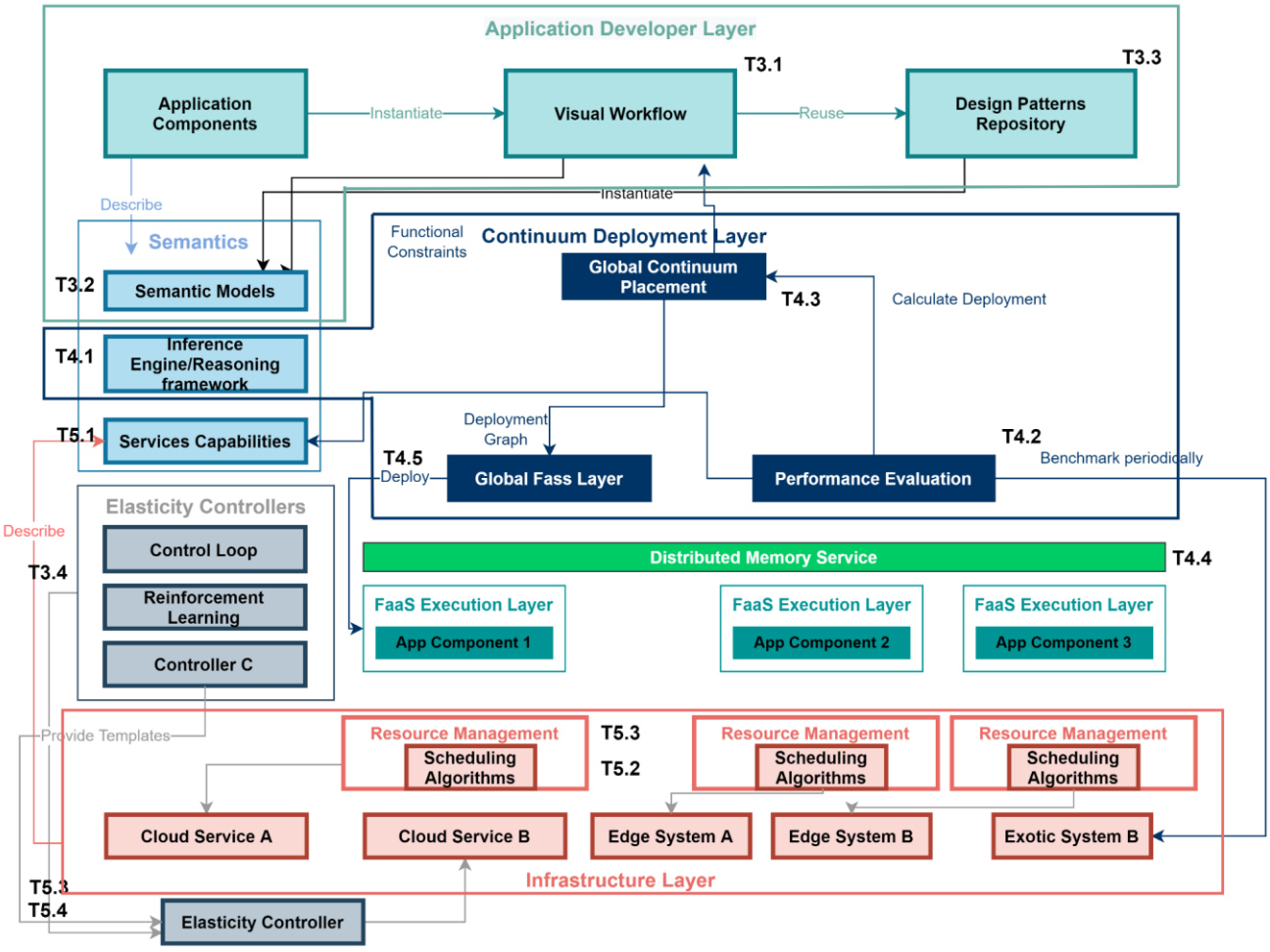

Figure 4 - PHYSICS Software Components

3. PHYSICS SOFTWARE COMPONENTS

3.1 Visual Workflow/Design Environment

Component Description

The PHYSICS Design environment 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 5, Figure 6 and Figure 7

Except for the main application creation operation, 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.

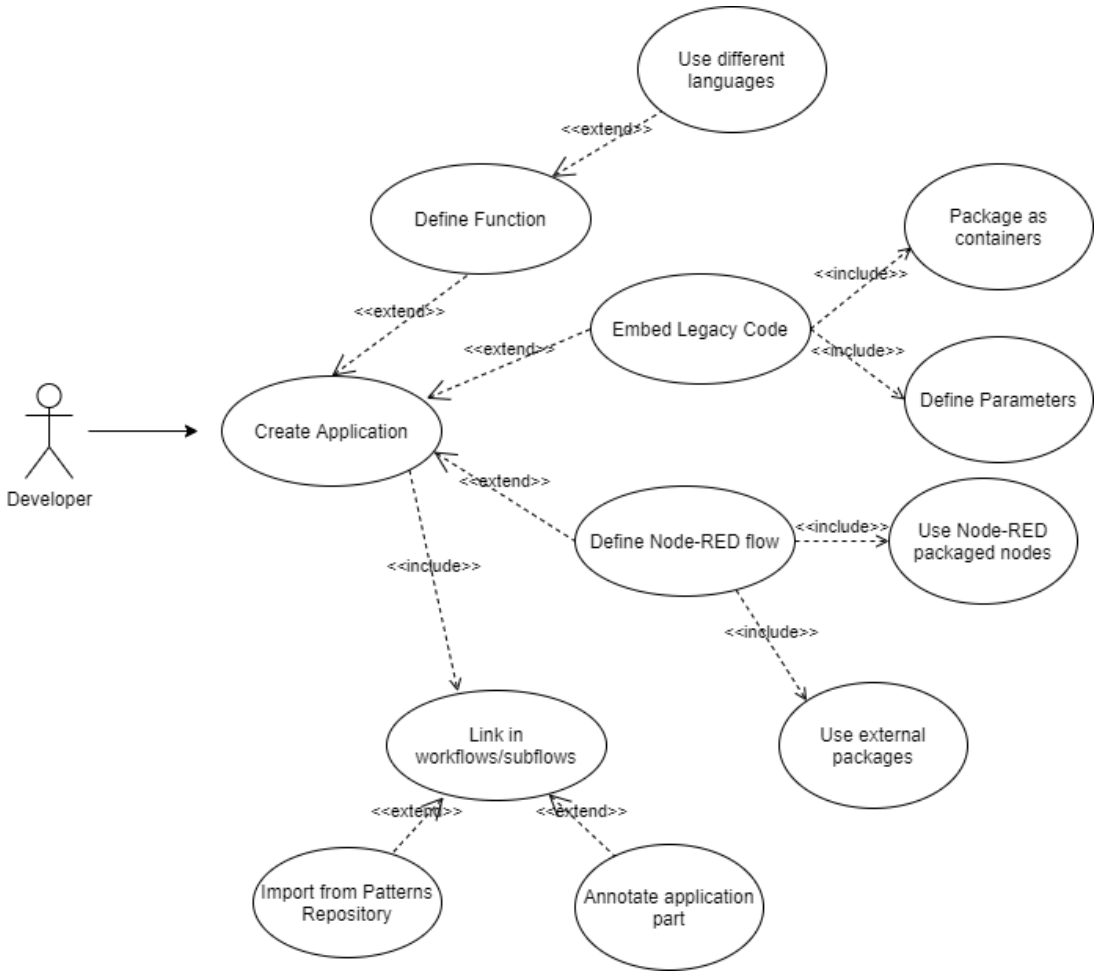

Figure 5 - Create Application Developer Use Case

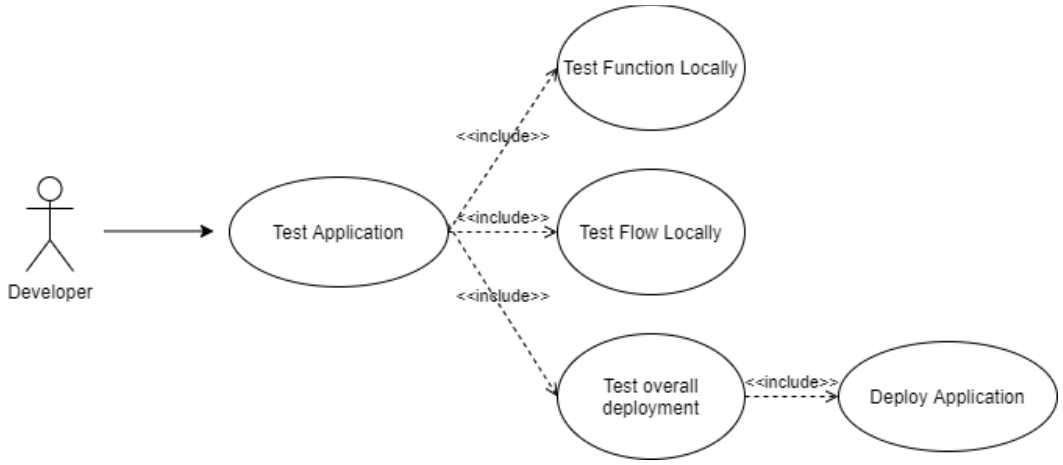

Figure 6 - Test Application Developer Use Case

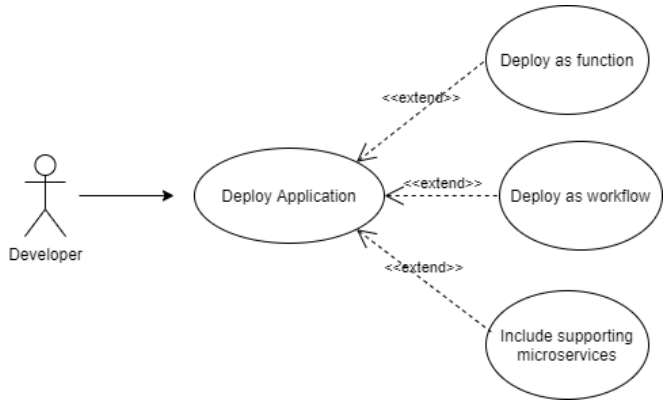

Figure 7 - Deploy Application Developer Use Case

The overall architectural diagram of the WP3 entry point for the developer appears in the following figure (Figure 8). This also includes interactions with the semantic block (Application Semantic Models (Section 3.2), Inference Engine (section 3.5)), the Design Patterns Repository (Section 3.3) and the FaaS Platform Deployment (Section 3.9).

The Design Environment will be a centralized UI App that will include and embed other elements/tabs offering the aforementioned functionalities. The central element is the Node-RED environment, 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 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. In order for the three modes to collaborate in the context of an application graph, the developer will need to exploit the environment in order to declare series of operations that are either orchestrated by the platform or are based on backend FaaS platform abilities such as function sequences, links between events and function executions etc. Testing of these operations and/or triggering of the according DevOps processes that are needed in order to build the respective deployable artefacts will be /supported by according UI tabs.

During this process, the created flows will be retrieved, and the overall necessary steps will be coordinated. As an example, functions developed within the environment may also need to be adapted/migrated to the FaaS platform runtime, through means of extracting their code and dependencies, creating the necessary image runtimes and declaring function sequences towards the FaaS platform (implemented by the Serverless Function Generator (SFG) subcomponent). Alternatively, specialized interaction nodes with the FaaS platform may be available in the palette, that will handle retrieval of available functions, editing, registration and invoking.

For the external components' incorporation, specific flows and tabs will be needed through which a developer may declare external dockerized components that need to be registered and wrapped like functions in order to be executed by the FaaS platform. This is based on the ability of the latter to execute any kind of dockerized image as the result of an event or part of a workflow as long as a specification for creating the image is followed. Thus, the wrapping abilities of the environment need to aid the developer in this process (Legacy Code/Component Wrapper).

For generic Node-RED flows that need to be created for any application use, the design environment needs to implement a specific process through which the created flow can be copied in a generic template Node-RED image, enriched with any needed extra dependencies (e.g. newly imported palette nodes from external repositories), and declared as an actionable docker image towards the FaaS platform (assisted through the Node-RED flow wrapper). This part may also be implemented as a pattern, so that it can be easily reused by the application developer through abstract means. This process includes also the availability of baseline processes and skeleton flows needed to interact with the FaaS platform. For example, the latter assumes that any registered function artefact exposes two endpoints (an /init method and a /run method) used by the FaaS platform to initialize and then execute the function logic.

Further functionalities may need to be included, such as the need for functional annotations at the function level (e.g., inclusion of external library dependencies), ability to use semantic annotator nodes that will enrich the semantic descriptions of a flow, and upon finalization, they will create the necessary instance triples to be stored in the Inference Engine (supported by the Semantic Extractor subcomponent that will be responsible for processing elements like the created flow and perform the relevant semantic queries in order to enrich their description). One aspect of investigation is whether recommendations and suggestions based on the initial annotations and the stored semantic information in the application and resource models can be applied during the application design time. Examples for such suggestions may include links between patterns since the latter typically come with suggestions for complementary or adversary patterns. Pattern implementations may be annotated with semantic fields so that they can be mapped during runtime (i.e., understand which subflow relates to which pattern and from that extract dependencies between needed patterns that will feed the suggestions). Another aspect of semantics includes the definition of the relevant images, endpoints etc. based on the declared developer annotations (e.g., use of specific images for GPU enabled actions etc.). This means that the produced application graph needs to pass through the semantic block in order to define and enrich the according fields, prior to the graph submission towards WP4. This process is also supported by the Semantic Extractor subcomponent.

The outcome of the process in the Design Environment is to have created and registered different application blocks on the FaaS platform and forward the overall application graph towards WP4, enriched

with a number of features such as annotations and semantics that can be used further down the PHYSICS process for purposes including functional adaptation, preferable means of management, non-functional requirements etc.

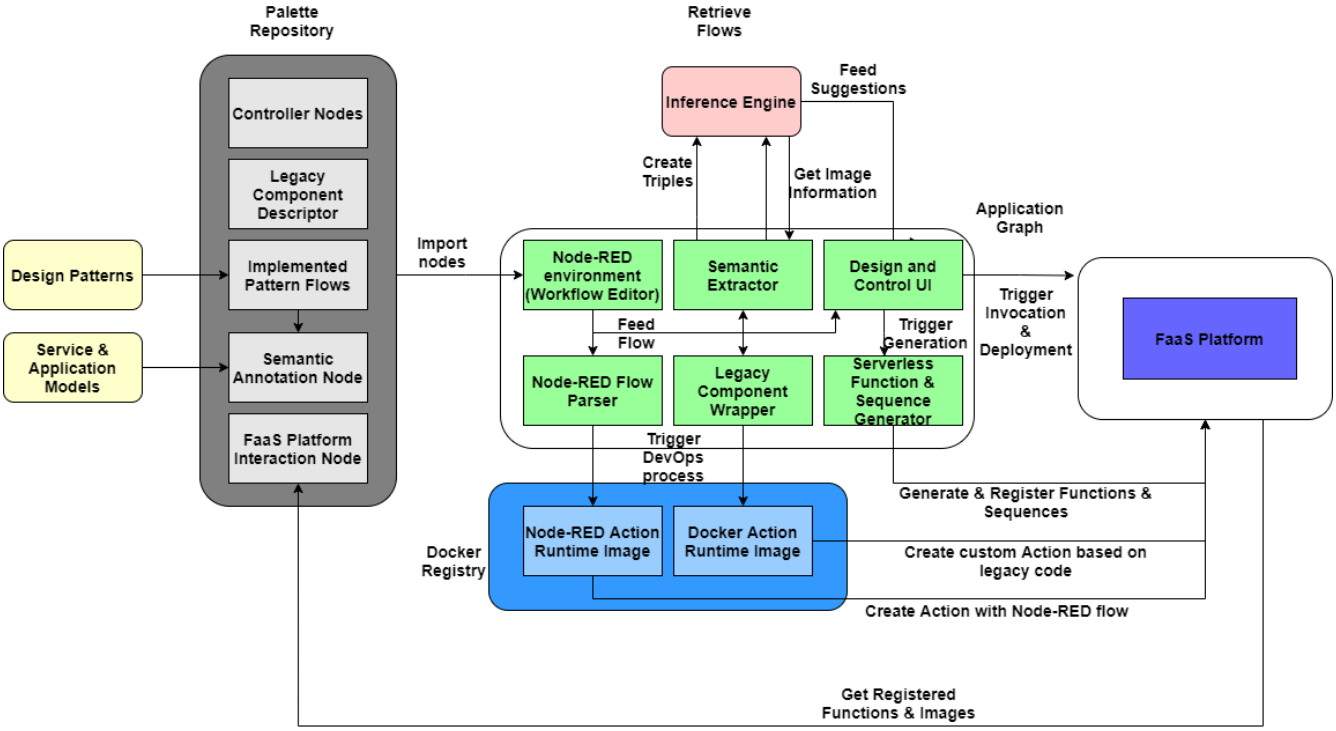

Figure 8 - Design Environment Components and Interactions with other elements of the PHYSICS platform

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).

3.2 Application Semantic Models

Component Description

The main goal of the Application Semantics Models 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 will provide the types of entities and their relationships, i.e., workflows, functions, resource requirements, and locality constraints, and will be expressed as an ontology. The workflows defined in the Visual Workflow component (section 3.1), are then expressed as individuals that belong to the classes of said 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 main output in this regard is the role of the Application Semantic Models in the reasoning processes implemented in the Inference Engine component.

The OWL ontology will describe 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 workload, 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. The target is to imprint all the information related to application workflows, so that it can be used by the Inference Engine in conjunction with the Service Semantic Models (section 3.10). The most fundamental operation to be done, driven by the Semantic Extractor included in Section 3.1, 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 functionality will be 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 data format to be 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 to be linked with the Services Semantic Models (section 3.10) through the resource constraints and requirements that workflow nodes have as attributes. The relationship of this component with other components is presented in Figure 9:

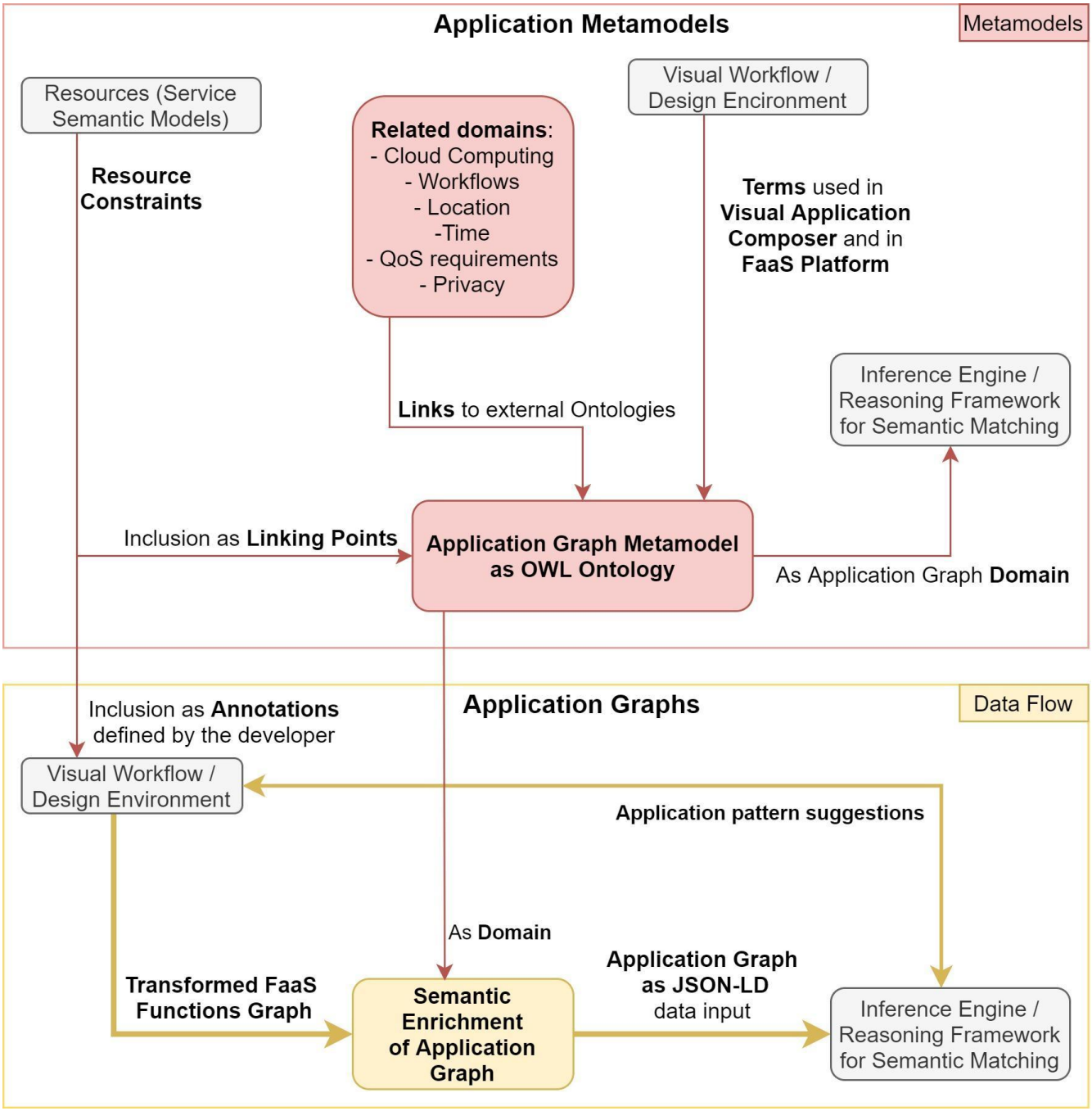

Figure 9 - Application Semantic Models components

In the above schema, the composition and usage of the ontology is highlighted in red arrows and boxes, the semantic enrichment of application graphs is presented in the yellow arrows and box, and the gray 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 to be handled by the component

- Annotating mechanisms at the function level
  - Through specialized annotation nodes in node-RED or code level annotations in the visual workflow 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 design graph
- Means of specifying the kind of environments that parts of the application can be deployed at
- Graph representation that is to be eventually forwarded to the resource optimizer

Outputs
- Application graph metamodel (a static ontology/metamodes model) that describes application models
- Transformed application model/graph that is inference-capable
- Application and function requirements and constraints within the application graph

3.3 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.

c

These patterns can be implemented in any language or framework the FaaS platform supports or even as Node-RED flows which provide the flexibility to maximize their optimization and re-usability in the FaaS platform or in more general contexts. The implemented patterns can exist in a repository, a file server or even directly to the FaaS platform as registered functions/sequences accessible to everyone, if the platform allows it. A pattern may be implemented in whatever execution mode included in Section 3.1. Depending on the way the patterns are created, they will be stored and made available through different means. For example, a Node-RED implemented pattern subflow will be incorporated in the default Node-RED palette of the environment. A pattern implemented as a microservice will be made available through the Docker Registry, accompanied by respective Node-RED declaration nodes, so that it can be included in the application graph. A pattern implemented as a function or sequence of functions will be registered during initialization of the FaaS platform. 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 according usage nodes in the Design environment will ensure the creation of flows that include the usage of these 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 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 Architecture in Section 3.1. Part of the pattern documentation will also include 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. Therefore, annotation of these cases will help the developer in understanding which patterns could or should work in collaboration and which ones cancel each other out.

Main issues to be handled by the component

- Means of 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 3.1.
- 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.

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.
- Semantics and description of patterns towards the Application Semantic Models.
- Reusable and deployable pattern artefacts.

3.4 Elasticity Controllers

Component Description

The main goal of this component is to scale a PHYSICS based deployment up or down 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 enough and economical way. 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). The recommendations of the controllers are later realized by the Co-allocation Strategies component (Section 3.13) calling the APIs provided by the Resource Management Controllers component (Section 3.12).

Main issues to be handled by the component

- Calculate minimal configurations based on changes that may require configuration changes (e.g. changes in the workload prediction, changes in the performance requirements, slow system performance).
- Make sure that the deployment is performant enough to meet users requirements and QoS, thus it integrates with the performance metrics monitoring system.

Inputs

- Semantic description of the components and the interaction between them.
- System workload prediction.

- Minimal required performance goal.
- System performance metrics.

Outputs

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

3.5 Inference Engine (Reasoning framework)

Component Description

The Inference Engine or Reasoning framework is responsible for completing three different tasks. Firstly, the Inference Engine will offer SPARQL[4] querying endpoints, to be used by the following components:

1. Global Continuum Placement will query the service graph in order to minimize the number of candidate services that may be used for the deployment of a given application.
2. Visual Workflow will query the Reasoning framework in order to recommend available design patterns to the application developer.
3. Adaptive Platform Deployment, Operation & Orchestration will query the semantic service in order to retrieve the necessary options that need to be defined in the deployment configuration.

To achieve that, this component will incorporate both application and resource data into semantic models. To this end, the ontologies describing applications and services, developed by the Application Semantic Models and the Service Semantic Models components respectively, will be pre-loaded to the Inference Engine. The latter will provide appropriate endpoints for injecting individual application and service data into each relevant pre-loaded ontology model. For instance, an application individual will be the annotated data describing a given application to be deployed by PHYSICS platform, while an individual service is an available resource (e.g. {InstanceType: t2.micro, SustainedClockSpeedInGhz: 2.5, MemoryInfo: {SizeInMiB:1024}}). In addition, the Inference Engine will provide the Quadstore to store all the input data enabling also their interpolation to a global graph which will consist of various nodes such as functions, patterns, containers, cloud resources, QoS metrics and security constraints.

The last task of this component will be to provide reasoning capabilities over the aforementioned graph of applications and services in order to uncover relationships between the graph nodes which are not previously defined. As a result, new edges will be created connecting these two graphs. Initially, this task will be based on open-source reasoners such as HermiT5 and Pellet6. However, after the first version of this component machine learning techniques will be incorporated for further improvement of the provided reasoning which may enable faster reasoning.

The above-mentioned interactions along with a preliminary reference architecture for the Inference Engine are depicted in Figure 10.

[4] https://www.w3.org/TR/rdf-sparql-query/
[5] http://www.hermit-reasoner.com
[6] https://www.w3.org/2001/sw/wiki/Pellet

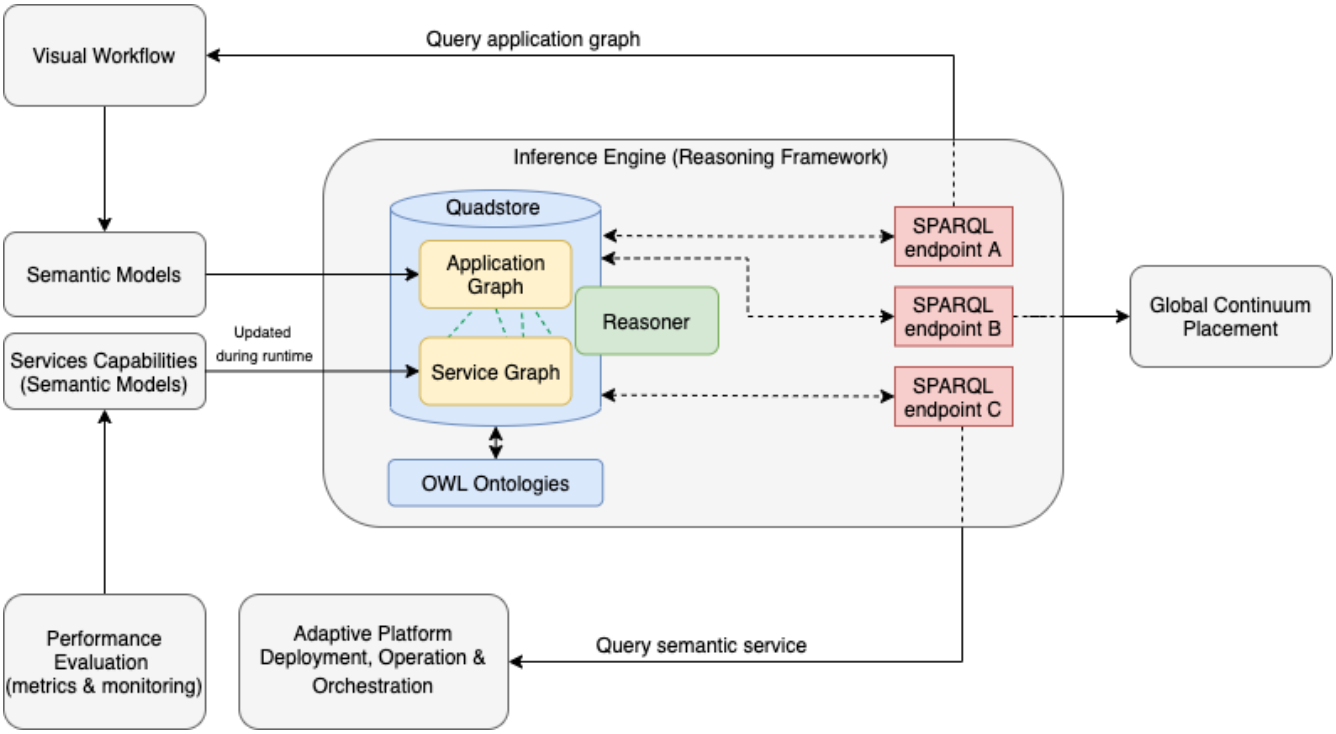

Figure 10 - Inference Engines interactions with other components

Main issues to be handled by the component

Inference Engine contribution is three-fold:

1. Resource Filter: Of the total resources available, only those that meet the given application requirements will be included in the deployment graph which will be accessible from SPARQL endpoint B. In this way, the Inference engine will reduce the search space of Global Continuum Placement.
2. Runtime adaptation: The service graph will be updated during runtime according to the latest performance data. As a result, an enriched deployment graph will be produced from the Inference Engine.
3. Querying nodes: As mentioned in Inference Engine description, the inference engine will implement appropriate queries to other tasks, as shown in Figure 9. Namely,
    a. Global Continuum Placement will perform queries such as "SELECT * FROM Service WHERE Service.GPUenabled=1".
    a. Visual Workflow will query whether a Design Pattern used in the current flow has any complementary or counter-productive pattern combinations. For instance, given that the developer is using a pattern that may halt requests due to a temporary failure (Circuit Breaker Pattern), the Visual Workflow will POST a relevant request to the Inference Engine which will trigger a query such as SELECT pattern, complementary WHERE pattern. description CONTAINS. In this case the Inference Engine will reply with the complementary 'Retry' pattern as a proposal.
    b. Adaptive Platform Deployment, Operation & Orchestration will query the Inference Engine for available images which are optimized for a certain type of service.

Inputs

- Application Ontologies and Service Ontologies in OWL[7] format, designed in the open-source Protege Editor[8], provided by Semantic models and Service Capabilities.
- Individual Application description (e.g., function requirements, functions sequence, location constraints etc.) provided by Visual Workflow in JSON-LD format.
- Individual Service description (e.g. type, CPU, RAM, location, etc.) provided by Service Capabilities also in JSON-LD format.

Outputs

- The Deployment Graph (i.e. all the available services which are able to run the given application) to the Global Continuum Placement component.
- Relevant Design Pattern id (if there is any applicable) to the Visual workflow according to the developer inputs.
- List of the available images which are optimized for the given type of service to the Adaptive Platform Deployment, Operation & Orchestration.

3.6 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 on top of the available services.

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) 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 benchmark 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. 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

[7] https://www.w3.org/OWL/
[8] https://protege.stanford.edu

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, or they can be triggered by the Co-allocation or Scheduling strategies when in need to evaluate the outcome of a specific strategy (e.g., scheduling strategy or collocation decision). 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). In any case, and given that these abilities are provided via APIs, they can be included in arbitrary implementation sequences. The subcomponents of this component and the interaction with other components are depicted in the Performance Evaluation Framework diagram of Figure 11.

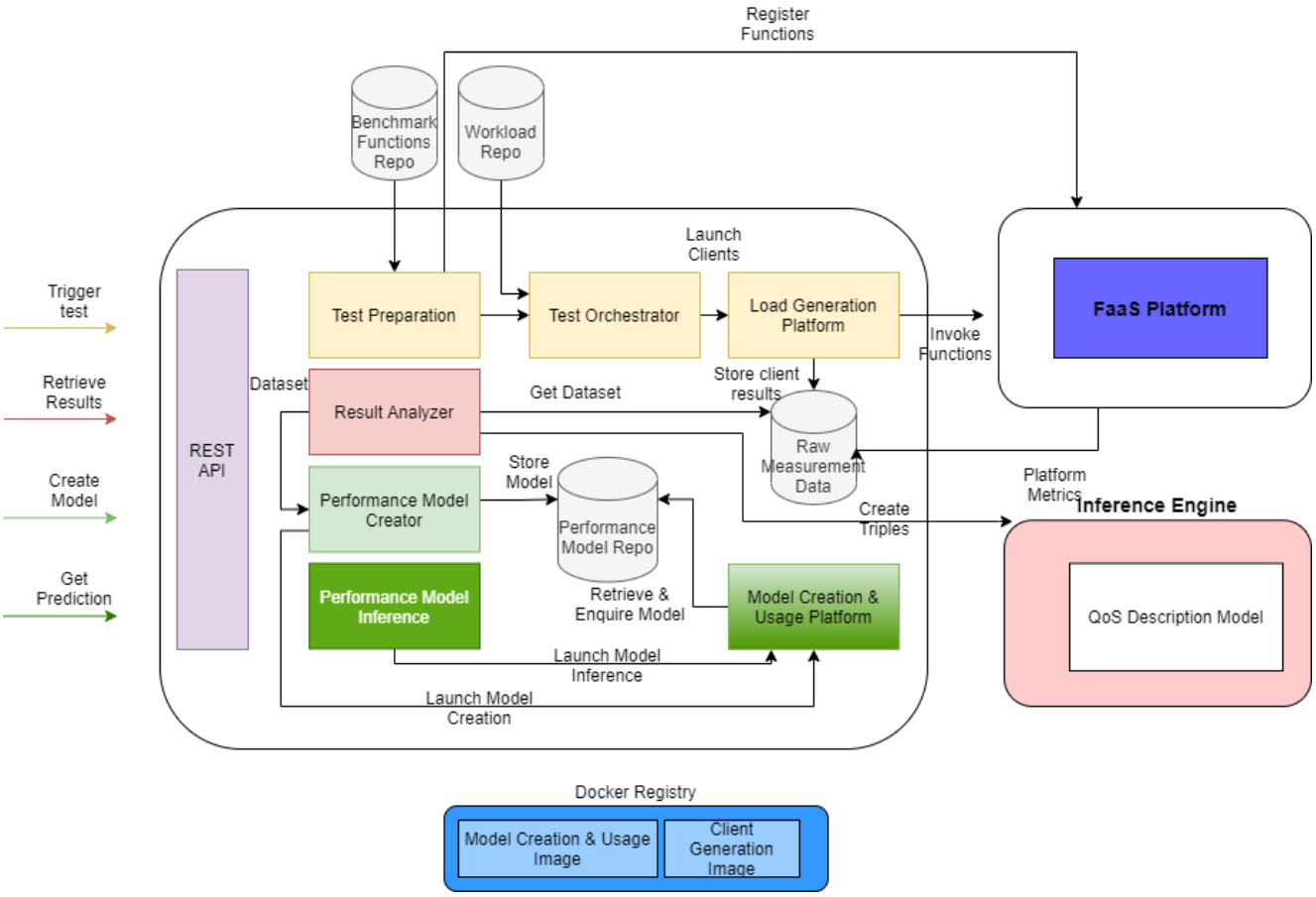

Figure 11 - Performance Evaluation Framework Diagram and Interactions

Initially a REST API layer will expose the main functionalities of the component (test triggering, result retrieval, model creation and model usage/inference). A variation of these calls will be created in order to address different requirements, e.g., in types of result retrievals such as averages, individual tests, grouping based on certain test features etc.). During test triggering, an initial phase includes the registration of the relevant test functions in the target FaaS platform, followed by the orchestration component for the test. This includes the launching and initialization of necessary load generation clients, the management of the test lifecycle (starting, finishing, retrieving results, cleaning up resources) and the management of the according client resources (residing in a container cluster such as Docker Swarm or Kubernetes and depicted as the Load Generation Platform). 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.

Main issues to be handled by the component

- Run periodic benchmarks towards available services in order to evaluate their performance, performance variability etc.
- Define and design workloads that are representative of testing scenarios, use case needs or anticipated usage
- Potentially analyze 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

- Benchmark functions executables.
- 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.

3.7 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 specific component will have the ability to schedule application workflows and propose a resources 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 DAG of tasks -functions- (in a FaaS programming model) with particular requests in resources (CPU, Memory, Bandwidth, GPUs, etc), possible constraints (execution only upon one infra: edge to satisfy data sensitivity/locality obligations, etc) and potential scheduling objectives (energy, latency, data movement minimization, etc). In parallel the component will consider the computing resources characteristics (number of total CPUs, amount of available bandwidth, energy, cost, etc) and availabilities (remaining amount of memory available for allocation, etc) along with potential proposition for resources selection and scheduling coming from offline/prior simulations. Based on these inputs the Global Continuum Placement component will offer the

possibility to perform placement using a group of scheduling policies which will vary in efficiency, complexity and speed. 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 MILNP or genetic) that consider multiple objectives and various QoS to address will return optimal or sub-optimal results in a less timely manner.
In particular, this component will be composed by: 1) a subcomponent that will consume 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 which will be 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 will provide the scheduling decision to be pushed as a YAML file to the centralized orchestration component.

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 availabilities.
- 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.

Inputs

- Application graph decomposed in functions including the resources needs and constraints coming from the applications semantic model 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 FaaS Scheduling Algorithms component as a YAML file.

3.8 Distributed Memory Service

Component Description

The Distributed Memory Service 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 Distributed Memory Service (DMS) component provides an in-memory distributed state service that allows functions to store objects out of main memory and share it among other functions efficiently. Several issues must be considered in order to achieve good performance in a FaaS scenario. The DSM must run collocated with in 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 DSE provides a simple interface to access the data: *get* and *put* operations

Figure 12 (top) shows the internal architecture of the DMS. The DMS design is based on *Pocket*[9]. The DMS has three sub-components: one *controller*, one or more *metadata servers* and one or more s*torage 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 is built on top of Apache Crail[10], The *storage servers* are in charge of storing the data. They can used different storage media such as: DRAM, NVMe, SSD or HDD.

Figure 12 (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. The workflow must import this library. Solid arrows in Figure 11 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 providing information about sequence (step i). These characteristics are degree of parallelism and latency-sensitivity, capacity, and throughput. The controller determines the storage server type, the number of storage servers and 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 (steps 1 and 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.

[9] https://www.usenix.org/conference/osdi18/presentation/klimovic
[10] https://crail.incubator.apache.org/

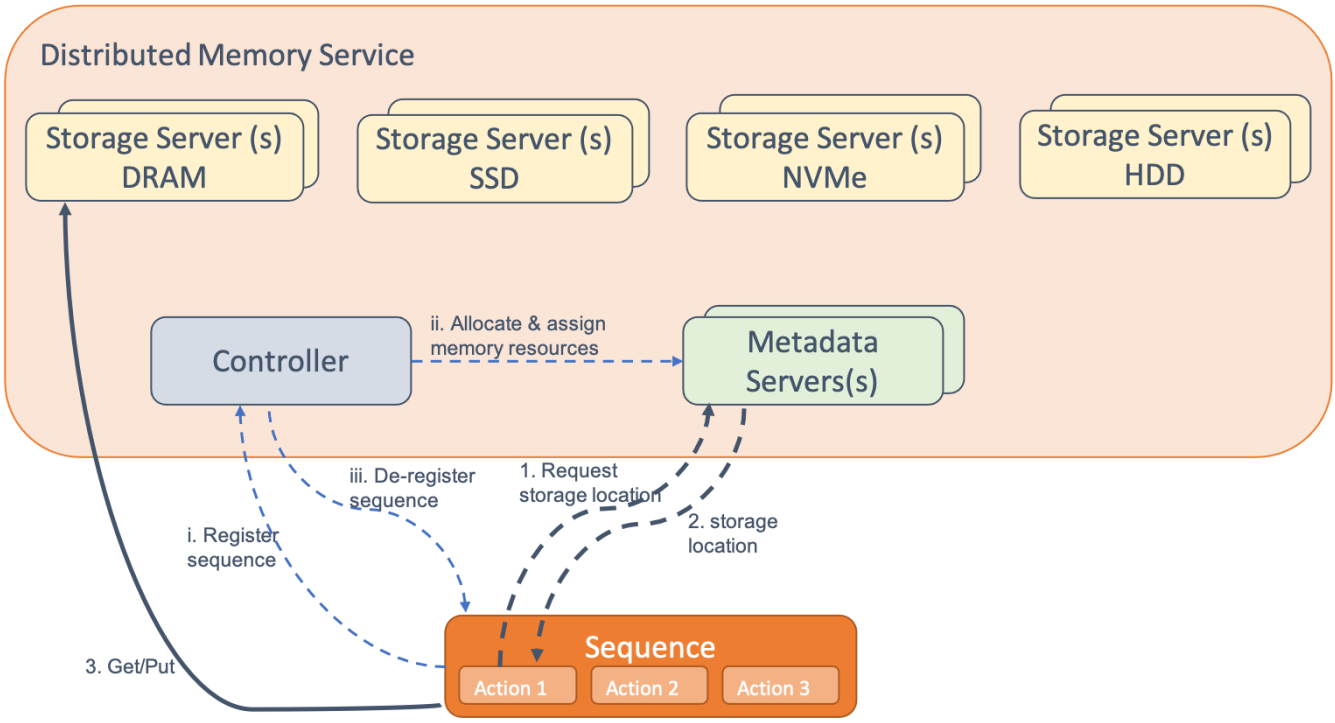

Figure 12 - Distributed Memory System Architecture

Main issues to be handled by the component
The Distributed Memory Service must meet some requirements:

- Data consistency
- Support different cloud providers (AWS, Google Cloud, …)
- Provide fast data access

Inputs

- The DMS needs the description of the cluster (number and location of nodes, type and size of available storage…) where the workflow will be executed in order to deploy the DMS components.
- Workflow definition: functions and maximum number of instances of a function.

Outputs

- The information stored in the DMS.

3.9 Adaptive Platform Deployment, Operation & Orchestration

Component Description

The objective of the Adaptive Platform Deployment, Operation & Orchestration component is to enable easy dynamic orchestration, reconfiguration & adaptation of the applications defined by the application definition schema. That schema is forwarded by the Inference Engine (Reasoning framework) component and the Global Continuum Placement component and, based on that definition, the orchestrator must be able to operate given multiple cloud providers and edge devices scenarios that can potentially be located in different regions, to have different resource catalog and to support multi-tenancy domains. The orchestrator is able to deploy the applications with a QoS and QoE definitions levels given by the application

and user, it traces those values by gathering the metrics performance from the underlying infrastructure, cloud providers and applications performance and, if necessary, the orchestration is able to respond to any deviation with the necessary operations dynamically.

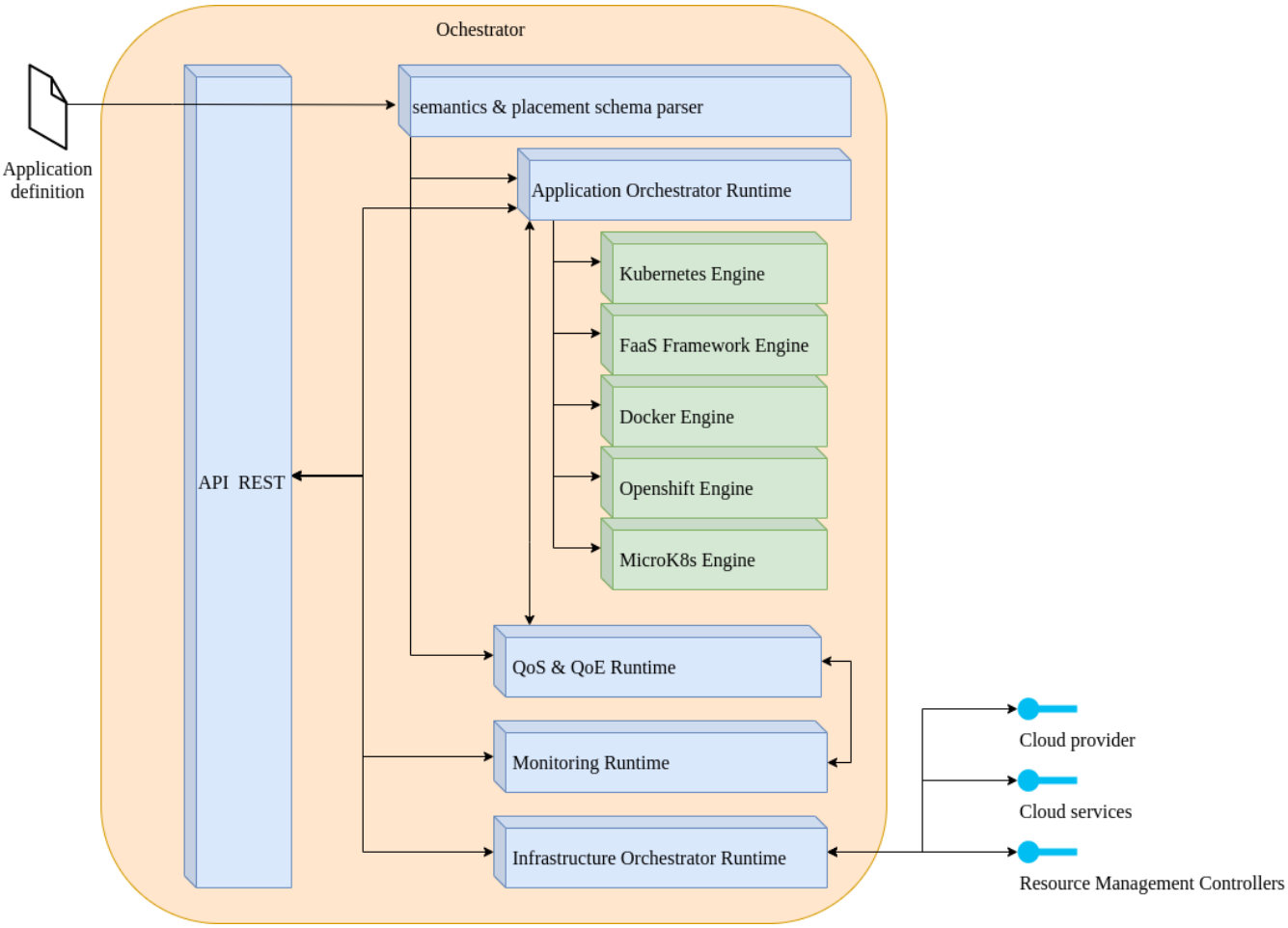

Figure 13 - Orchestrator Architecture

Main issues to be handled by the component

- Fault tolerance
- Scalability
- Interpret and understand the cloud application specification coming from the Reasoning framework
- Interpret and understand the placement location defined from the Global Continuum Placement component
- Cloud provider catalog agnostic
- Trace and assist to the necessary QoS and QoE levels of the applications and, if necessary, respond to any deviation of that conflict with the quality level definition.
- Support multi-tenancy
- Collect the performance metrics of the applications and cloud services.

Inputs

- Definition schema of the application components defined by the Reasoning framework.
- Definition schema of the applications workload defined by the Global Continuum Placement component.

- Qos & QoE definitions of the applications.
- Metrics from the application and cloud services performance.

Outputs

- Resource Management Controller API: to provision, tune and adapt the underlying resources physics controller
- Cloud Provider API: to provision, tune and adapt required infrastructure
- Cloud Services API: to provision, tune and adapt existing cloud services used
- Applications deployments based on the definition schema
- QoS & QoE monitoring metrics

3.10 Service Semantic Models

Component Description

The main goal of the Service Semantics Models component is to capture information of the available physical and virtual resources (smart devices, cluster nodes, vm instances etc.) and describe them in a format that is homogenous in structure and can be utilized by other components. This information in turn, will be utilized to enhance the application deployment and resource management/optimization processes by the relevant components. In accordance to the application semantics respective topologies will be formed that can later on be instantiated with specific information to be further analyzed. The main components that are to benefit from this information are:

- Inference Engine, in order to match application components with resources able to host them.
- Scheduling Algorithms & Co-allocation Strategies, to achieve resource deployment and allocation optimization.

The resources that are to be described include public, private and hybrid cloud servers on a node level and smart devices as resources (e.g. raspberry pi). Furthermore, information such as api keys and specifications will be captured to enable the (automated) invocation of specific cloud vendor services such as Amazon's lambda, EC2, S3 and Azure Function in a programmatic fashion.

The component will exploit a Domain Specific Language such as TOSCA, to capture functional and non-functional properties of the resources. These will include aspects such as physical capabilities (CPU, RAM,etc.), networking, endpoints required for application realization, performance benchmarks of different types of applications, location and vendor specific information. This produced file will be in a TOSCA appropriate YAML format that can be accessed via specified API calls.

Main issues to be handled by the component

Two main challenges emerge that are directly associated with the component's functionalities; The first one is which information to include in the semantic representation of resources and the latter, how to structure and represent this information. The plethora of standards and practices on these topics still holds a portion of subjectivity and relativity to the use of the ontology at hand. The exploration of semantic representation for ontology creation in FaaS applications and edge resources is still new and remains to be researched in order to better understand the potential domain specific problems that will emerge. Outside of the more traditional ontology aspects new cloud services like Amazon Lambda and Azure Function will be necessary to be included in the respective ontology and aspects such as availability and resource consumption could be potentially crucial for semantic representation of edge devices.

The second challenge that emerges is gathering information that originates from different types of resources that can be accessed by different native API's and operational systems. To that end this component will act as a middleware that with minimal input such as credentials to a provider's API or keys required to establish ssh connection, will query the resources to gather the necessary information.

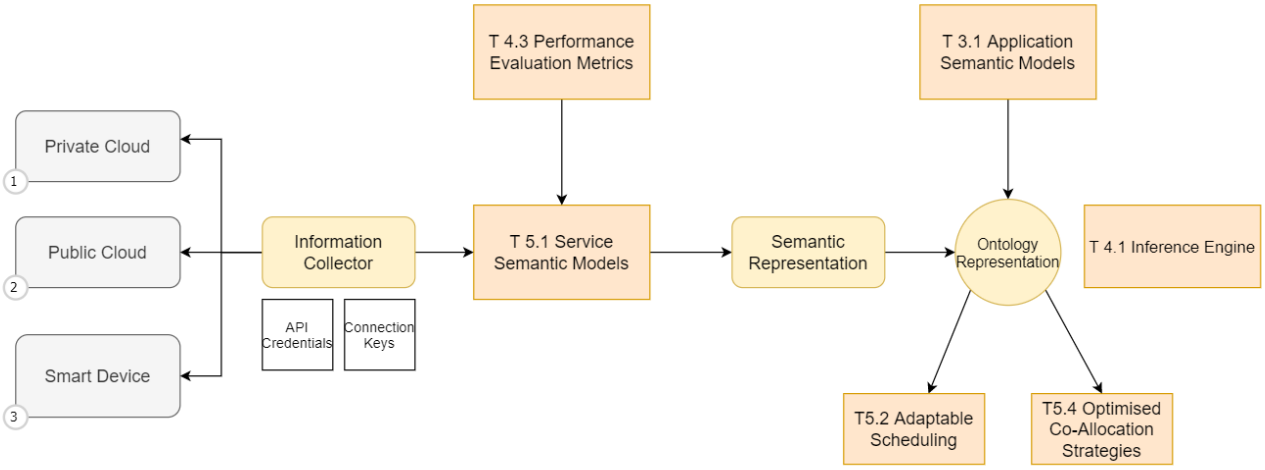

Figure 14 - Service Semantic Models Architecture and semantics interactions

Inputs

- API credentials. This includes Vendor specific credentials and access keys for platform API's.
- Information on resource performance produced by relevant benchmarks and/or aggregated metrics from application deployment monitoring.

Outputs

- Semantics modelled according to TOSCA and OWL standards represented in the appropriate file formats, YAML and JSON.

3.11 Scheduling Algorithms

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.

In particular, this component will be composed by: 1) a subcomponent that will consume inputs collected through API calls related to 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 with the ability to be programmed in different programming languages and deployed as Kubernetes pod along with an OpenWhisk scheduling related part which will 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.
Each local cluster will have 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. 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 utilize 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 orchestrator, each cluster will have the ability to deploy different scheduling algorithm per deployed function and even allow the simultaneous usage of multiple algorithms within the cluster at each moment.

Main issues to be handled by the component

- Efficient scheduling of FaaS applications' functions upon the computational resources of local cluster considering resource availabilty and tasks' needs.
- Dynamic adaptation of resource allocations based on possible tasks' needs change (autoscaling).

Inputs

- Local cluster resource availability status coming from the Resource Management Controllers through API calls
- 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
- The application task graph coming from the Service Semantic Models in the form of YAML

Outputs

- The resulted scheduling decisions are forwarded to Resource Management Controllers for deployment in the form of YAML.

3.12 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 will 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.
- **Specific purpose Schedulers**: Provides the needed API to select the scheduling algorithm (developed in the previous subsection) to use for specific applications (i.e., Kubernetes pods). This could be managed by its integration with the OCM, which will ensure the right scheduler is configured in the cluster where the application is going to be deployed.
- **Descheduler Operator:** After the initial scheduling, there are several events in a given cluster that can make the initial scheduling decision less optimal, such as nodes failing, applications consuming less/more cpu/memory/bandwidth than expected, … This operator can reschedule the pods after its initial deployment based on an optimization function.
- **Performance Operator:** For applications that require some specific QoS levels (like real time applications), or to ensure a better mapping of a group of applications in the same node, there are different knobs in the linux kernel that can be used, such as hugepages, cpu pinning, NUMA-awareness for process to core assignment, etc. This operator is in charge of exposing those knobs at the Kubernetes level so that better performance/isolation can be achieved for the applications that need it.
- **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 centralized way (install, configure, manage) as the number of edges can grow fast. To tackle this problem we are working on a new OpenShift/Kubernetes flavor optimized 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 an simple, descriptive way on the set of kubernetes clusters
- Provide the needed APIs for the upper layers to:
    - Get monitoring information about the cluster and application 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 to use, the collocation preferences, ...

Inputs

- Cluster information needed to install/configure a new cluster (subnets, IPs, size, provider, ...)
- Set of YAMLs defining the application and its extra configurations such as the specific scheduler to use, the collocation hints (affinities, anti-affinities), the isolation needs (cpu pinning, NUMA-awareness, network bandwidth limits, ...)

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.

3.13 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 the workflows.

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 15 shows the different components of the Co-allocation Strategies component. The components whose output is the input of the Co-allocation strategies are shown on the left of the figure. The workflow is needed for knowing which functions use the output of the previous function. These functions are candidates to be co-allocated if the size of the output data from the first function is large (avoiding network costs). The workflow may include information such as minimum hardware resources needed by a function, if parallel branches must be executed on different nodes or may be executed on the same node, these restrictions are analyzed and taken into account by the a*pplication constraints/preferences* subcomponent. The available resources are analyzed by the *computational resource requirements* component. The *cluster architecture* component creates a representation of the topology of the infrastructure which contains as nodes the hardware and its main features and as edges the latency among two nodes and the bandwidth. The *resource consumption* component retrieves information on the current resource usage of each node. The *application co-allocation algorithm* component will process the previous information and produce a co-allocation strategy for the new workflow that will be send to the Scheduling Algorithms component.

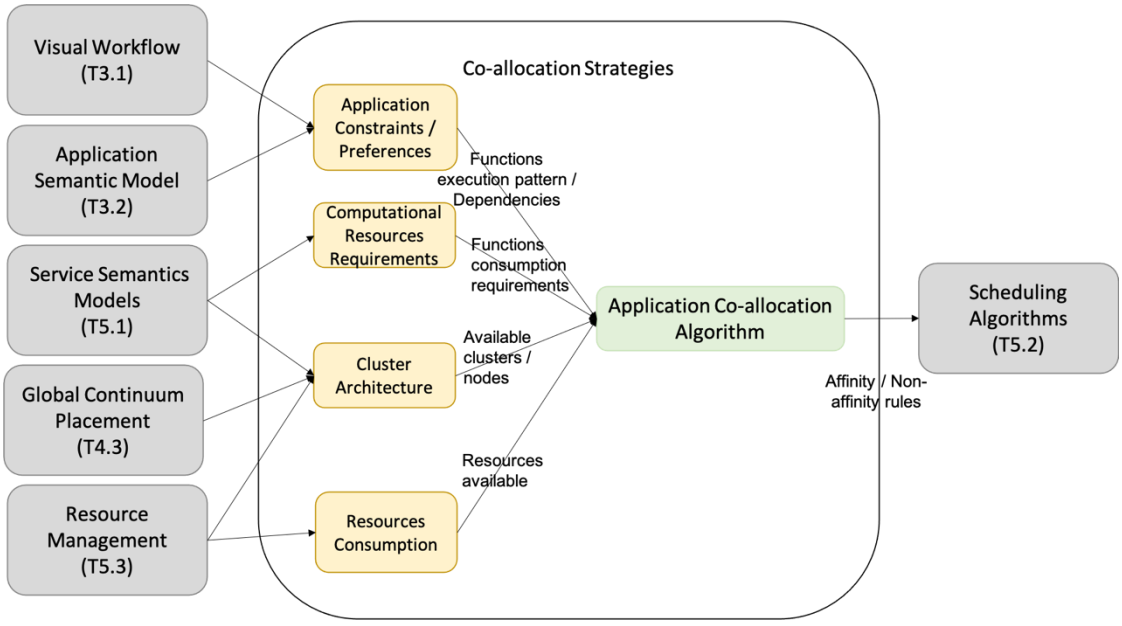

Figure 15 - Co-allocation Strategies component internal architecture

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 collected by the Resource Management Controller component and stored in Prometheus. The metrics will show the CPU, memory and network usage of the different functions and nodes available in the system.

Outputs

- Affinity rules on the pods for a workflow will be consumed by the Scheduling Algorithm component.

4. COMPONENTS INTERACTIONS

4.1 Application Development Environment (WP3)

Figure 16 describes the interactions between the components of the visual design environment and external ones in PHYSICS and beyond. In this image 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 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 subflows 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 the container management environment for microservices, which are however part of the interactions in Section 4.3), but will need also to contact the Design Environment (and specifically the SFG component described in Section 3.1) in order to handle the function registration process in the target Openwhisk instances. This is specifically needed for the code adaptation process described in Section 3.1, necessary in order to create the registered functions in the FaaS platform. This process needs an application code level view that is available in WP3, hence this is the location where the according functionality needs to be implemented. 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. If there is need for communication/triggering between application parts across these instances, web actions/events should be used from the first subflow to the next. This needs to be considered in the application design phase since it creates the problem that now the ability to define sequences of functions in Openwhisk cannot be used for the overall application flow. A potential solution is for the developer to define groups or “regions” of functions, that imply that all the functions in this group will be sent to the same Openwhisk instance (and therefore the sequence feature of Openwhisk can be used for this specific part of the application). Based on the placement decision of the Optimizer for which subflow/group/function region to go in which Openwhisk instance, the WP4 Orchestrator can then guide the process of registering the corresponding function groups in the associated according Openwhisk instances. In order to achieve this, it should call the API of the SFG component in WP3 in order to dictate the according registration in each selected Openwhisk instance.
5. Following the receipt of the according call, indicating the function group and the target FaaS platform instance, the Design Environment (and the SFG component) will need to carry out the

registration process of the functions, sequences, or container functions on the target platform. During this transformation, it may collaborate with the semantic application models in order to retrieve functional information (such as the images needed based on the selected architecture). The result of this process, once all calls for each function group are complete, is to have a registered application across the continuum.

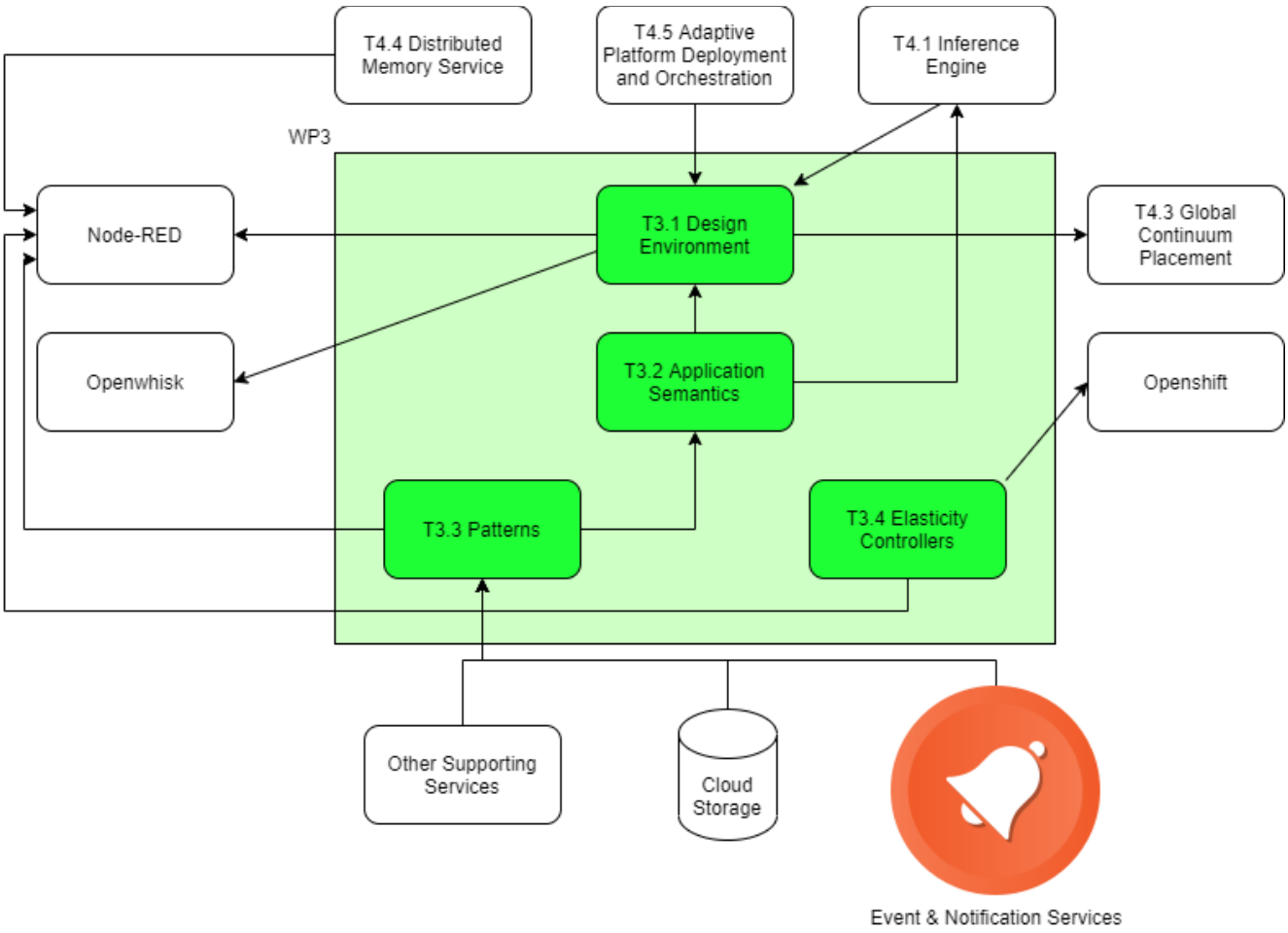

Figure 16 - WP3 internal and external interactions

4.2 Continuum Deployment Layer (WP4)

Figure 17 represents the WP4 components interaction and describes the workflow followed to deploy the Physics applications.

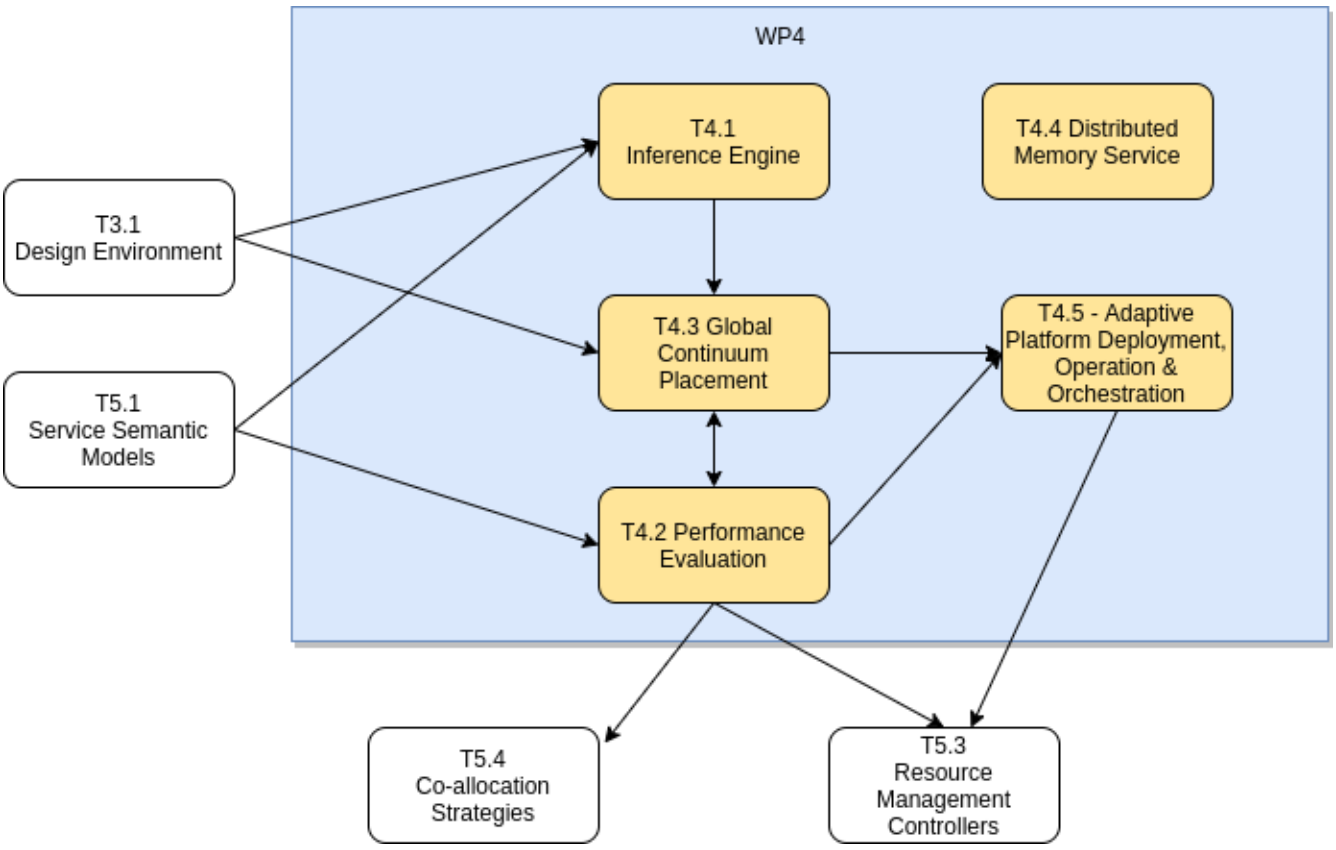

Figure 17 - WP4 internal and external interactions

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

1. The Inference Engine will collect and process the input from the Design environment component. The Inference Engine Component will aggregate all the semantic description onto the application description together with the rest of the detailed configurations for the initialization. Besides the description, all dependencies required for the application functionality will be included and sent to the Global Continuum Placement component.
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 interact with the Adaptive Platform Deployment, Operation & Orchestration component. The Adaptive Platform Deployment, Operation & Orchestration component will expose an API that can be used across the rest of the WP4 components, that will simplify the endpoints request to a single interface endpoint. In regard to the Placement information, it will consume the information related with the availability of the infrastructure available within the platform and use that information for the aggregation phase after the input of the Inference Engine. Once the placement information is completed, it will be sent to the Adaptive Platform Deployment, Operation & Orchestration component.
3. The Adaptive Platform Deployment, Operation & Orchestration component will collect the input from the Global Continuum Placement component through an API REST. It will read and parse the Application schema including the configuration setup and the placement information and will prepare and execute the required operations to achieved deployment success. If there is some infrastructure demand according to the Application definition schema 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 Adaptive Platform Deployment, Operation & Orchestration component performance metrics data through the Orchestrator API REST interface. These data are kept and analyzed, 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.

4.3 Infrastructure Layer (WP5)

Figure 18 represents the WP5 components interaction and describes the workflow followed to deploy the Physics applications.

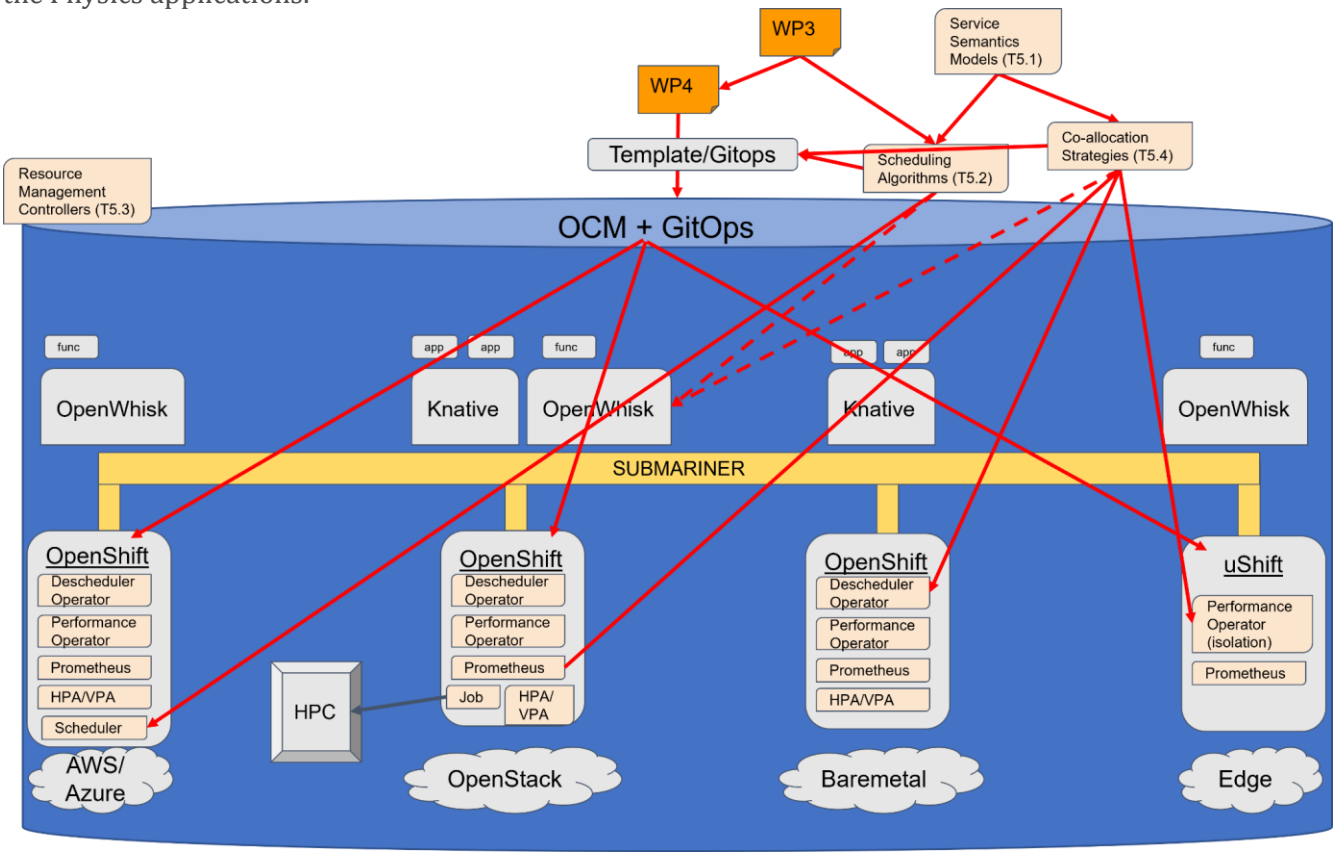

Figure 18 - WP5 internal and external interactions

The interactions between the components are the next:

1. Output of WP3/4 will be a YAML file that the OCM would need to process to ensure it is executed in the desired cluster, with the correct settings. This YAML file has information about the application to deploy, its needed component, and perhaps information about what cluster can/cannot be used (by using labels that reference the available clusters). This is the format the OCM knows how to process
2. The Scheduling Algorithms component will be in charge of extending the YAML file with information regarding the scheduler to use, as well as the need for deploying the scheduler plugin in the desired cluster. It will take its decisions based on the inputs provided by the Semantics Model Service as well as monitored information from the clusters
3. The Co-allocation Strategies service will extend the YAML file with information about the affinity and/or anti-affinities for the different application components. To provide that output it will use the

information provided by the Semantics Model Service, as well as monitored information from the cluster(s) where the application is going to be deployed.

4. Finally OCM is in charge of executing the final template at the selected cluster(s), creating the needed components, deploying the applications/pods (maybe even the requested scheduler algorithm) with the requested options (isolation needs, affinities, scheduler algorithm to use, …).

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 Kubernetes[11] cluster.
This section contains an overview of the before mentioned strategies, while further details will be provided in the future deliverable D6.1 – "Prototype of the Integrated PHYSICS solution framework and RAMP V1", which will be the outcome of WP6 task T6.1.

5.1 Development Strategy

The PHYSICS RA design approach plans to consider a microservices architecture implementation, with services/functions interacting among them through REST APIs based on OpenAPI specification. In that respect, all microservices run in containers on the Kubernetes platform. In order to support the development and testing activities, a CI/CD approach leveraging DevOps methodologies will be 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's 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 will be implemented in a blueprint reference testbed environment. The Continuous Integration tools will be 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* will be created for hosting the DevOps tools, which are:

- **Gitlab**[12] is a Git repository manager that lets each developer teams collaborate on PHYSICS 's source code.

[11] Kubernetes (https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/)
[12] Gitlab (https://about.gitlab.com/solutions/agile-delivery/)

- **Jenkins**[13] is the de-facto standard open-source automation server for orchestrating CI/CD workflows. At the same we also plan to evaluate the possible usage of Tekton[14] tools, since it allows to implement pipeline using YAML file
- **Harbor**[15] is a popular Docker registry CNCF compliant.
- **OpenLDAP**[16] is used as the single user directory for all tools, centralizing authentication and simplifying management of developer accounts.
- **Helm**[17] is a package manager that streamlines installing and managing Kubernetes applications.

The Figure 19 below shows a possible workflow describing how CI/CD works for a specific partner (e.g. Partner "A"). When a developer pushes new component code, Gitlab 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 will have as many namespaces as the partners in order to maintain the correct isolation between all PHYSICS partners. In order to deploy the component Helm manager will be used. At the end of the process, Jenkins sends a notification to a dedicated CI/CD channel on the PHYSICS **Slack**[18] 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.

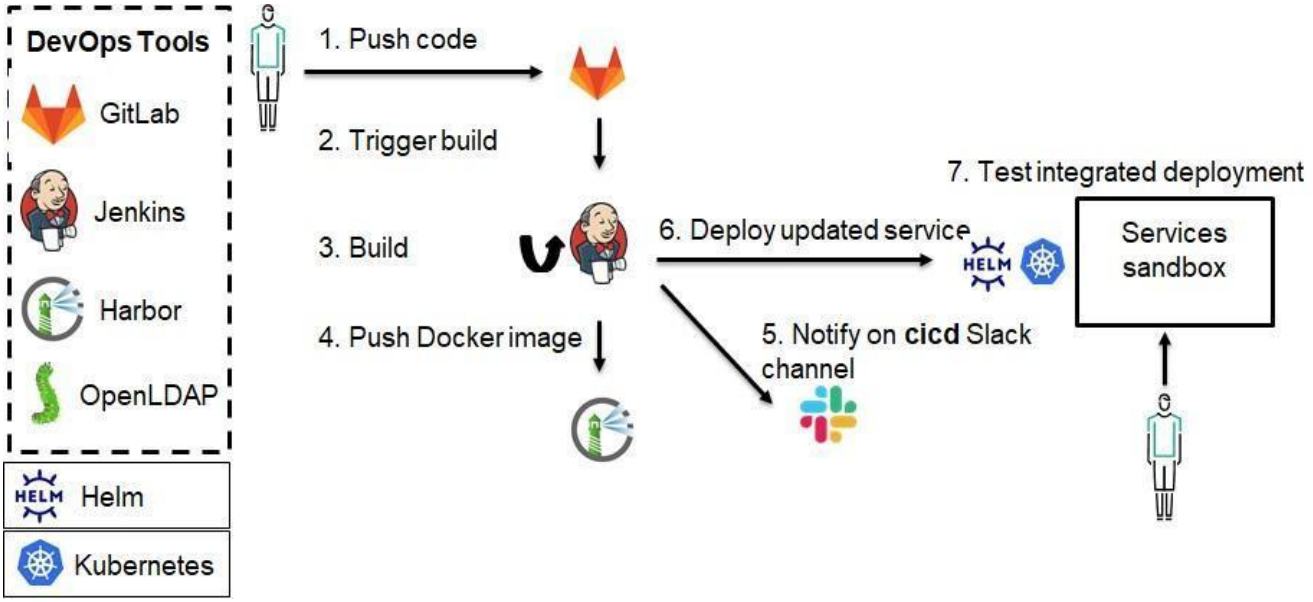

Figure 19 - CI/CD workflow example

Moreover, in the second phase of the project we intend to evaluate possible further enhancements of the process by adopting a DevSecOps [15] approach and including the related tools like **Sonarqube**[19] in the CI/CD pipeline.

[13] Jenkins (https://www.jenkins.io/doc/)
[14] Tekton (https://tekton.dev/docs/)
[15] Harbor (https://goharbor.io/docs/2.3.0/install-config/)
[16] OpenLDAP (https://www.openldap.org/doc/admin25/)
[17] Helm (https://helm.sh/docs/intro/)
[18] Slack (https://slack.com/intl/en-pt/features)
[19] Sonarqube (https://docs.sonarqube.org/latest/)

DevSecOps aims at including security in the software development life cycle since the beginning, following the same principles of DevOps. Security is then considered throughout the process and not just as an afterthought at the end of it, so that different kinds of security checks are executed continuously and automatically, giving developers quick feedback if the latest changes introduced a vulnerability that must be corrected.

Moreover, in this context, in order to facilitate possible Machine Learning (ML) based development and testing, we have also planned to evaluate the introduction of the MLOps methodology20, a compound of Machine Learning and IT Operations), focused on:

- Facilitate communication and collaboration between teams.
- Improve model tracking, versioning, monitoring and management.
- Standardize the machine learning process to prepare for increasing regulation and policy.

In this sense, **Kubeflow**[21] platform could be a good candidate to integrate in our blueprint reference

5.2 Deployment Strategy

The PHYSICS RA design approach plans to consider as baseline deployment strategy 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.

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 problematic to know the POD IP, because the POD could die in any moment so its IP could change many time and this would make 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 POD.
- Automated rollouts and rollbacks
  - Kubernetes allows you to deploy 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 a 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-grade way the allocation of resources (RAM, CPU and Storage) to a specific POD, application or tenant.

[20]MLOps (https://ml-ops.org/content/mlops-principles)
[21] Kubeflow (https://www.kubeflow.org/docs/about/kubeflow/)

- 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 will use two Kubernetes' concepts:

- **Namespace**: They are a logical grouping of a set of Kubernetes objects to whom it's possible to apply some policies, in particular:
    - Quote sets the limits on how many HW 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 also plans to leverage a IaC (Infrastructure as Code) tool like **Terraform**[22] to easily recreate on demand the blueprint environment. Terraform is going to be 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 safe way; moreover, It is an open-source software with a very large community and it is infrastructure agnostic.

In Figure 20 is presented the flow used to deploy PHYSICS components. This flow is composed in 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.

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

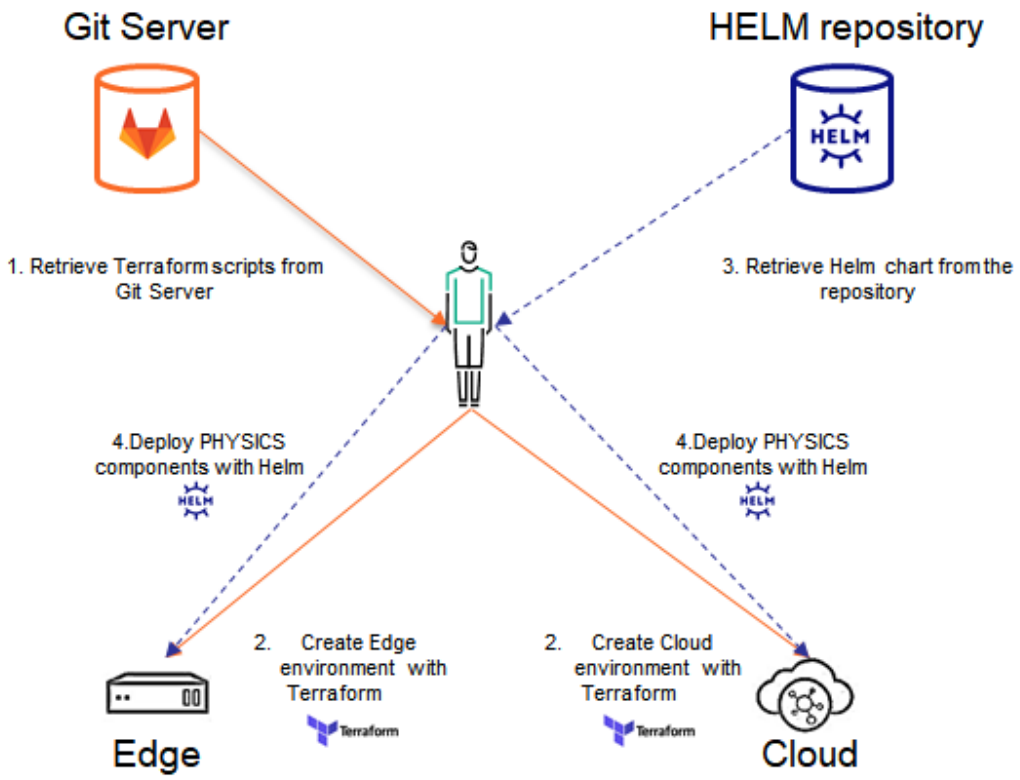

Figure 20 - PHYSICS components deployment flow

6. CONCLUSIONS

This deliverable presents the first version of the PHYSICS architecture. The architecture is defined as a set of views. In this deliverable the functional view of the architecture is presented. This view includes the definition of the software components of PHYSICS, the challenges each component will face during the development of the component, the input and output of each component. The interactions among components are also described.

This deliverable will guide the design and development of the three layers of PHYSICS, which will be documented in upcoming deliverables in work packages W3, W4 and WP5, respectively. This document will be considered a live document to be updated as needed until a second final version of the deliverable is produced in month 23 after the first integrated prototype of the system is evaluated.

DISCLAIMER

COPYRIGHT MESSAGE