PHYSICS

oPtimized HYbrid Space-time servIce Continuum in faaS

D3.2 – Functional and Semantic Continuum Services Design Framework Sci. Report and Prototype Description

Lead Beneficiary	HUA
Work Package Ref.	WP3 – Functional and Semantic Continuum Services Design Framework
Task Ref.	Tasks 3.1, 3.2, 3.3, 3.4
Deliverable Title	D3.2 – Functional and Semantic Continuum Services Design Framework Sci. Report and Prototype Description
Due Date	2023-09-30
Delivered Date	2023-09-29
Revision Number	1.0
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

CONTRIBUTING PARTNERS

Partner Acronym	Role[1]
HUA	Lead Beneficiary, Contributor
GFT	Contributor
HPE	Contributor
InQ	Contributor
RH	Contributor
FTDS	Contributor
ATOS	Internal Reviewer
ISPRINT	Internal Reviewer
INNOV	Quality Assurance

[1] Lead Beneficiary, Contributor, Internal Reviewer, Quality Assurance

Revision History

Version	Date	Partner(s)	Description
0.1	2023-07-15	HUA	Initial ToC Version based on V1
0.2	2023-07-28	HUA	Updated ToC Version, including new section headers where appropriate
0.3	2023-08-10	HUA	Inclusion of performance pipeline text
0.4	2023-8-31	InQ	Inclusion of Gaming server inputs
0.41	2023-9-1	GFT, HUA, HPE	Finalization of the DevOps pipelines and Design Environment updates
0.42	2023-09-08	FTDS	Optimization Digital Annealer Optimizer Pattern and new chapter Advanced customization of Digital Annealer
0.43	2023-9-12	HUA	Inclusion of new patterns text
0.44	2023-9-13	HUA	Final updates in the Semantic Annotations section
0.45	2023-9-18	RHT, InQ	Additions in Section 6, new patterns in Section 4
0.46	2023-9-20	HUA	Final editing and reformatting
0.5	2023-9-21	iSPRINT, FTDS	1st Version for Internal Review
2.0	2023-9-27	HUA	Version for Quality Assurance
2.1	2023-9-28	INNOV	QA performed
3.0	2023-9-29	HUA	Version for Submission

List of Abbreviations

Action	Openwhisk terminology for function
AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
CI/CD	Continuous Integration/Continuous Delivery
DAG	Directed Acyclic Graph
DAU	Digital Annealer Unit
DB	Database
DevOps	Development and Operations
ETL	Extract Transform Load
FaaS	Function as a Service
GCF	Google Cloud Functions
HOBO	Higher Order Binary Optimization
IaaS	Infrastructure as a Service
IE	Inference Engine, part of the Reasoning Framework
IoT	Internet of Things
JS	Javascript
JSON	Javascript Object Notation
JSON-LD	JSON Linked Data
KEDA	Kubernetes Event Driven Autoscaler
K8s	Kubernetes
LDAP	Lightweight Directory Access Protocol
MPI	Message Passing Interface
MVP	Mean Viable Product
NPM	Node Package Manager, repository of Node.js
OKD	Origin Key Distribution
OW	Openwhisk
OWL	Ontology Web Language
PaaS	Platform as a Service
QoS	Quality of Service
QUBO	Quadratic Unconstrained Binary Optimization
REST	Representational State Transfer
SFG	Serverless Function Generator
SPMD	Single Program Multiple Data
UC	Use Case
UI	User Interface
UML	Unified Modeling Language
URL	Uniform Resource Locator
YAML	Yet Another Modelling Language

Executive Summary

The aim of this deliverable is to present the complete PHYSICS Design and Development Environment, the entry point to the PHYSICS platform that aims to overcome existing limitations of FaaS platforms, especially in relation to application creation in a functional programming style. This document has been built on top of the according V1 produced in M13, including the previous content, updated where necessary, so that there is one overall document that describes the entire WP3 outcomes at the end of the project. A summary of the changes made is included in the Summary of Changes table, detailing which sections are identical or very similar to the V1 version and which ones are new or have been heavily updated since V1.

The PHYSICS Design environment is based on the popular Node-RED function and workflow editor, coupled with an internally developed backend system to adapt the described application to the FaaS platform. It offers an extensive list of features such as the ability to use built-in and extended Node-RED packaged nodes from external repositories, packaging all the necessary artefacts aiming at the creation of the deployable function artefact to the FaaS platform. The developers may utilize the presented environment to either create functions directly for execution, workflows for executing functions, as well as workflows executed as functions, including the ability to dynamically alter the location of these deployed functions. Any Node-RED flow can be deployed as a function, exploiting the abundance of existing logic in Node-RED and the visual, user-friendly environment. Two ways of including annotations (semantic nodes and in-code annotations) have been created, giving the ability to the developer to pass directives and preferences down the management stacks, including aspects of locality, affinity, sizing etc. The environment provides a set of built-in patterns and subflows that can be dragged and dropped in the user created flows in order to easily augment the application creation features with functionalities revolving around context management, FaaS interfaces adaptation, ETL and parallelization processes, security and privacy, workflow primitives etc.

The PHYSICS application concepts have been based on an Application side ontology, in order to support the modelling of the application in the context of PHYSICS, as well as to include the relevant defined annotation needs. The aim of this model is to be the basis of the application graph, that will be forwarded to the platform (WP4) for reasoning on the application needs, placement and operation constraints.

With relation to the elasticity logic, a design for bridging it to wider parameters such as cluster management cooperation and the nature of FaaS systems has been sketched, aiming at integrating the application side management with the overall behaviour of the platform and intended needs of both developers and system administrators.

During the final period of the project, the PHYSICS Design Environment has been complemented with a variety of new features, including:

- A centralised login mechanism, integrating with the project main login LDAP process that gives access to the various subsystems (code repository, DE, image registry etc.)
- The ability to import existing images for the creation of a function, exploiting legacy or otherwise created code that needs to be imported and executed as a function
- A cloud-based version of the DE, removing the need for developers to have their own local containers, thus reducing the complexity of using the PHYSICS solution and migrating to a complete SaaS solution

- A subflow export functionality, that aims to automate the process of transforming created subflows into npm packaged nodes, thus simplifying and automating the subflow installation and usage process
- A newly introduced Performance Pipeline functionality, that aims to automate the process of performance benchmarking of the created functions, as well as enabling resource usage annotations for them, leading to optimised management of their execution in the subsequent platform scheduling layers
- Newly introduced visualisations at the user interface, in order to present information on the FaaS cluster status, as well as the results of the function performance analysis
- Finalisation of the PHYSICS Application model and according graph, aiming to include the various different semantics needed by either the developer or the platform layers for optimised management and operation, while enabling its usage at runtime through the Semantic Extractor service
- New provided patterns and helper subflows that extend the currently available list of PHYSICS-produced nodes and subflows available in the PHYSICS palette of the Node-RED environment. The new advancements refer to enhanced monitoring, function result acquisition, routing between alternative endpoints at the application level, automated object files annotation, enhanced optimization capabilities for application level problems, semaphore structures for application level synchronisation, dynamic orchestration patterns for invoking dynamically placed functions
- A Gaming Server implementation, that aims to organise and gamify developer training on the PHYSICS environment usage and function/workflow creation, based on guided flow creation scenarios and automated ways of scoring and validating user progress in the training process
- An updated approach on the usage of KEDA as the basis for autoscaling logic, adapted to the specific needs of FaaS systems and applications

At the end of the project, we have reached a point in which the Design Environment offers a unique set of features and has been significantly tested and improved, based on developer and use case feedback. Experimentation has been performed in order to enhance the robustness of the environment, indicate ranges of parameters or abilities, benefits, or potential drawbacks of the provided functionalities. Examples of the usage (e.g. in the case of patterns) have been included in order to aid developers in their uptake, as well as to facilitate investigation and experimentation on their behaviour. Furthermore, patterns available from the first period of the project have been enhanced. As an example the Openwhisk skeleton interface pattern has been updated in order to support better function-level error logging, while the Split-Join Multiple has been enhanced in order to cover for errors during the parallelization of a task. Inclusion of new features such as the Performance Pipeline support enable project-wide goals regarding optimised placement and operation across the continuum, tailored around the specific functions that are included in the application graph and their computational nature.

The features offered enable the creation of FaaS based applications with limited time and knowledge of the FaaS approach, while the environment has been designed and implemented with the goal of being extensible and adaptable to potential new needs and desires from the users.

Summary of Changes from D3.1

CHAPTER	UPDATE	SECTION(s)
1,2,3,4	Minor or no updates from V1 (relevant sections can be skipped in the Y3 review)	Sections that can be skipped: 1.1,1.2,1.3, 2.1, 2.2, 2.3,2.4, 3.1, 3.2, 3.3, 4.1-4.9
2	New features of the DE and new development of the Performance Pipeline Design and Rationale	2.5, 2.6
3	Updates in the semantic annotators section, update of the PHYSICS ontology and Semantic Extractor implementation	3.4-3.7
4	Newly introduced patterns	4.3.2.Improved Logging, 4.10-4.15
5	New development (Gaming Server for training)	5
6	Almost whole new section	6.1, 6.2, 6.3, 6.4

Contents

LIST OF FIGURES

LIST OF TABLES

-

LIST OF CODE SNIPPETS

1. Introduction

In recent years, infrastructures and services have been characterised by the continuous advancements in the area of cloud computing, starting from the use of ephemeral and elastic virtual machines and reaching the level of lightweight containerization approaches for application packaging and deployment. In order to adapt and exploit this new way of resource provisioning, application developers had to go through an immense adaptation process, having to tackle issues like state handling, ephemeral nature of the resources, need for more distributed and elastic application behaviour, as well as avoid common design pitfalls in the microservice domain [1] .

Architectural approaches such as microservice-based design principles [2] or cloud native application design considerations [3] have aided as a guide in the process, but have not alleviated the developer from the main effort needed to break down their applications, as well as handle the aforementioned issues. Nowadays, with the advent of further computing models like serverless computing and specifically Function as a Service (FaaS) [4], the developers are yet again facing challenges to migrate the core of their applications into more fine-grained, function-oriented chunks.

Indicative challenges [5] in the new computing model include extension of the application domain through a suitable trade-off between expressivity (of the application graph) and simplicity, maintainable composition models for serverless workflows, usage of patterns for serverless applications, inclusion and support for the legacy part of serverless applications, versatile development processes supported by relevant development tools and CI/CD processes. Other approaches [6] also suggest the application development continuum approach for combining functions and services in one environment, using annotations to propagate dictations and managerial approaches directly from the developer to the underlying management layers, abstractions and visual development tools for building non-trivial FaaS applications, while combining and adding legacy parts to short-lived functions without the need for extensive application refactoring. The addition of programming patterns offered as reusable components may significantly aid application adaptation and functionality.

1.1. Objectives of the Deliverable

The main objective of WP3 is to enable abstracted and more flexible exploitation of compute continuum services by the application layer of the FaaS model, thus maximizing the benefits from a transition to cloud-related environments. Aspects under consideration include the ability to add diverse annotations to an application, ability to visually design and compose a FaaS based application structure, while reusing existing templates and functionalities out of the box, as well as packaging functionalities for minimizing the knowledge barrier for incorporation and deployment along with the service graph by the cloud service adopters. The resulting description should be mapped and translated to a specification understood by the platform layer (WP4), while offering directives and annotations that can be used by WP4 and 5 in order to better adapt to application needs (Figure 1).

The purpose of the deliverable is to present the PHYSICS Cloud Design Environment for FaaS, a framework for enabling easier application workflow creation and adaptation to the FaaS model. The PHYSICS environment encapsulates the widely used (in the IoT domain) Node-RED web-based function framework for event driven applications, coupled with a back-end system that undertakes the preparation of the provided code and functionality for registration and deployment to a target FaaS platform (based on Openwhisk). The goal of the environment is to provide the following contributions:

- User-friendly visual way of creating functions and linking them together in workflows, exploiting a palette of existing functionality from the Node-RED environment as well as function orchestration abilities.
- Provide a set of functionalities in the form of patterns (subflows), that may aid the developer in the adaptation to the FaaS paradigm, used directly in a drag and drop manner in the workflow.
- Support a wide variety of features through a modular DevOps process, including the ability to execute as one Openwhisk function (based on the Node-RED runtime) for an entire flow of functions, as an orchestrator function (orchestrating other deployed functions), or as a combination of the two, while automatically generating the deployable artefact images as well as embedding more complicated processes (involving backend platform services) in a seamless manner (like in the case of the performance pipeline).
- Include diverse annotations as guidelines to other management components down the stack, enabling extended options such as function placement, preferences in scheduling, inclusion of an accompanying service component, elasticity based on a number of parameters or factors etc.

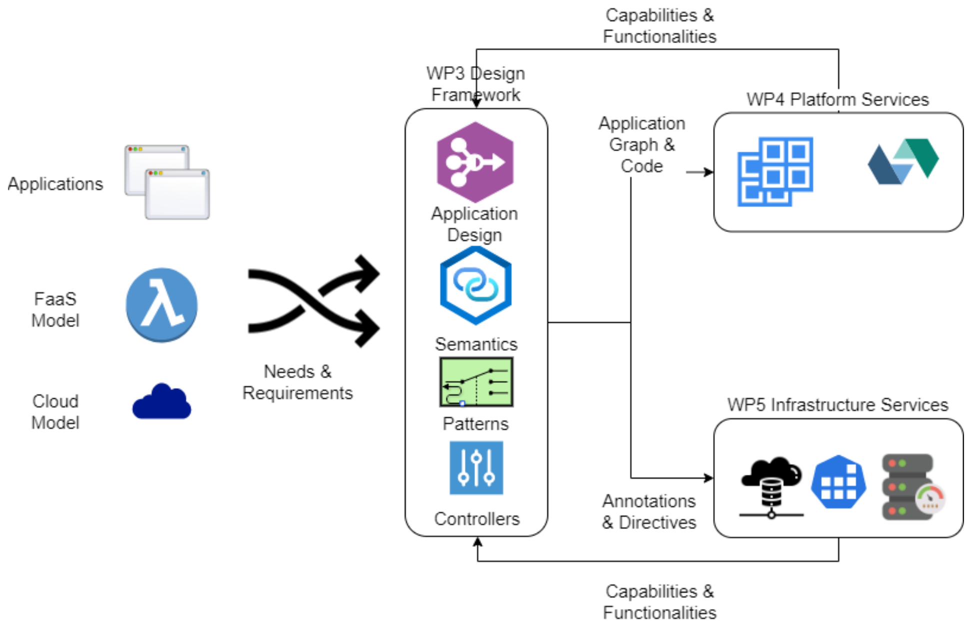


Figure 1: Positioning of WP3 related tools in PHYSICS

Through the aforementioned features, application creation, adaptation and migration to the FaaS platform can be significantly enhanced, limiting the learning curve and development time, as well as runtime operation needs.

1.2. Insights from other Tasks and Deliverables

1.2.1 Insights from the Use Case Applications

Especially for WP3, since it is the main entry point of the PHYSICS platform, the interaction with the Use Case applications and the respective developers has been extensive. From as early as M5, storyboard creation and scenario drafting were performed in a co-creative manner, in order to help the transition to a more function-oriented implementation (more details are included in D6.3). Furthermore, by investigating the application functionality and scope, WP3 has proposed a set of patterns that were anticipated to yield benefits for them. As early as M10 the specific example patterns had been presented in a relevant UC demo workshop that incorporated 8 separate example processes utilizing provided Node-RED flows for a series of aspects like design of a flow in the Design Environment, inclusion of a parallelization pattern (inspired from the needs of the Smart Agriculture UC), an arbitrary shell script and relevant docker image inclusion (both as an executable and as a semantic description for importing legacy code), privacy and security flows etc. Generic examples of Node-RED development as well as Openwhisk function creation in Node-RED were also demonstrated. Further patterns were designed and implemented in the following months following relevant discussions in regular meetings. During the second iteration of the project, the feedback from the use cases has led to the development of new patterns (e.g. the High Availability Router one) as well as enhancements in others (e.g. the enhanced monitor outputs for function level errors).

1.2.2 Insights from the main technical tasks

With relation to the main technical tasks, the overall PHYSICS architecture plays a central role as input, as well as the relevant requirements posed in D2.2. However, the layers following WP3 (i.e. WP4 and 5) are also considered as a source of input, with relation to the capabilities offered to the applications, since these capabilities need to be exposed to the developer. The latter would need to select and specify the relevant options, as means and directives for influencing further operations down the PHYSICS stack. An example of such a process was the drafting of the annotation list and the semantic models, the definition of which was, in any case, an open and continuous process.

1.3. Structure

The document is structured as follows: Section 2 presents the main architecture and building blocks of the PHYSICS Design Environment, as well as the rationale of the execution modes and build processes. Section 3 presents the Application side semantic block, including the different needed annotations, the way these are imported and processed by the Design Environment before being forwarded to the platform level. Section 4 provides the rationale of the patterns' implementations in PHYSICS, including details on their implementation, variations, examples of usage and experimentation. Section 5 presents a gamification and training server creation approach, that aims to make PHYSICS training more attractive and interesting for the end users. Section 6 documents the approach on the adaptive application elasticity controllers while Section 7 concludes the deliverable.

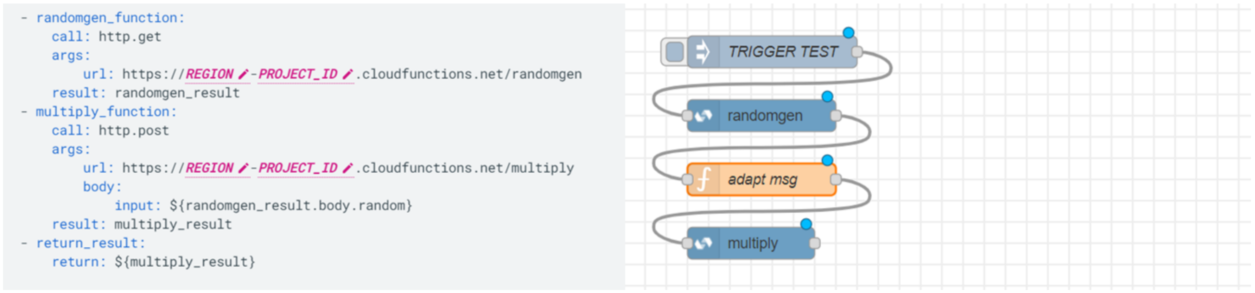
2. PHYSICS Design Environment Core

2.1. Introduction-Scope

As mentioned in the Introduction, the aim of PHYSICS is to provide a Design and Development environment to aid application migration to the FaaS model in a more abstract manner. This is based on the fact that current environments include a significant number of limitations, as well as the fact that the functional programming style of FaaS creates significant difficulties to typical application developers. A major drawback of current FaaS platforms is the availability of tools related to the deployment and function reuse [7]. Function composition however can be used in order to provide more complex groups of functions through the interaction of simpler ones, provided that there are according orchestration and grouping capabilities [8]. The notion of patterns can in this case be useful in order to group appropriate functions that intend to solve a specific problem (e.g. AI training and optimization [9]).

From the main open source platforms, Openwhisk is the only platform that has a native functionality in place (sequence operator, i.e. the ability to declare function chains in which the output of a function is passed to the next one) at the runtime level [10], although it only supports simple serial execution of functions. A further extension involves the IBM Composer [11], which implements a set of orchestration primitives, although in a javascript library-based form [12]. The functionality is also ported to IBM Cloud Functions, the commercial cloud solution from IBM, one of the main contributors of the Openwhisk project. OpenFaaS has an external plugin (FaaS-flow [13]) for declaring sequences of functions in a textual, code-like manner (including more complex workflow primitives). The orchestrating logic is also executed as a function. Kubeflow [14] has the pipeline definition language as well as an editor extension (Elyra) through which pipelines can be visually defined. However, the concept of workflow in this case is that of a static sequence of operations (with information passing from one step to another via intermediate cloud object storage files). Therefore, it lacks the dynamicity and the abilities of an actual runtime environment correlating the passing of arguments between functions and writing arbitrary orchestration code.

From the main cloud vendors, AWS Step Functions includes a visual programming style, as well as a number of further operators (including state management), however, it is directly tied to the AWS services and thus is an option that increases vendor lock-in. Google Cloud Functions supports the creation of workflows through relevant YAML files and syntax [15], assuming that the functions have already been deployed. In Figure 2, the comparison between a GCF-based syntax (left) and an equivalent Node-RED flow (right) implementing the same functionality indicates the differences in the usability of the two forms, even for simple flows. In the Node-RED flow, only the ready-made client nodes [16] for the FaaS platform (in this case, Openwhisk) are needed (with the name of the function to use) plus a small custom function for adapting message fields. Going to even more complex workflows, including a large number of functions and diverse connections between them, becomes tiresome and error-prone for the developer in the YAML format, as well as in the other text/code-based ones such as IBM Composer or FaaS-flow. Other comparisons performed [12] indicate drawbacks of workflow management abilities among cloud providers in the case of fork-join primitives, with the exception of IBM Composer.



```
- randomgen_function:
    call: http.get
    args:
        url: https://REGION-PROJECT_ID.cloudfunctions.net/randomgen
    result: randomgen_result
- multiply_function:
    call: http.post
    args:
        url: https://REGION-PROJECT_ID.cloudfunctions.net/multiply
        body:
            input: ${randomgen_result.body.random}
    result: multiply_result
- return_result:
    return: ${multiply_result}
```

Figure 2: Comparison between a text-based workflow syntax (GCF) and Node-RED flow

Apache Airflow is a workflow design and management tool. It allows for planning, scheduling and automating the flow of data through nodes. Graphs of Directed Acyclic Graphs (DAGs) type represent the direction of the data, while the output of one node (task) is usually the input of the next node. Airflow having complex operators cannot be used as a generic function editor. One very interesting work is TriggerFlow [17]. In this case, different workflow primitives are offered, as well as eventing mechanisms in order to regulate the execution. The main difference of our work is that in our case the environment can be used for both function and workflow creation. Furthermore, due to the usage of Node-RED, we are able to import ready-made functionality in the form of reusable subflows and nodes, either at the workflow structure or at the core functionality level. Another difference is the visual support for the workflow creation.

What can be observed from the investigation of the related work is that a number of solutions exist, but primarily for the definition of workflows either without a proper runtime mechanism, or through code-level libraries/YAML files difficult to write for flows containing many functions and complex connections. They have very little support for function re-use (especially function groups reuse) and no support for annotations that can somehow be propagated to lower levels of management. There is no single environment that can help the developer code, visually design, test and deploy the functions while also acting as an orchestrator during runtime.

For this purpose, the main editing environment selected in the PHYSICS project is Node-RED. Node-RED is a very popular tool in the context of IoT, for building event driven, functional programming style applications. From that aspect, Node-RED portrays a number of significant advantages in terms of the aforementioned features:

- A visual server environment for wiring and deploying together functions into complex and arbitrary workflow structures, without the aforementioned limitations in terms of types of wirings or implemented orchestration patterns. The programming style of the environment follows a functional, event driven programming approach, fitting to the baseline FaaS model.
- A more augmented runtime environment can be used as the basis of execution, aiding in creating FaaS functions (or actions) that internally consist of multiple Node-RED functions. This enhances the development process (due to the runtime abilities for message tracking and manipulation) as well as enables local flow testing directly in the Node-RED server

editor, skipping the costly and time consuming actual FaaS deployment. A large number of errors (e.g. typical Javascript syntax or logic errors, message handling and field setting, etc.) can be handled at this level prior to the actual deployment. Furthermore, the inclusion of more than one Node-RED functions in a FaaS function output would result in less needs for containers during execution, therefore less back-end contention.

- Abundance of ready-made nodes [18], especially from the IoT domain but also for general systems, exploiting the generic npm repository of Node.js [19], one of the largest open source repositories.
- Ability to group functions as subflows, aiding in code reusability, sharing and function management, workflow simplification and abstraction.
- Ability to treat the workflow definition as a meta-specification layer. Given that Node-RED has its own simple workflow definition schema, this can act as a meta-specification from which translations to different provider syntaxes can be performed in order to mitigate vendor lock-in.

Node-RED typically runs as a server, so the main question is: can it be used initially as an editor and testing environment for creating functions and workflows that are afterwards deployed on a typical FaaS platform? For this purpose, further backend services and functionalities need to be offered. Furthermore, can it also be used for orchestrating functions and in what way? And what happens in this case with common issues like double billing in the serverless trilemma [20], i.e. the principle in which no function should wait (and get billed) while waiting for another function to complete?

It is necessary to stress that in the context of PHYSICS, there are two levels of orchestration. Initially, at the application level, orchestration (or in other words function choreography) implies the coordination between function execution as well as the passing of arguments across the function flow. In essence, it is the way the functions are linked together in order to form the business logic of the application. On the other hand, at the platform level, orchestration refers to the coordination of the deployment, monitoring and runtime management of the resources (FaaS platforms, container clusters, networks, etc.) needed to execute the application based on the placement decisions, as well as translation of the app structure to the specification of the FaaS platform. Application level orchestration is included in this document (and is in scope of WP3) whereas platform level orchestration is included in D4.1 (and is in scope of WP4 and mainly T4.5).

2.2. Relation to project requirements

With relation to requirements expressed in D2.3, the PHYSICS Design Environment is affected by the following ones:

- Req-3.1-WorkflowDef
    - Ability to define a workflow of functions that implement the business logic
- Req-3.1-MultiTenancy
    - Ability to distinguish between different branches/users in the environment
- Req-3.1-LogsService
    - Ability to have concentrated logs from involved microservices in one location
- Req-3.1-BuildsHistory
    - Ability to have a record of the builds performed in the environment

- Req-3.1-SupportedRuntimes
    - Ability to support multiple runtimes (e.g. nodejs, python, Node-RED)
- Req-3.1-CustomDockerImages
    - Ability to define (or create) an arbitrary docker image as the actionable artefact of a function

2.3. Application Creation Use Cases

The main use cases that have been foreseen in D2.4 are included in the following figures (Figure 3, Figure 4) in order to give an overview of the functionalities needed, in the two main operations, creating of an application through defining or embedding application logic, and testing the designed application. In the following sections, refinement and increased detail on parts of the use cases are also presented.

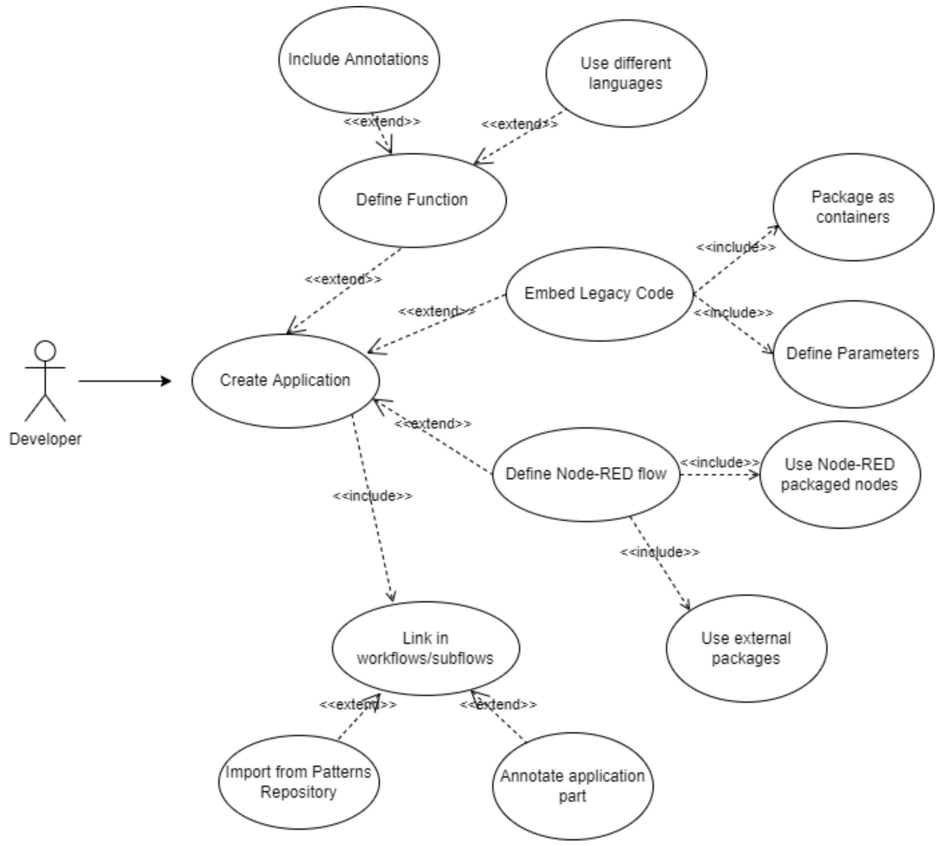


Figure 3: Create Application UC from D2.4

2.4. Design Environment Overview

The overview of the PHYSICS Design and Development Environment appears in *Figure 5*. The main editing environment is an embedded container of a Node-RED server, with an enriched palette of nodes (including the built-in ones, additions from the Node-RED community repository as well as extensions provided by the PHYSICS environment). The PHYSICS editor extensions include either ready-made subflows that are built for a specific purpose (see section 4 for details) or semantic annotation nodes (see section 3 for details) and any other node from the Node-RED ecosystem that may be useful. All these elements can be dragged and dropped directly in an arbitrary workflow in order to augment its functionality. During this development, the developer can also use semantic annotations that give a wide number of possibilities in order to affect various aspects of a flow or function execution. As an example, sizing of the function (in terms of memory) can be dictated,, while other considerations may include the deployment target (e.g. function or flow A needs to be deployed on Edge B).

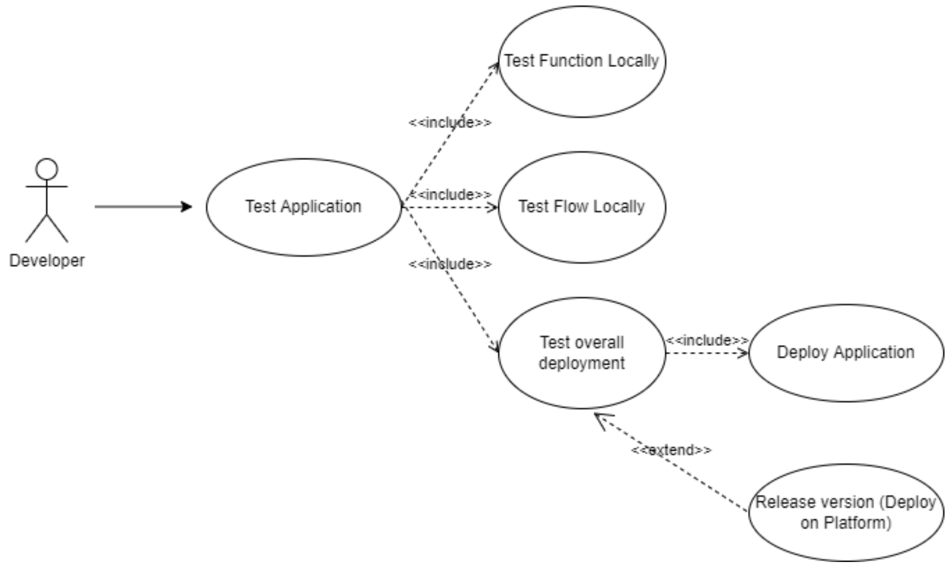


Figure 4: Test Application UC from D2.4

Once the developer finishes the development of the flows, they can move to the Design and Control UI, in which they select which flows to prepare for instantiation. From then on, the process is orchestrated by this component, which contacts the Serverless Function Generator for extracting the code from the Node-RED environment and calls a relevant DevOps process implemented as a Jenkins pipeline. Different pipelines are supported (e.g. create function from flow or from imported image) in order to adapt to the needed steps in each case and generate the final deployable artefact (code bundle, image etc). This process is supported by relevant repository and docker image registry services.

The flow also passes through the Semantic Extractor component, which extracts the declared annotations in the selected flows and maps them to ontological concepts defined in the PHYSICS Ontology. The resulting data structure is derived from the JSON specification of the typical Node-RED flow, and the declarations in the overall application graph. Thus, the application graph is created as a representation of the flow and the associated annotations encoded as semantic triples in JSON-LD form. The triples are stored for later use (during deployment) in the Reasoning Framework and Inference Engine (WP4). Upon finalization of this process, the annotated app graph is forwarded to the PHYSICS platform management layer, which includes functionality to process the graph and register the according functions and native sequences to the FaaS platform. It also maintains the relevant annotations (e.g., mapped to Kubernetes keywords or Openwhisk options) in order to dictate the expressed developer needs (in terms of deployment, function management, affinity, sizing, etc.). Following, details on the different execution patterns are presented.

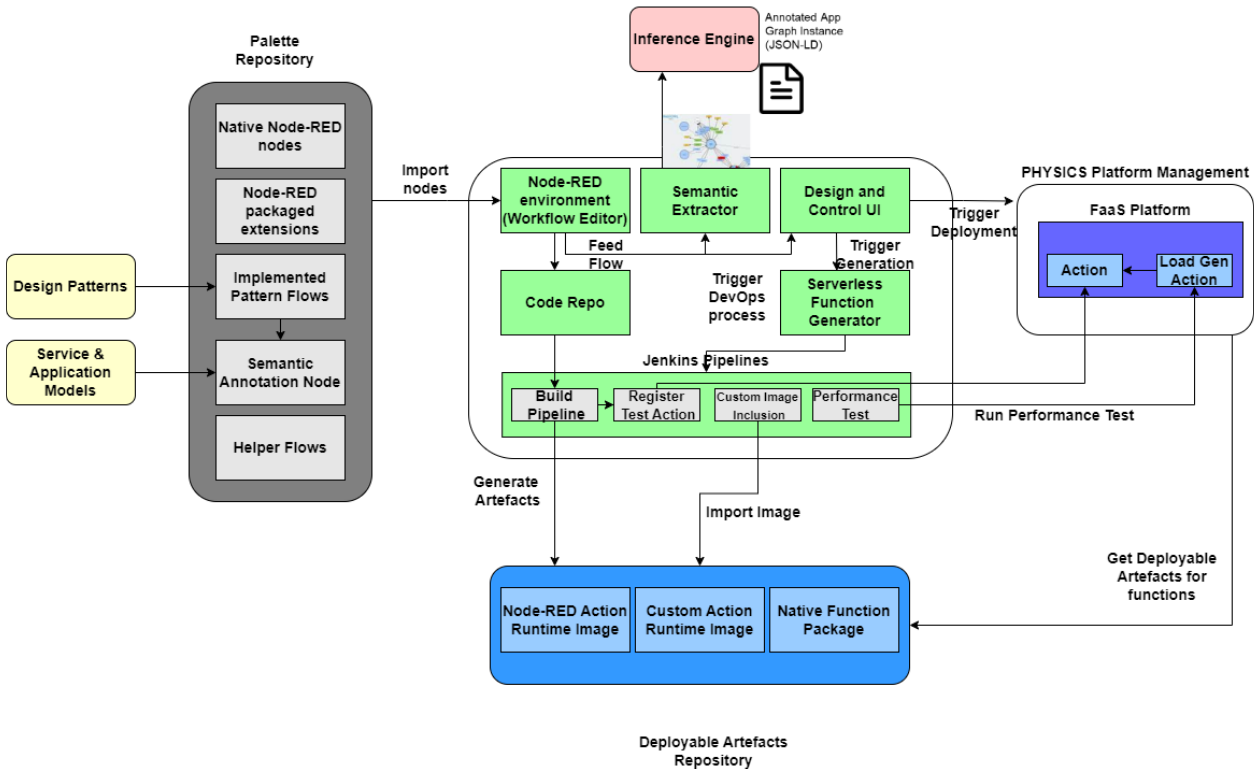


Figure 5: Overview of the PHYSICS Design Environment

2.4.1. Custom Runtimes and Execution/Orchestration modes

In order to enable multiple manners and environments of execution the Design environment has defined different possibilities.

Custom Runtimes Ability

The DE comes with a baseline image template that includes the main dependencies (Node-RED, palette of PHYSICS-provided flows and patterns etc). However in many cases the application developers may need to add their own environments, scripts etc. For this reason, the PHYSICS DE gives them the ability to change the baseline image Dockerfile, so that they can install anything they need such as Python frameworks for AI, relevant scripts or other tools (Figure 6). Thus they can use the Node-RED flows as the main interface to Openwhisk or as a generic orchestrator, whereas the main logic resides in their imported code.

A second option was added in the second iteration of the project, through the ability to upload a custom image. This image can be based on any relevant template however it must also contain the necessary interface so that it can be executed by Openwhisk.

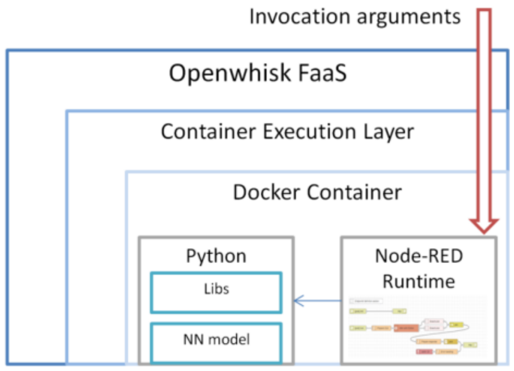


Figure 6: Example Customization of the Function Execution Runtime

<u>Node-RED flow embedded in a Node-RED container for execution as service or function</u>

One of the main features of PHYSICS is the ability to write any arbitrary workflow in Node-RED, exploiting any type of node that can be then executed either as a service (typical Node-RED execution) or as a function. This process is split into two parts. Initially, an in-flow support is provided in the form of a skeleton flow that will be detailed according to the pattern. Inside this flow, any node-RED packaged node, as well as npm-based libraries, can be exploited, thus leading to the inclusion of a code base with extensive capabilities. In this mode, the developer can wire nodes in more versatile manners (also supported by relevant patterns of Section 4), since their execution is performed within the Node-RED runtime of the Openwhisk action container, not limited by any workflow specification limitation of the FaaS platform.

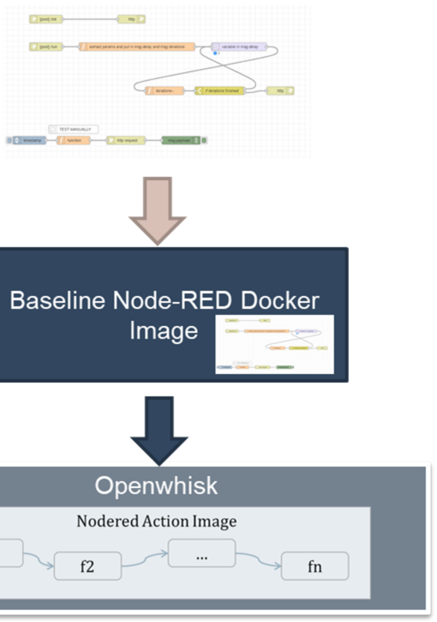


Figure 7: Node-RED flow to Node-RED runtime Action Image

An extra benefit of this case is that it solves the issue of splitting the function logic packaging into many different separate functions. Managing hundreds of functions in the context of one application has been identified as one of the obstacles in FaaS development. Furthermore, grouping many lightweight functions in one actionable function image, based on the Node-RED runtime, results in much fewer container overheads due to the fact that no separate container needs to be raised for each individual and potentially small function. However, relevant DevOps processes need to be in place in order to generate the respective deployable image, including all necessary dependencies, settings, etc.

Orchestrator Flows for complex function workflows

As mentioned in the introduction of this section, due to the limitations of the reviewed environments and the inherent ability of Node-RED to act as an orchestrator, by passing messages that trigger functions, one key feature is to use a Node-RED flow in order to orchestrate complex function wiring and workflow primitives. In this case, the execution of an orchestrating flow can also be performed as a function. This has the extra benefit that the specific orchestration definition, based on the Node-RED workflow meta-specification, can be afterward translated and executed potentially on multiple providers with limited changes, by adapting to the underlying workflow specifications used by each provider. This directly leads to a decreased vendor lock-in for the case of FaaS.

The created orchestration flow can be used either as a function or as a service. However, in the service mode one is constrained by the scalability of a single Node-RED environment used to orchestrate many executions. Furthermore, they get billed for the constantly running orchestrator service. These two arguments are the most commonly used for moving a functionality to a serverless paradigm in any case. The two potential orchestration ways appear in Figure 8. It needs to be stressed that the inner invoked actions may be Node-RED-based actions or actions following any other runtime.

Despite the aforementioned advantages, usage of a Node-RED flow acting as an orchestrator and executed as a function would result in a double billing issue [20] (i.e. the fact that a function should not wait for another function to execute), since the orchestrator function would need to wait for the orchestrated functions to finish. However, there are various arguments for exploiting this approach, presented in the next paragraphs.

Argumentation against the double billing principle in Orchestrator Functions

The main arguments against considering orchestration as part of the double billing principle are the following:

- Whether the orchestrating function would be billable is primarily a business model decision and should be separated from the technical ability. I.e., the provider might choose to offer such functionality for free or with a different cost model than function execution time. If the respective provider wants to gain a competitive advantage and give the ability to their customers to easily create and deploy arbitrary workflows, they could follow this approach.

In a similar manner, many other types of services (like usage of dashboards) are not separately billed by the providers, even though there are available REST APIs for every possible action performed through the dashboard. The aforementioned services are offered primarily as a more user-friendly means to create and manage cloud resources, thus leading to increased domain uptake. Similarly, in the serverless domain specifically, there is no charge for e.g., the gateway service that is running and listens for events that trigger functions.

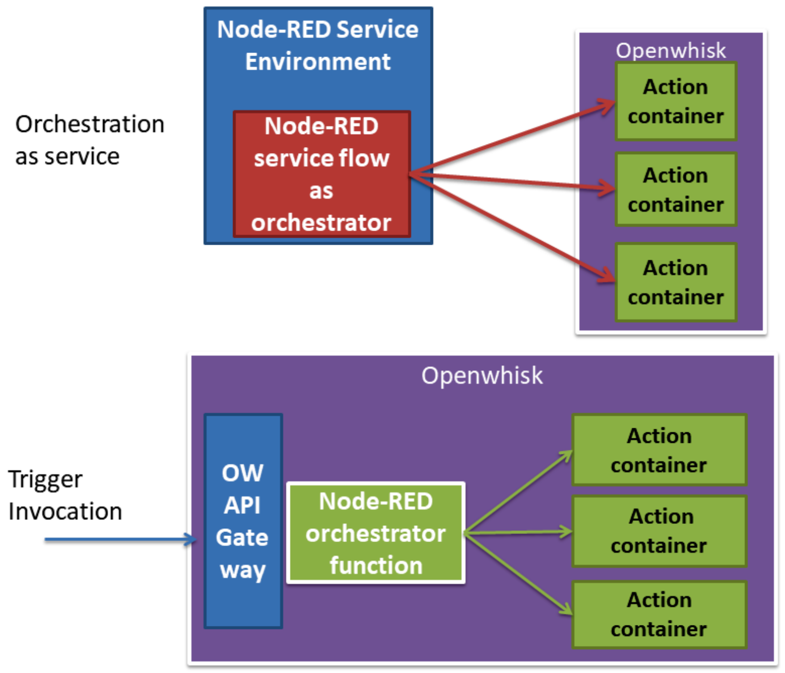


Figure 8: Different Means of Node-RED Orchestrator Execution

- Usage of such an approach would alleviate the need for a scalable orchestrator, a daunting issue on its own, since each separate orchestrated flow would be in its separate function execution. This setup is by default scalable. Furthermore, the existence of an actual orchestrating runtime (Node-RED runtime) would mean that any workflow primitive could be applied based on custom logic and appropriate message handling.
- 82% of serverless applications use 5 or fewer functions and only 31% of them include workflows [21]. Even these are mostly simply structured, small, and short-lived. Having the ability to create more complex application workflows would be in favour of the providers in the long run, since more complexity in the workflows would directly incur higher number of included functions and according invocations.
- If we consider that a function should not wait for an operation (i.e. blocking call) since in that time it is billed without being useful, why do we accept blocking calls in the most typical serverless use cases, e.g. the retrieval of a data object from an object storage prior to

feeding it to an AI image detection function, and not go for alternative near-data processing models [22].

In a nutshell, we consider the fact that the double billing principle can aid when writing single functions. It leads to a more asynchronous style of programming, however it should not be applied at the function workflow orchestration level, given the constrained current abilities of FaaS platforms in this feature.
Following, further information is provided on the main components and processes of the Design Environment.

2.5. Design Environment App

The PHYSICS Design Environment is a web application, which embeds the Node-Red environment and extends it with features for communication with other PHYSICS components such as the deployment preparation, the semantic extraction and the deployment process itself.

2.5.1. Component Design and Processes

The Design environment allows the developer to perform following actions:

- Develop Flows in Node-RED: The embedded Node-RED environment tab in the PHYSICS Design Environment (Figure 9) allows the developer to use it the same way as the standalone Node-RED application/server.

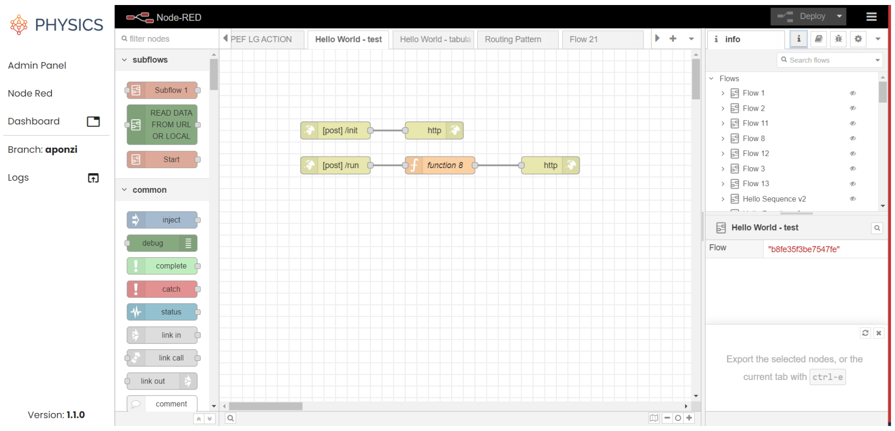
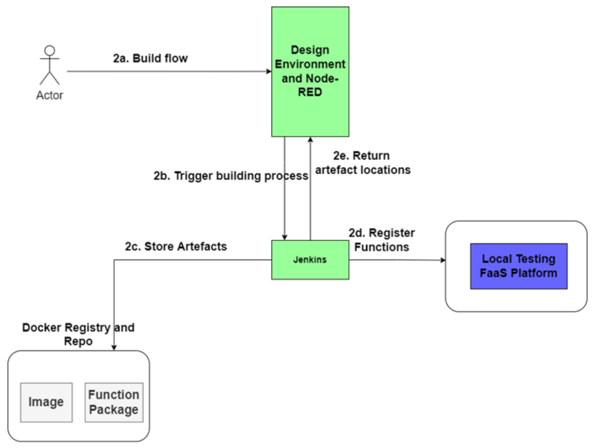
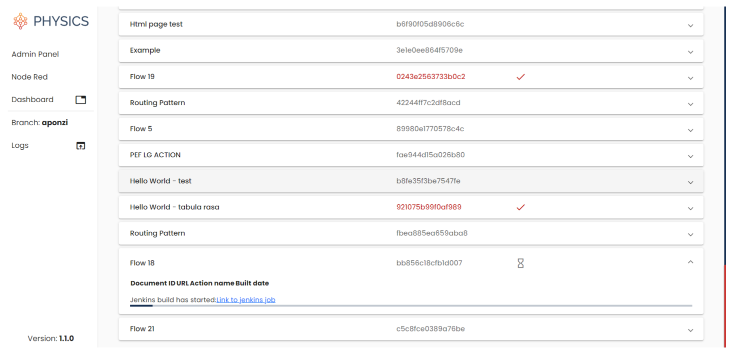


Figure 9: Embedded Node-RED environment in PHYSICS Design Environment for flow creation

- Build and deploy to test environment: The Developer can choose the flow (Figure 10), which will be extracted from Node-RED, uploaded to the object bucket and used by a Jenkins Job to create the Docker Image with a Node-RED environment containing only the chosen flow, which will be later deployed as FaaS to the test environment.


Actor
2a. Build flow
Design Environment and Node-RED
2b. Trigger building process
2e. Return artefact locations
2c. Store Artefacts
Jenkins
2d. Register Functions
Local Testing FaaS Platform
Docker Registry and Repo
Image
Function Package



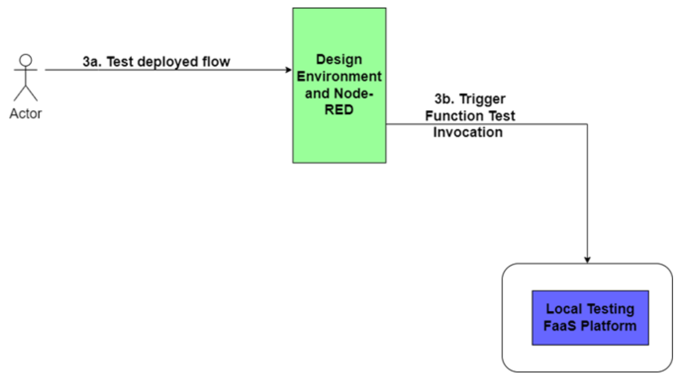
Html page test	b6f90f05d8906c6c	
Example	3e1e0ee864f5709e	
Flow 19	0243e2563733b0c2	✓
Routing Pattern	42244ff7c2df8acd	
Flow 5	89980e1770578c4c	
PEF LG ACTION	fae944d15a026b80	
Hello World - test	b8fe35f3be7547fe	
Hello World - tabula rasa	921075b99f0af989	✓
Routing Pattern	fbea885ea659aba8	
Flow 18	bb856c18cfb1d007	⌛
Flow 21	c5c8fce0389a76be	

Document ID URL Action name Built date

Jenkins build has started:Link to jenkins job

Figure 10: Design Environment Build Process and Relevant UI

- Test deployed flows: All already built flows are also available as OpenWhisk actions in the test environment, which can be triggered from the Design Environment in order to test the solution (Figure 11). In that case, the developer may use the Openwhisk Action client node inside Node-RED to trigger an invocation of the function on the target Openwhisk installation. The user can also test the action with a performance evaluation on the local environment or in a remote cluster.


Actor
3a. Test deployed flow
Design Environment and Node-RED
3b. Trigger Function Test Invocation
Local Testing FaaS Platform

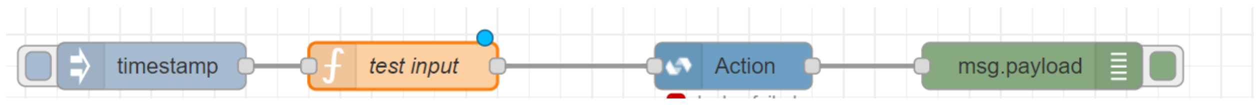


Figure 11: Design Environment Test Process and Example Node-RED flow

- Create Application Graphs: Developers can also group developed flows into application graphs, start the same building process as for the test environment if needed and store them in the Reasoning Framework of WP4 as triples (Figure 12 and Figure 13). More information is included in Section 3.7, linked to the PHYSICS ontology and annotation mechanisms.

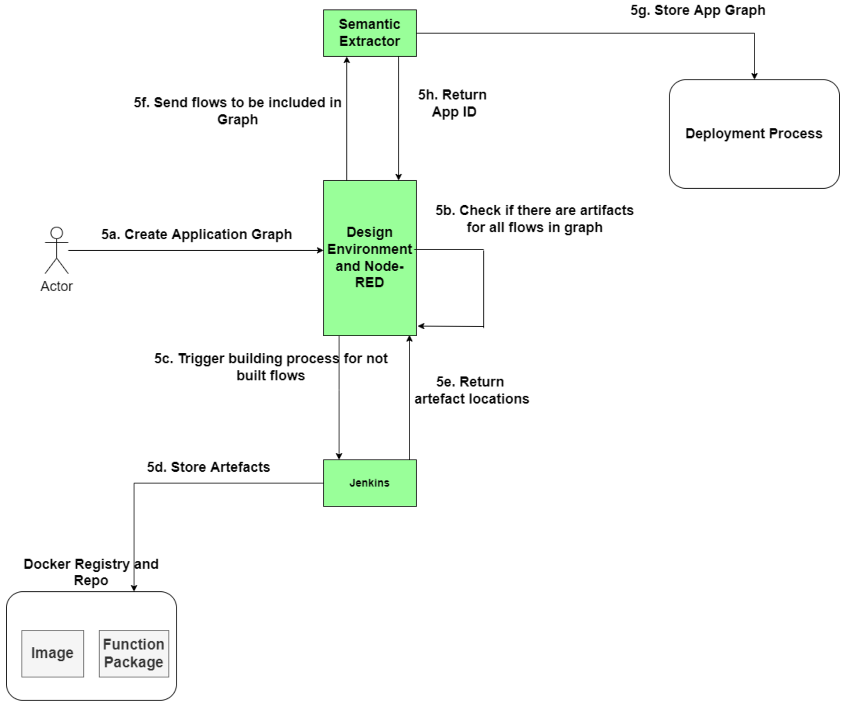
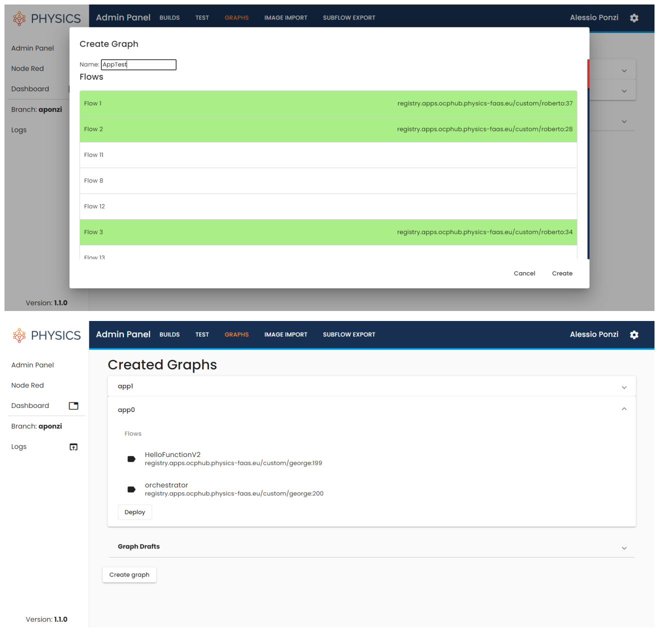

Semantic Extractor
5g. Store App Graph
Deployment Process
5f. Send flows to be included in Graph
5h. Return App ID
5a. Create Application Graph
Actor
Design Environment and Node-RED
5b. Check if there are artifacts for all flows in graph
5c. Trigger building process for not built flows
5e. Return artefact locations
5d. Store Artefacts
Jenkins
Docker Registry and Repo
Image
Function Package

Figure 12: Design Environment Create Application Graph Process



Create Graph

Name: AppTest

Flows

Flow 1	registry.apps.ocphub.physics-faas.eu/custom/roberto:37
Flow 2	registry.apps.ocphub.physics-faas.eu/custom/roberto:28
Flow 11	
Flow 8	
Flow 12	
Flow 3	registry.apps.ocphub.physics-faas.eu/custom/roberto:34
Flow 13	

Branch: **aponzi**

Version: **1.1.0**

Created Graphs

app1

app0

Flows

HelloFunctionV2
registry.apps.ocphub.physics-faas.eu/custom/george:199

orchestrator
registry.apps.ocphub.physics-faas.eu/custom/george:200

Graph Drafts

Branch: **aponzi**

Version: **1.1.0**

Figure 13: Design Environment Create Graphs UIs

- Deploy Application Graphs: The Developer can trigger the whole app deployment, which will send the information about flows and built artifacts to trigger the right deployment process (Figure 14).

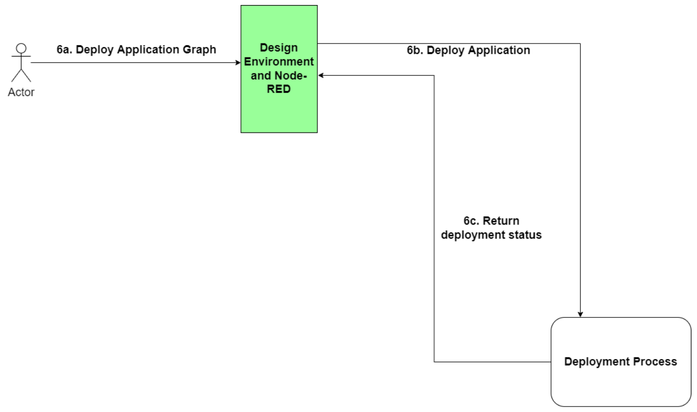

Actor
6a. Deploy Application Graph
Design Environment and Node-RED
6b. Deploy Application
6c. Return deployment status
Deployment Process

Figure 14: Design Environment Deploy Application Process

- Export Created Subflow: Developers have the possibility to export custom subflows as packages, ready to be deployed on npm and Node-RED package registry (Process in Figure 15 and UI in Figure 16).

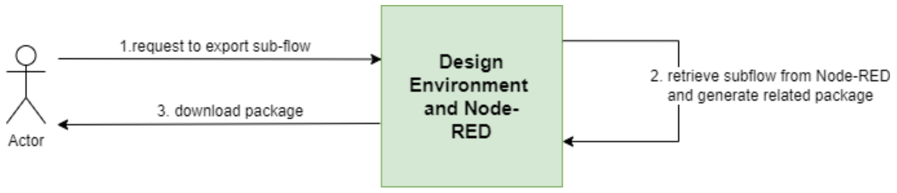
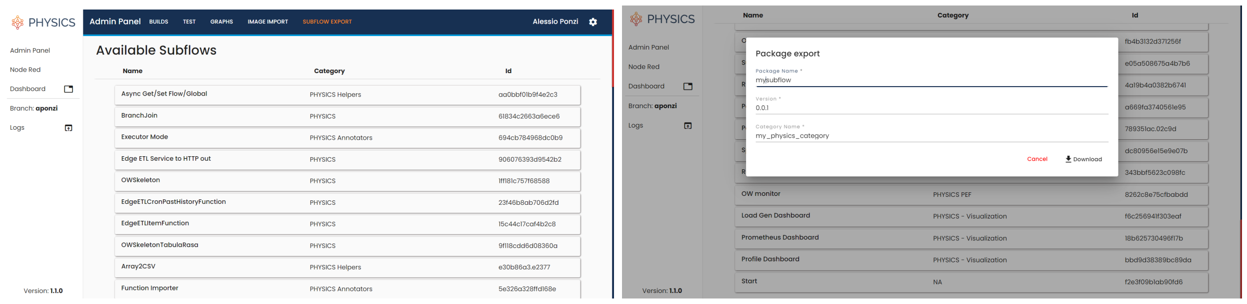

Actor
1.request to export sub-flow
3. download package
Design Environment and Node-RED
2. retrieve subflow from Node-RED and generate related package

Figure 15: Subflow Export Process

- Performance Visualization: The solution provides a fully customizable dashboard, where the Developer can view performance pipeline results and benchmarks for every function as well as monitor of the OpenWhisk cluster (Process in Figure 17 and UI in Figure 18).



Available Subflows

Name	Category	Id
Async Get/Set Flow/Global	PHYSICS Helpers	aa0bbf01b9f4e2c3
BranchJoin	PHYSICS	61834c2663a6ece6
Executor Mode	PHYSICS Annotators	694cb784968dc0b9
Edge ETL Service to HTTP out	PHYSICS	906076393d9542b2
OWSkeleton	PHYSICS	1ff181c757f68588
EdgeETLCronPastHistoryFunction	PHYSICS	23f46b8ab706d2fd
EdgeETLItemFunction	PHYSICS	15c44c17caf4b2c8
OWSkeletonTabulaRasa	PHYSICS	9f118cdd6d08360a
Array2CSV	PHYSICS Helpers	e30b86a3.e2377
Function Importer	PHYSICS Annotators	5e326a328ffd168e

Package export

Package Name *
mysubflow

Version *
0.0.1

Category Name *
my_physics_category

Cancel Download

Name	Category	Id
O		fb4b3132d371256f
S		e05a508675a4b7b6
R		4a19b4a0382b6741
P		a669fa3740561e95
P		789351ac.02c9d
S		dc80956e15e9e07b
R		343bbf5623c098fc
OW monitor	PHYSICS PEF	8262c8e75cfbabdd
Load Gen Dashboard	PHYSICS - Visualization	f6c256941f303eaf
Prometheus Dashboard	PHYSICS - Visualization	18b625730496f17b
Profile Dashboard	PHYSICS - Visualization	bbd9d38389bc89da
Start	NA	f2e3f09b1ab90fd6

Figure 16: Subflow Export Process in the UI

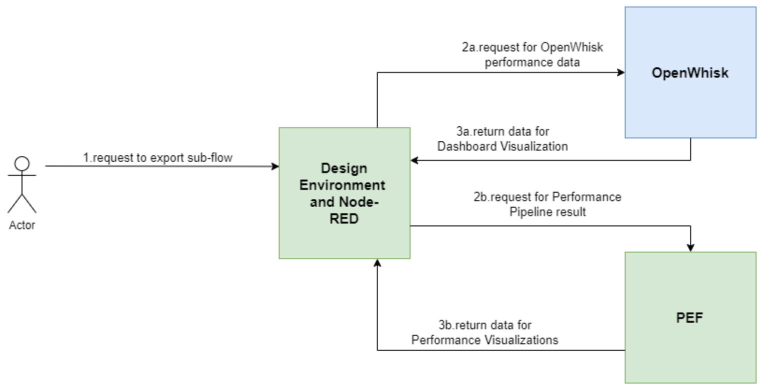

2a.request for OpenWhisk performance data
OpenWhisk
3a.return data for Dashboard Visualization
1.request to export sub-flow
Design Environment and Node-RED
Actor
2b.request for Performance Pipeline result
PEF
3b.return data for Performance Visualizations

Figure 17: Performance Visualization Process Diagram

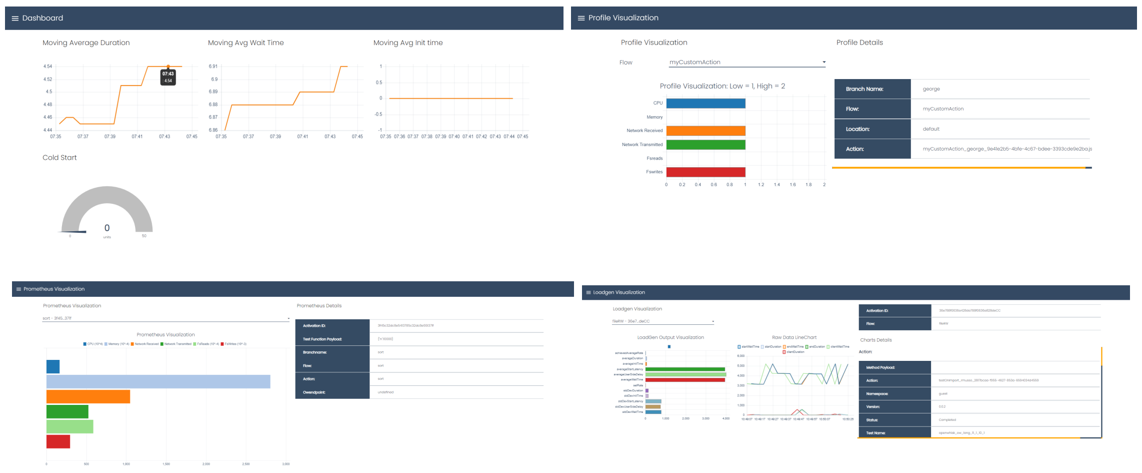


Figure 18: Performance Visualization Information in the DE

- Import image: The solution offers the possibility (Figure 20) to import into the Physics environment an application image from an external or Physics repository, be this public or private, so long as the application is compatible with the Openwhisk API specification. In the case of a private repository the user and password must be provided during the import flow, which will be stored encrypted in Jenkins. The imported image is also available as OpenWhisk actions in the test environment, executable in the test section of the Design Environment. After the request, the user can monitor the state of the import directly from the import page, where it reports the history of all imported images, with the action name associated, the date of the import request and the state. The according process diagram and interactions between the existing components appears in Figure 19. When the user requests for an import the DE calls a specific Jenkins pipeline named "load-custom-dockerimage". The DE interacts directly with the REST API exposed by Jenkins. The pipeline was created using the groovy language provided natively by Jenkins.

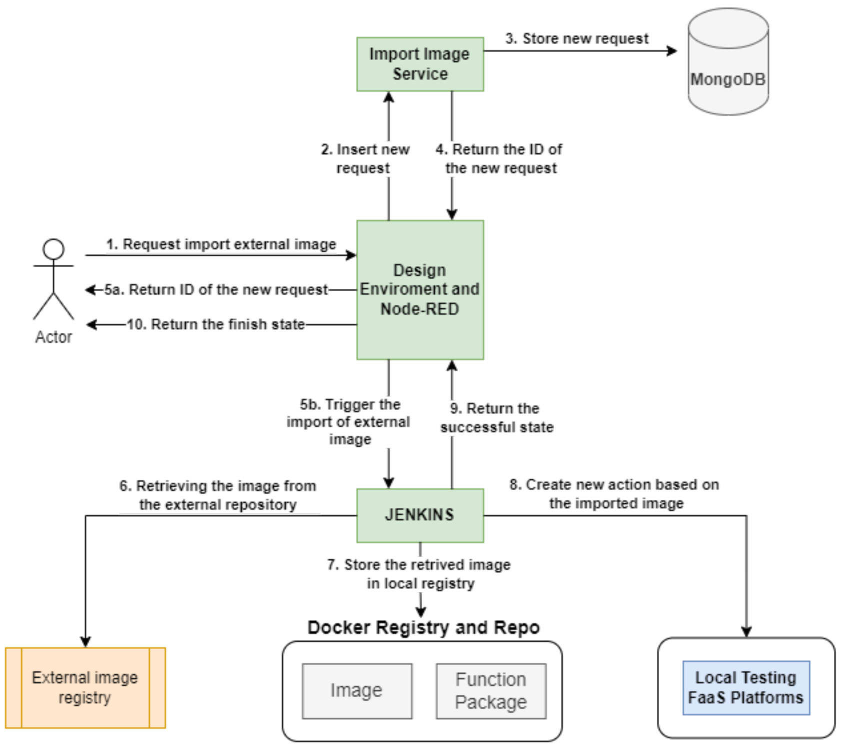
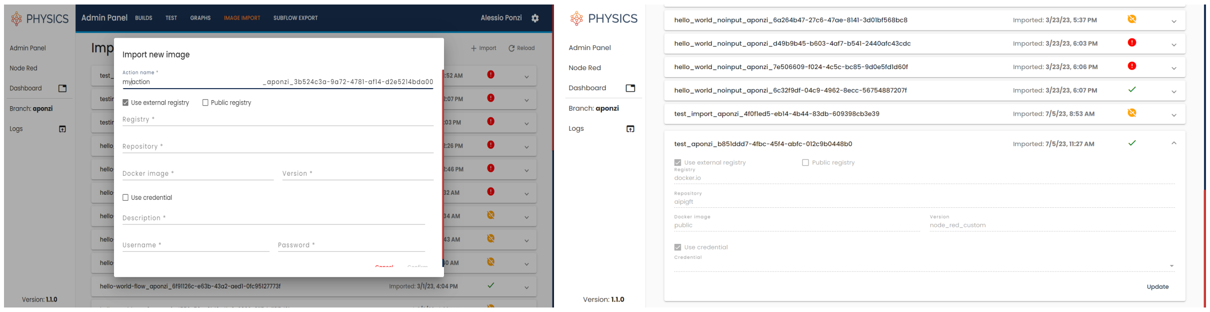

Import Image Service
3. Store new request
MongoDB
2. Insert new request
4. Return the ID of the new request
1. Request import external image
5a. Return ID of the new request
10. Return the finish state
Actor
Design Enviroment and Node-RED
5b. Trigger the import of external image
9. Return the successful state
6. Retrieving the image from the external repository
JENKINS
8. Create new action based on the imported image
7. Store the retrived image in local registry
Docker Registry and Repo
External image registry
Image
Function Package
Local Testing FaaS Platforms

Figure 19: Image import Process Diagram



Import new image

Action name * myaction _aponzi_3b524c3a-9a72-4781-af14-d2e5214bda00

☑ Use external registry ☐ Public registry

Registry *

Repository *

Docker image * Version *

☐ Use credential

Description *

Username * Password *

hello-world-flow_aponzi_6f91126c-e63b-43a2-aed1-0fc95127773f Imported: 3/1/23, 4:04 PM ✓

hello_world_noinput_aponzi_6a264b47-27c6-47ae-8141-3d01bf568bc8 Imported: 3/23/23, 5:37 PM

hello_world_noinput_aponzi_d49b9b45-b603-4af7-b541-2440afc43cdc Imported: 3/23/23, 6:03 PM

hello_world_noinput_aponzi_7e506609-f024-4c5c-bc85-9d0e5fd1d60f Imported: 3/23/23, 6:06 PM

hello_world_noinput_aponzi_6c32f9df-04c9-4962-8ecc-56754887207f Imported: 3/23/23, 6:07 PM ✓

test_import_aponzi_4f0f1ed5-eb14-4b44-83db-609398cb3e39 Imported: 7/5/23, 8:53 AM

test_aponzi_b851ddd7-4fbc-45f4-abfc-012c9b0448b0 Imported: 7/5/23, 11:27 AM ✓

☑ Use external registry ☐ Public registry

Registry: docker.io

Repository: aipigft

Docker image: public Version: node_red_custom

☑ Use credential

Credential

Figure 20: Image import User Interface

The structure of the pipeline can be divided into 4 macro areas. The first macro area in which the input parameters (Figure 21) to the pipeline are defined. These parameters include:

- The external registry boolean variable, indicates whether the image should be imported and stored in the external registry
- The public boolean variable indicates whether the image should be imported and stored in the PHYSICS registry
- The registry variable contains the reference to the registry from where to get the external docker image

- The repo variable specifies the repository, within the remote registry, where the docker image is stored
- The credid variable is the ID of the secret in Jenkins in which the credentials have been saved, in encrypted mode, to access the remote registry
- The dockerimage variable specifies a docker image to fetch
- The version variable specifies a docker image version to fetch
- The actionname variable defines the name of the action to be created in OpenWhisk associated with the imported docker image
- The old_action variable, optional field, specifies the name of an action already associated with the imported docker image
- The user variable specifies the user who has requested the import image

```
properties(
  [
    parameters(
      [
        booleanParam(name: 'externalregistry',description: 'Required if the registry is external', defaultValue: false),
        booleanParam(name: 'public',description: 'Set if external registry is public', defaultValue: false),
        string(name: 'registry', defaultValue: 'docker.io', description: 'This field is MANDATORY'),
        string(name: 'repo', defaultValue: 'unix3'),
        string(name: 'credid', defaultValue: 'dockerhub'),
        string(name: 'dockerimage', defaultValue: 'python'),
        string(name: 'version', defaultValue: '1.0'),
        string(name: 'actionname', defaultValue: ''),
        string(name: 'oldaction', defaultValue: 'empty'),
        string(name: 'user', defaultValue: '')
      ]
    )
  ]
)
```

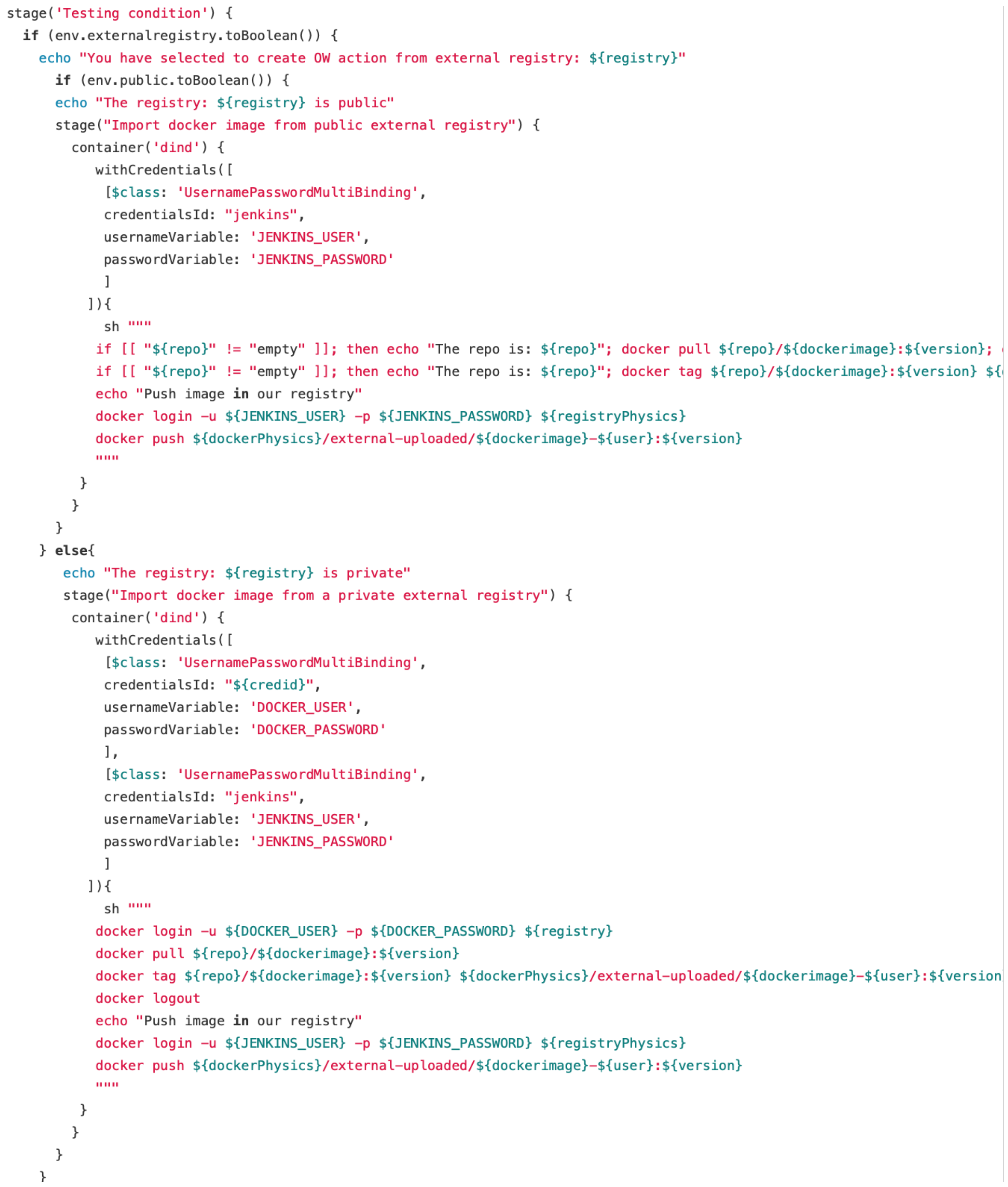
Figure 21: Import parameters for Jenkins pipeline

The second macro block defines the POD parameters(Figure 22) within OpenShift that will be used to execute the pipeline; this POD will be a Jenkins slave POD governed by the Jenkins master POD.

```
podTemplate(label: label, namespace: 'dev', yaml: """
apiVersion: v1
kind: Pod
metadata:
  labels:
    some-label: dind-agent
spec:
  serviceAccountName: jenkins
  containers:
  - name: dind
    image: docker:20.10.12-dind
    resources:
      requests:
        cpu: 100m
        memory: 256Mi
      limits:
        cpu: 500m
        memory: 1Gi
    imagePullPolicy: IfNotPresent
    tty: true
    securityContext:
      privileged: true
    volumeMounts:
      - name: docker-graph-storage
        mountPath: /var/lib/docker
  volumes:
    - name: docker-graph-storage
"""
```

Figure 22: POD parameters for Jenkins slave

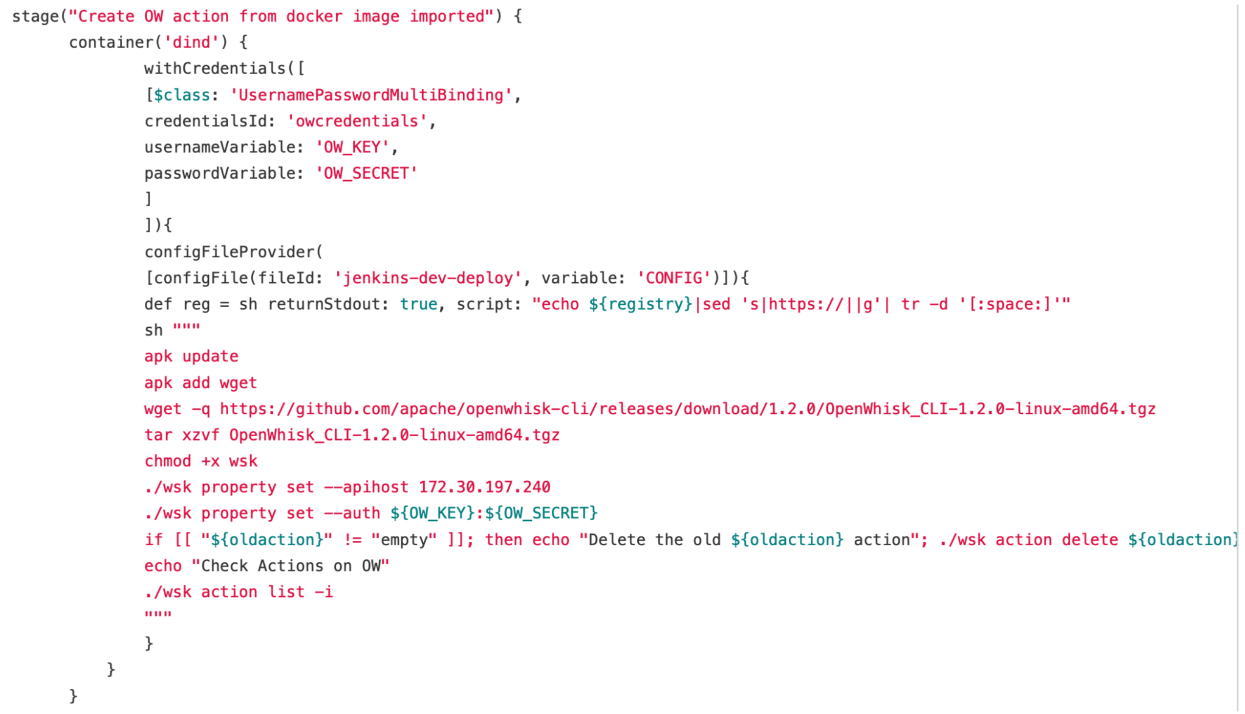
The third macro area is composed of two stages (Figure 23) used to evaluate whether the user wants to import the image from the external (private or public) or from PHYSICS registry.



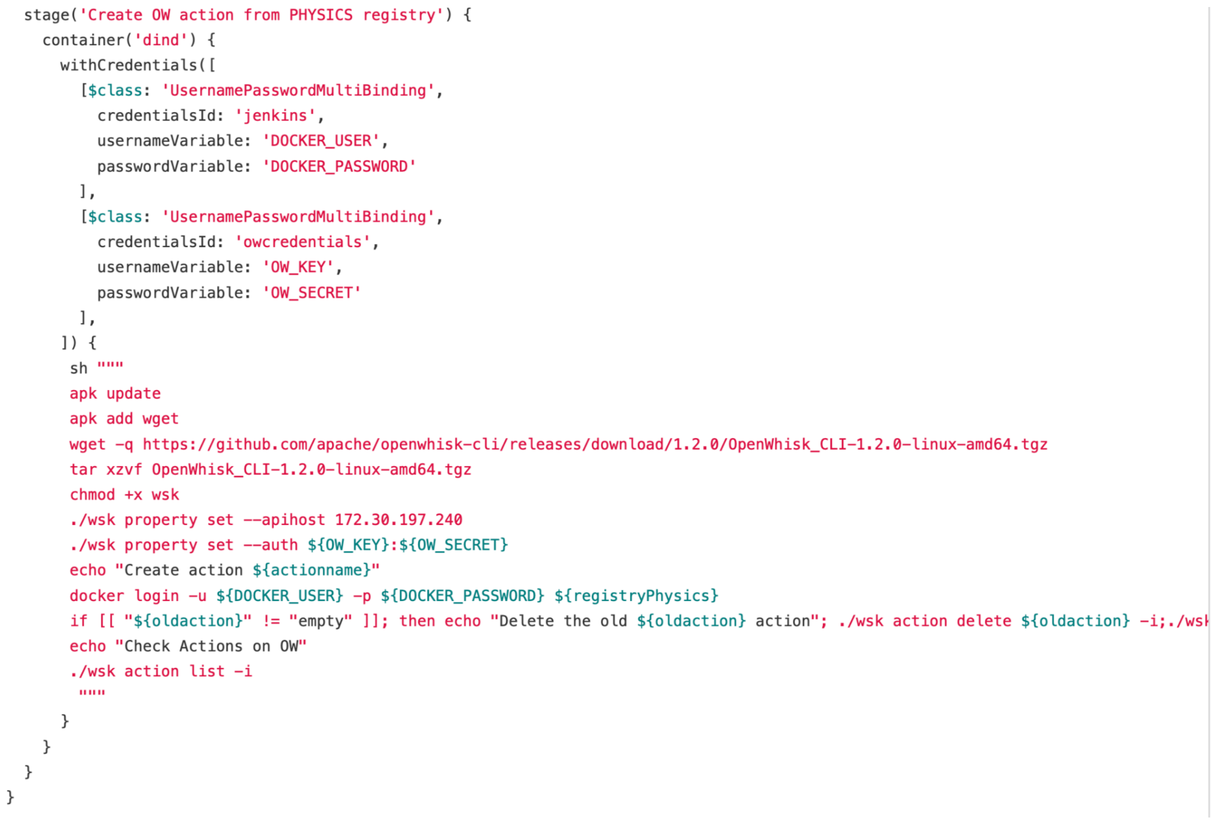
```
stage('Testing condition') {
  if (env.externalregistry.toBoolean()) {
    echo "You have selected to create OW action from external registry: ${registry}"
      if (env.public.toBoolean()) {
      echo "The registry: ${registry} is public"
      stage("Import docker image from public external registry") {
        container('dind') {
           withCredentials([
            [$class: 'UsernamePasswordMultiBinding',
            credentialsId: "jenkins",
            usernameVariable: 'JENKINS_USER',
            passwordVariable: 'JENKINS_PASSWORD'
            ]
          ]){
            sh """
           if [[ "${repo}" != "empty" ]]; then echo "The repo is: ${repo}"; docker pull ${repo}/${dockerimage}:${version};
           if [[ "${repo}" != "empty" ]]; then echo "The repo is: ${repo}"; docker tag ${repo}/${dockerimage}:${version} ${
           echo "Push image in our registry"
           docker login -u ${JENKINS_USER} -p ${JENKINS_PASSWORD} ${registryPhysics}
           docker push ${dockerPhysics}/external-uploaded/${dockerimage}-${user}:${version}
           """
          }
        }
      }
    } else{
       echo "The registry: ${registry} is private"
       stage("Import docker image from a private external registry") {
        container('dind') {
           withCredentials([
            [$class: 'UsernamePasswordMultiBinding',
            credentialsId: "${credid}",
            usernameVariable: 'DOCKER_USER',
            passwordVariable: 'DOCKER_PASSWORD'
            ],
            [$class: 'UsernamePasswordMultiBinding',
            credentialsId: "jenkins",
            usernameVariable: 'JENKINS_USER',
            passwordVariable: 'JENKINS_PASSWORD'
            ]
          ]){
            sh """
           docker login -u ${DOCKER_USER} -p ${DOCKER_PASSWORD} ${registry}
           docker pull ${repo}/${dockerimage}:${version}
           docker tag ${repo}/${dockerimage}:${version} ${dockerPhysics}/external-uploaded/${dockerimage}-${user}:${version
           docker logout
           echo "Push image in our registry"
           docker login -u ${JENKINS_USER} -p ${JENKINS_PASSWORD} ${registryPhysics}
           docker push ${dockerPhysics}/external-uploaded/${dockerimage}-${user}:${version}
           """
          }
        }
      }
    }
```

Figure 23: Check from which location to retrieve the docker image

The last macro area, also composed of two stages (Figure 24) defines whether the action on OpenWhisk will be created from the imported docker image or using the image already available in the PHYSICS registry.



```
stage("Create OW action from docker image imported") {
      container('dind') {
              withCredentials([
              [$class: 'UsernamePasswordMultiBinding',
              credentialsId: 'owcredentials',
              usernameVariable: 'OW_KEY',
              passwordVariable: 'OW_SECRET'
              ]
              ]){
              configFileProvider(
              [configFile(fileId: 'jenkins-dev-deploy', variable: 'CONFIG')]){
              def reg = sh returnStdout: true, script: "echo ${registry}|sed 's|https://||g'| tr -d '[:space:]'"
              sh """
              apk update
              apk add wget
              wget -q https://github.com/apache/openwhisk-cli/releases/download/1.2.0/OpenWhisk_CLI-1.2.0-linux-amd64.tgz
              tar xzvf OpenWhisk_CLI-1.2.0-linux-amd64.tgz
              chmod +x wsk
              ./wsk property set --apihost 172.30.197.240
              ./wsk property set --auth ${OW_KEY}:${OW_SECRET}
              if [[ "${oldaction}" != "empty" ]]; then echo "Delete the old ${oldaction} action"; ./wsk action delete ${oldaction}
              echo "Check Actions on OW"
              ./wsk action list -i
              """
              }
          }
      }
```



```
stage('Create OW action from PHYSICS registry') {
  container('dind') {
    withCredentials([
      [$class: 'UsernamePasswordMultiBinding',
        credentialsId: 'jenkins',
        usernameVariable: 'DOCKER_USER',
        passwordVariable: 'DOCKER_PASSWORD'
      ],
      [$class: 'UsernamePasswordMultiBinding',
        credentialsId: 'owcredentials',
        usernameVariable: 'OW_KEY',
        passwordVariable: 'OW_SECRET'
      ],
    ]) {
     sh """
     apk update
     apk add wget
     wget -q https://github.com/apache/openwhisk-cli/releases/download/1.2.0/OpenWhisk_CLI-1.2.0-linux-amd64.tgz
     tar xzvf OpenWhisk_CLI-1.2.0-linux-amd64.tgz
     chmod +x wsk
     ./wsk property set --apihost 172.30.197.240
     ./wsk property set --auth ${OW_KEY}:${OW_SECRET}
     echo "Create action ${actionname}"
     docker login -u ${DOCKER_USER} -p ${DOCKER_PASSWORD} ${registryPhysics}
     if [[ "${oldaction}" != "empty" ]]; then echo "Delete the old ${oldaction} action"; ./wsk action delete ${oldaction} -i;./wsk
     echo "Check Actions on OW"
     ./wsk action list -i
      """
    }
  }
 }
}
```

Figure 24: Create the OpenWhisk action with the docker image retrieved

- Authentication: The solution integrates a Single-Sign On solution based on the open source identity provider Keycloak and the LDAP directory to prevent unauthorised access to the develop environment (Figure 25). To speed up the onboarding process of new developers, a web based portal has been created for the self-provisioning of new accounts, where at the end of the provisioning the user can download the installation guide of the solution.

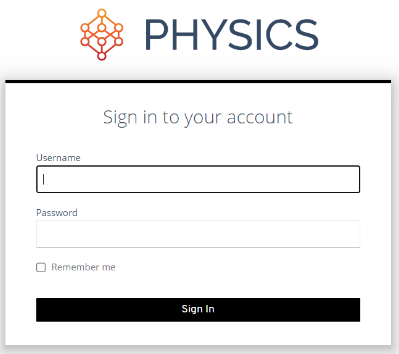


Figure 25: Authentication

2.5.2. Subcomponents Implementation

The Design Environment is distinguished in the following two parts:

- Control UI (Frontend)

The UI currently provides two main functionalities/tabs: one for importing the Node-RED environment and one for additional control features. In addition, the Control UI contains a navigation panel for Builds (building and deploying to test environment), Test (to trigger flows deployed in test environment), Graphs (creating application graphs and deploy them through WP4), Export Subflow (to export a specific subflow as a npm package), Visualization (a Node-RED dashboard embedded inside the Control UI, where the Developer can view the performance pipeline result for a specific function), Import Image (import application image from external registry in Physics environment) and Configuration (to configure deployment, i.e. choose Edge Locations). A number of indicative screenshots were listed above per process. The Control UI is an Angular Application written with Angular Material component library.

- Serverless Function Generator (backend)

Serverless Function Generator (SFG) is a REST service which works as a backend for the Control UI. It allows the frontend application to communicate (with help of supporting microservices described below) with all other external resources like Jenkins, Semantic Extractor, Node-RED Admin API or the Deployment Process (interface to WP4). SFG is built on the NestJS framework. There are other JavaScript REST frameworks like NextJS or ExpressJS, however, NestJS is built based on dependency injection and modularization mechanism of Angular so going with both Angular and NestJS makes the whole solution more consistent. The Serverless Function Generator API appears in Table 1.

Table 1: Serverless Function Generator API

HTTP	Path	Description
POST	/build	Extracts flow for given flowId (together with configuration and subflows), update repository with Node-RED data, upload the JSON file with flow to the object bucket and trigger Jenkins job with url to the uploaded flow.
POST	/graph	get the application graph, extract its flows and load the artifacts for each of them (trigger build process for the ones which are not built yet) and send it to Semantic Extractor to be stored
GET	/graph	Get all created graphs.
GET	/graph/draft	Returns all graphs, which are still waiting for some of its flows to be built.
GET	/function	Returns all available functions' names
POST	/function/invoke	Invoke function for given name and parameters
GET	/function/:activationID	Get result of function invokation
GET	/flow	Get all flows available in Node-RED
GET	/subflow	Get all subflows available in Node-RED
POST	/npm-packages	Requires a subflow id as input and return as output the related package, ready to be published on a package registry.

- Artifact Query Service (supporting microservice)

Artifact Query Service is a microservice, which allows us to query MongoDB to get artifacts for already built flows. Artifact Query Service is built on the NestJS framework (Table 2).

Table 2: Artifact Query Service API

HTTP Method	**Path**	**Description**
GET	/artifact	getting all artifacts for already built flows

- Graph Draft Service (supporting microservice)

Graph Draft Service is a microservice, which allows us to query MongoDB for graphs, which waits for its flows to be built to be created as an application in WP4. Graph Draft Service is built on the NestJS framework (Table 3).

Table 3: Graph Draft Service API

HTTP Method	**Path**	**Description**
GET	/draft	getting all graph drafts from database
POST	/draft	Add graph draft into database

- Function Service

Function Service is a microservice, which allows us to communicate with OpenWhisk to get a list of actions, invoke one of them and get its result. Function Service is built on the NestJS framework (Table 4).

Table 4: Function Service API

HTTP Method	Path	Description
GET	/function	Returns all available functions' names
POST	/function/invoke	Invoke function for given name and parameters
GET	/function/:activationID	Get result of function invocation

- Semantic Extractor Local (optional supporting microservice)

Semantic Extractor Local is a microservice, which can be used optionally to simulate functionalities provided by WP4 components to allow WP3 components working as a standalone platform. Its API is a merged API of the Semantic Extractor and Reasoning Framework. Semantic Extractor Local is built on the NestJS framework.

- Build Result Processor (supporting asynchronous microservice)

Build Result Processor is a microservice, which reacts on messages on RabbitMQ queue populated by Jenkins Job after successful build and process the data to be usable for other asynchronous processor described below. Build Result Processor is built on the NestJS framework.

- Graph Processor (supporting asynchronous microservice)

Graph Processor is a microservice, which reacts on messages on RabbitMQ queue populated by Build Result Processor and uses the build information to send graphs to semantic extractor if there are drafts of graphs, which needs only currently built flow to be created. Graph Processor is built on the NestJS framework.

- Artifact Processor

Artifact Processor is a microservice which reacts to messages in the RabbitMQ queue populated by the Build Result Processor, and saves built artifacts in the database to be used by SFG during graph creation. The Artifact Processor is built on the NestJS framework.

- Import Image Service

Import Image Service is a microservice, which allows us to query MongoDB for imported image data, to maintain trace of the state and the history of the user imported image. Import Image Service is built on the NestJS framework (Table 5).

Table 5: Import Image Service API

HTTP Method	**Path**	**Description**
POST	/import-image	Add import image into the database
GET	/import-image	Retrieve all import-image by user
PUT	/import-image/:id	Update import image state by id
GET	/import-image/docker-image	Get import image by docker image
GET	/import-image/cred-id	Get user label repository credentials

- Cluster Service

Cluster Service is a microservice, which allows us to query MongoDB to get the configured cluster where it can run the action to get the performance evaluations. ClusterService is built on the NestJS framework (Table 6).

Table 6: Cluster Service API

HTTP Method	Path	Description
GET	/cluster	Returns all available clusters

An overview of the Design Environment Architecture appears in Figure 26.

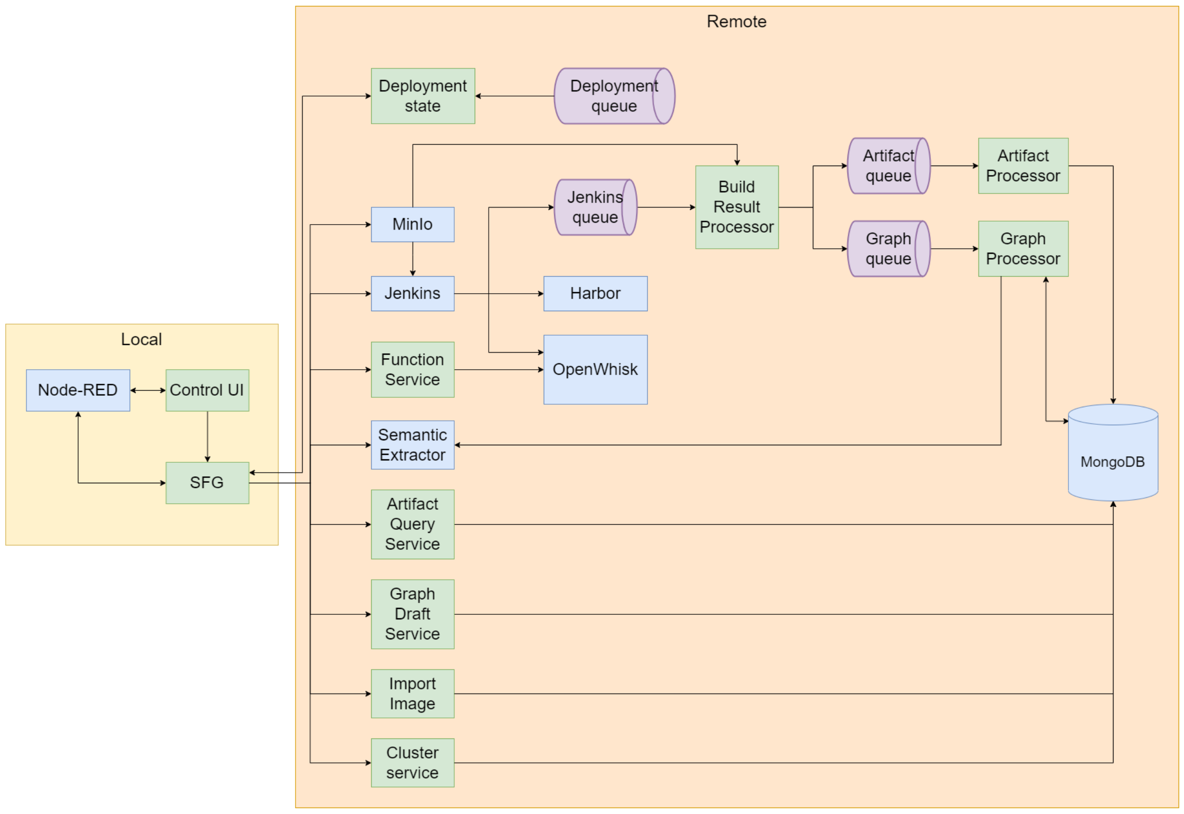


Figure 26: Overview of the Inner Architecture of the Design Environment

2.5.3. Local Design Environment Deployment

For testing purposes, the component can be served locally using NX [23] cli from the repository available on the PHYSICS DevOps environment or by the docker-compose command using a relevant docker-compose file containing Node-RED environment image, SFG image and Control UI image (Code 1). It communicates with Jenkins, available in the PHYSICS cloud environment. Besides that, it can work with the Semantic Extractor to allow Design Environment work as a standalone component, separate from the other PHYSICS components.

This setup can prove useful in dissemination activities as well as in cases of an interested entity only in the Design environment leading to a function registration and execution on a specific and already available local Openwhisk platform. Furthermore, it gives advantages in terms of latency between the developer and the Node-RED environment.

```
version: "3.9"
services:
 web:
  image: <url_to_control_ui_image>
  ports:
   - "4200:80"
 node-red:
  build: <path_to_Dockerfile_in_repository>
  volumes:
   - <path_to_data_directory_in_repository>:/data
  ports:
   - "1880:1880"
   - "8080:8080"
 sfg:
  image: <url_to_serverless_function_generator_image>
  volumes:
   - ./node-red-env:/repository
  ports:
   - "3001:3001"
 semantic-extractor:
  image: <url_to_semantic_extractor_image>
  ports:
   - "3000:3000"
```

Code 1: docker-compose.yml for launching local Design Environment

2.5.4. Cloud Design Environment Deployment

The next step for the Design Environment is to release it as a web based solution, with the aim to enlarge people's engagement and remove the difficulty of a local installation based on docker compose, although all the benefits of having a centralised application.
With that objective, a reengineering of the Design Environment architecture is performed, to enforce the authentication process on backend features and give to the user the same experience as in the local installation, to achieve that the architecture is redesigned as presented in Figure 27.

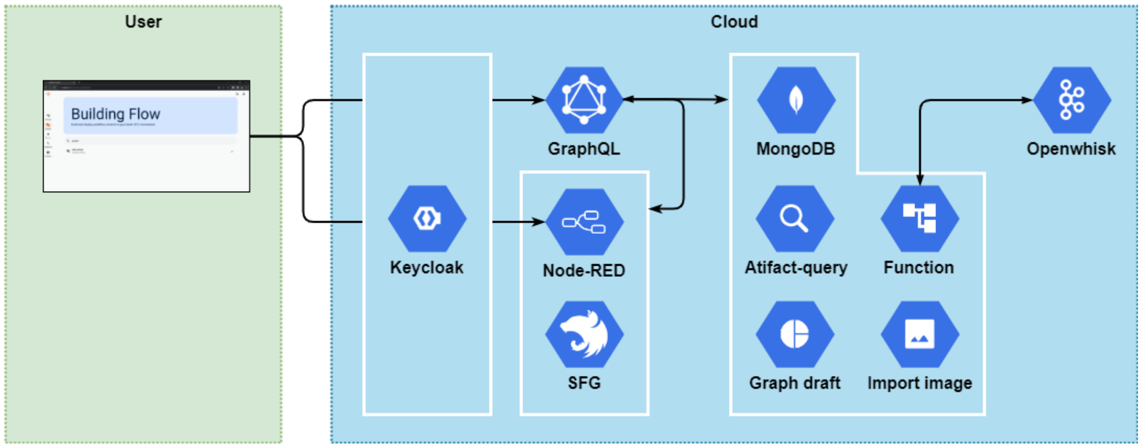


Figure 27: Cloud Design Environment Architecture

The implementation has integrated a new centralised backend based on an open-source, spec-compliant GraphQL server, that acts as middleware to the actual microservices and gets directly the data from MongoDB. On top of them, Keycloak, an Open Source Identity and Access Management solution, is used to authenticate all communications from frontend to backend.

With the development of the DE in the cloud, a new login page was created from which each user, after registering, has the possibility of running their DE and, once they have finished their work, to stop it. This provides two advantages:

1. Shutting down the environment when not in use eliminates any security risk.
2. Using it only when necessary allows better management of resources.

From an infrastructural point of view, the segregation of each environment was achieved by creating a specific POD for each user. Each POD within it is characterised by a dedicated Node-RED instance and a lightened version of actual Serverless Function Generator (SFG) is deployed per user. The creation of the user container is provisioned during the user creation phase through a new dedicated pipeline, which takes care of instantiating the container with the user's configurations while also defining a route, composed with the registered keycloak user name, for directing access to the Node-RED instance. To improve the performance and reduce the requested resource, the user container will start up at the login on the Design Environment and it will shutdown at the logout or automatically after one hour of inactivity of the container.

2.6. DevOps Subsystems Support in the DE

2.6.1. General DevOps process

As described in D2.5, the PHYSICS framework development and deployment has been split into two different strategies. The Development strategy defines the collaborative work of the technical WPs to build up the framework, with the goal of creating a MVP of the PHYSICS platform. The Deployment strategy defines a uniform approach to deploy all the PHYSICS components, particularly how to deploy them inside a cloud provider or an edge location based on a Kubernetes cluster. While this process is primarily intended for the PHYSICS provided tools, similar processes have been applied for generating the deployable artefacts in the context of WP3, i.e. the deployable versions of the code inserted by the developer during the implementation of an application in the context of PHYSICS.

The available tools for the CI/CD processes defined in the context of T6.1 (and described in D6.2) can also be exploited in the context of WP3 for the aforementioned artefact generation. These tools include:

- Gogs: A Git repository manager that lets each developer teams collaborate on PHYSICS 's source code.
- Jenkins: The de-facto standard open-source automation server for orchestrating CI/CD workflows. At the same time, it is also planned to evaluate the possible usage of Tekton tools, since it allows the implementation of pipelines in YAML format.
- Harbor: A popular Docker registry which is CNCF compliant.
- OpenLDAP: Used as the single user directory for all tools, centralising authentication and simplifying management of developer accounts.
- Helm: A package manager that streamlines installing and managing Kubernetes applications.

The DevOps strategy has been built around a set of pipelines, each one serving a different purpose in the overall process. Details on these are presented below.

2.6.2. Build Pipeline Design and Implementation (Local version)

In the context of the WP3 core processes, there is the need to get the developer-injected code/flow and create an artefact (software package, docker image etc) that can then be deployed on a target FaaS platform. The process of adapting the DevOps processes for building the artefacts is illustrated in Figure 28.

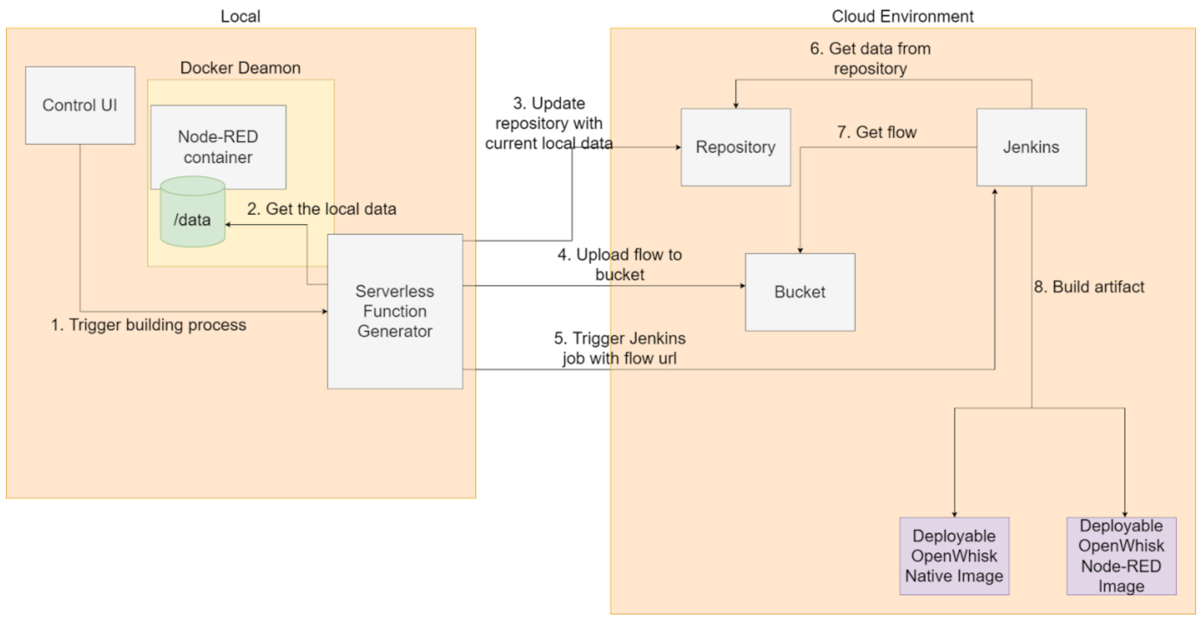

Local
Docker Deamon
Control UI
Node-RED container
/data
2. Get the local data
1. Trigger building process
Serverless Function Generator
3. Update repository with current local data
4. Upload flow to bucket
5. Trigger Jenkins job with flow url
Cloud Environment
6. Get data from repository
Repository
7. Get flow
Jenkins
Bucket
8. Build artifact
Deployable OpenWhisk Native Image
Deployable OpenWhisk Node-RED Image

Figure 28: DevOps process for the generation of deployable artefacts of the user

Once the developer chooses which Node-RED flow to build and triggers the process, the SFG prepares the flow for the Jenkins build, uploads it to the bucket and triggers the Jenkins Job with the URL to the flow. The Jenkins Job builds a base docker image importing the Node-RED data from the working Node-RED environment, in order to include any added nodes and dependencies added by the developer. Then it builds a second image for injecting only the specified flow. The created image is pushed to the docker repository registry and used by another Jenkins job for deploying the according OW action for that image to the test FaaS environment. Examples of the Jenkinsfile job that implements this process follow (Code 2, Code 3, Code 4).

```
...
 node(label) {
   def repo = '<node_red_data_repository>'
   def project = 'physics'
   def dockerPhysics = '<docker_images_subname>'
   def registryPhysics = '<docker_registry>'
   stage('Clone repository') {
     container('docker-cmds') {
        withCredentials([[$class: 'UsernamePasswordMultiBinding',
        credentialsId: '<credentials_to_use>',
        usernameVariable: 'JENKINSGO_USER',
        passwordVariable: 'JENKINSGO_PASSWORD']]) {
        sh """
          apk update
          apk add git
          git clone https://${JENKINSGO_USER}:${JENKINSGO_PASSWORD}@${repo}
          """
        }
    }
  }
  stage('Build Base Docker image') {
    container('docker-cmds') {
      def REP = sh returnStdout: true, script: "echo '${repo}'|awk -F / '{print \$3}'|sed s/.git//"
      withCredentials([[$class: 'UsernamePasswordMultiBinding',
        credentialsId: '<credentials_to_use>',
        usernameVariable: 'JENKINS_USER',
        passwordVariable: 'JENKINS_PASSWORD']]) {
        sh """
        cd ${REP}
        docker login -u ${JENKINS_USER} -p ${JENKINS_PASSWORD} ${registryPhysics}
        docker build -t ${dockerPhysics}/${project}/base:$BUILD_NUMBER .
        """
        }
    }
  }
  stage('Push Base Docker image') {
    container('docker-cmds') {
      withCredentials([[$class: 'UsernamePasswordMultiBinding',
        credentialsId: '<credentials_to_use>',
        usernameVariable: 'JENKINS_USER',
        passwordVariable: 'JENKINS_PASSWORD']]) {
        sh """
          docker image ls
          docker login -u ${JENKINS_USER} -p ${JENKINS_PASSWORD} ${registryPhysics}
          docker push ${dockerPhysics}/${project}/base:$BUILD_NUMBER
          """
        }
    }
  }
  stage('Build Custom Docker image') {
    container('docker-cmds') {
      def REP = sh returnStdout: true, script: "echo '${repo}'|awk -F / '{print \$3}'|sed s/.git//"
      withCredentials([[$class: 'UsernamePasswordMultiBinding',
        credentialsId: '<credentials_to_use>',
        usernameVariable: 'JENKINS_USER',
        passwordVariable: 'JENKINS_PASSWORD']]) {
        sh """
        cd ${REP}
        docker login -u ${JENKINS_USER} -p ${JENKINS_PASSWORD} ${registryPhysics}
        docker image ls
        docker build -t ${dockerPhysics}/${project}/custom:$BUILD_NUMBER --build-arg flowUrl="${flowUrl}" --build-arg
buildNumber="$BUILD_NUMBER" -f Dockerfilecustom .
          """
        }
    }
  }
  stage('Push Custom Docker image') {
    container('docker-cmds') {
      withCredentials([[$class: 'UsernamePasswordMultiBinding',
        credentialsId: '<credentials_to_use>',
        usernameVariable: 'JENKINS_USER',
        passwordVariable: 'JENKINS_PASSWORD']]) {
        sh """
          docker login -u ${JENKINS_USER} -p ${JENKINS_PASSWORD} ${registryPhysics}
          docker push ${dockerPhysics}/${project}/custom:$BUILD_NUMBER
          """
        }
```

```
   }
  }
 }
 }
```

Code 2: Example Jenkins file

```
FROM  <registryPhysics>/physics/debian-base:latest
USER root
RUN apt-get update && \
   apt-get install -y --no-install-recommends \
   default-jdk

ENV JAVA_HOME=/usr/lib/jvm/java-1.11-openjdk
ENV PATH="$JAVA_HOME/bin:${PATH}"
RUN java -version
RUN javac -version
RUN export JAVA_HOME

RUN apt update
RUN apt install -y python3-dev python3-pip python3-venv
RUN pip3 install --upgrade pip

COPY python-requirements.txt .
RUN pip3 install -r ./python-requirements.txt

RUN chown -R node-red /data
RUN chmod -R 775 /data

USER node-red
```

Code 3: Dockerfile example for Node-RED base image update

```
ARG buildNumber
FROM <registryPhysics>/physics/base:$buildNumber
ARG flowUrl
USER root
WORKDIR /usr/src/node-red
COPY ./data /data

WORKDIR /data
RUN npm install

WORKDIR /usr/src/node-red
RUN chown -R node-red /data
RUN chmod -R 775 /data
USER node-red
RUN curl -v $flowUrl > /data/flows.json
RUN ls -a

ENV PORT 8080
EXPOSE 8080
```

Code 4: Dockerfilecustom example for final Function Image Generation

2.6.5. Performance Pipeline Design

2.6.5.1. Performance Pipeline Motivation and Overview

The goal of the Performance Pipeline is to embed and automate the collection of performance data for a given function during the function development in order to serve a number of purposes. Initially, from an application/function creation (WP3 point of view), developers may need more direct information on the performance aspects of a given function version they created. In many cases, performance stress testing is performed at the end of the development. This implies that any performance related problems may be detected too late, or they may hide behind a complex application structure. Executing the performance test directly on each function version can help in the early detection as well as isolation of performance bottlenecks. Checking the performance of the same function under different versions is another use of a performance test embedded in the development process.

Furthermore, in typical FaaS cost models, the memory assigned to each function is one of the main aspects of cost consideration. A different rate applies for each memory size, with the other parts of the cost being the number of invocations and how much time each function execution takes for this memory setting rate [24]. Thus, analyzing the benefit of a certain memory setting relates to both functional (enabling the function to run successfully without out-of-memory errors) as well as non-functional aspects (benefit of using a higher level memory). This benefit should also be mapped to the difference in the achieved performance of this function. In this way, the function owner may be given an informed choice for the desired setting.

What is more, in the PHYSICS context, the existence of multiple available clusters as well as the optimization of function placement across these, implies the existence of relevant performance data that can help in that optimization. Having test executions of a given function across these clusters can aid in a more targeted and adapted decision making process in WP4.

Finally, from a resource management perspective (WP5 point of view, more information in D5.2 for the collocation strategies), resource contention due to concurrent container placement on the FaaS execution substrate is still a problem [25], generating delays of even 11x times higher function execution duration in cases of small-scale infrastructures such as private clouds. The identification of the resource profile of a given application component (i.e. if it is memory or CPU-intensive) has shown in the past that it can lead to significant minimization of this overhead [26], following its consideration from a provider point of view, and improved resource selection [27], from an application owner point of view. However the provider needs to have some information or annotation on the created functions in order to enable optimization of the co-allocation of these functions execution.

The Performance Pipeline is designed as an optional (from a developer point of view) extension to the conventional function packaging and deployment process in PHYSICS described in the previous sections. The mechanism integrates a load generation process towards the available clusters, retrieval of function performance evaluation (results of the function execution duration, wait time etc) as well as acquisition of the function execution resource usage profile. During the performance testing of the function, the pipeline collects low-level resource usage statistics (CPU, memory, network in/out, and file system use) for each function, creating a resource usage profile. This profile is stored and compared to other functions through a clustering and classification approach (with the help of the PEF in WP4), annotating each function based on its behaviour in the used metrics (e.g. low/medium/high memory). This way, the platform can benefit from a kind of "function crowdsourcing", categorising the available functions in a relative manner. During the app graph deployment, the included functions profiles are retrieved and embedded into the application graph by the WP3 services, thus forwarding them to be used by the according PHYSICS components down the stack (Global Continuum Placement in WP4 and Coallocation Strategies in WP5). From a FaaS provider's perspective, these annotations can be utilised to optimise function placement on available nodes. For instance, anti-affinity groups can be determined for similarly behaving functions based on their usage footprint, thereby reducing concurrent container overheads and competition for the same type of resources. More information on the background profiling and classification process can be found in D5.2.

2.6.5.2. Related Work

In [28], research was conducted on how serverless (mainly FaaS) affected DevOps practices. By hosting workshops and interviewing five company employees that were most involved with the development of the DevOps pipeline, the authors concluded that more than half of the DevOps practices are affected by serverless computing. It was also noted that one of the most critical parts of the DevOps development process is a reliable pipeline. Lastly, they ranked the importance of each DevOps practice. Automated performance and security tests in the target environment were graded with seven out of ten, however, the fully scripted deployments, where the performance pipeline is part of, were graded with nine out of ten and end-to-end testing was also graded with nine. The work does not include conclusions about how the pipeline should be implemented specifically.

The work in [29] presents an approach for developing a serverless pipeline by taking advantage of AWS serverless technologies in order to replace traditional CI/CD tools. This approach presented significant price benefits by using serverless technologies in the context of a pipeline. In our work, serverless functions are used for the main functionality of the performance pipeline (load generation, clustering, classification), aligning with the recommendations of the specific work. The authors of [30] applied processes similar to ours that include triggering a pipeline and gathering performance metrics through an analytics and monitoring solution. The testing was done in a range of RAM availability for the function runtime from 128 to 2048, with some exceptions for each major cloud provider (AWS, Azure, Google Cloud, IBM Cloud). The study focused on comparing the performance of different FaaS platforms for different RAM availabilities. Thus it covers the benchmarking part of our work (function owner view) yet not the profiling one (provider view).

Other works focus on the creation of a suite or tool for FaaS performance evaluation. In [31], a Serverless Application Analytics Framework was developed in order to help developers get performance metrics for their functions with minimal cost. This framework collects Linux Time Accounting metrics such as CPU idle, user, kernel, I/O wait time, wall-clock runtime, and memory usage directly from the function by requiring a few additional lines of code to be added to the function. Then a client-side application can be used to define and execute tests. This approach can be more detailed, however it intervenes with the function development process and needs the developer's collaboration. Compared to this, in our work, the profiling is performed in a non-intrusive manner.

A different approach was developed in [32] where instead of adding code inside the tested function the authors used a proxy cloud function (PCF) that was invoked before the target cloud function and was responsible for the data collection. The data collected by PCF includes execution time, time needed to create and route a request, latency time, response time, time to transmit data unidirectional, throughput and size of http request/response. So with this approach the authors managed to retrieve data from the function performance level, compared to the low level traces of the [31]. In other cases the data for the performance evaluation are pulled from the cloud service provider [33] [34].

From the investigated works, there are either profiling or benchmarking approaches, but there is no combined approach. The work presented in this section combines information from both function-related metrics, such as average wait time, initialization time, function execution duration, success rate, cold starts etc., as well as the resource usage of the function execution (i.e. low-level metrics including CPU, RAM, network and filesystem). Furthermore, it categorises the given function into relative low/medium/high categories per resource metric in order to be used during the function placement process [35], taking into consideration all other functions available in the platform.

2.6.5.3. Performance Pipeline Design and Process

The overall architecture of the performance pipeline appears in Figure 29 and has been integrated into the PHYSICS Cloud Design Environment process. The performance pipeline is an extension to this process, that allows either plugging in the extra performance analysis step or invoking it on demand based on user preferences. A series of steps are outlined here, corresponding to Figure 29, to facilitate reader understanding of the specified process:

- Function (Action) development: the process during which the developer creates the code and crafts a deployable artefact of a function (or action in the Openwhisk terminology)
- Load generation (Steps 1 and 2): the process of generating artificial load (Step 1: benchmarking) towards the deployed function in order to check its baseline execution time. The goal is to have exactly one function executing for multiple times and extract its average execution time as well as other parameters such as wait time in the system etc, which are stored during Step 2.
- Profiling (Step 3):process of acquiring the resource usage trace of an executed function during Step 1. After the execution of the load generator, the pipeline proceeds with querying the resource usage metrics from the PHYSICS platform. These metrics include averages for

function pod CPU, memory, file system and network data usage in the defined query interval. The data are retrieved based on the current timestamp after the finish of the load generation process minus the main load test duration set in the input parameters. Then the specific profile vector (trace) data is also saved in the performance DB and is also forwarded to the next stage (Function Classification). The acquired vector is stored during Step 4.

- Clustering (Steps p1-p3): the machine learning process of comparing the traces from multiple functions in order to create groupings of more similar values. The traces are gathered in Step p1, the clustering process is run in Step p2 and the results (centroids of resource behavior) are stored in Step p3. This process is executed offline and in a potentially periodic manner in order to get updated by new executions. It is not included in each pipeline run since the cluster centers are not expected to change significantly unless a significant increase in the dataset appears. In this way the pipeline delays are reduced.
- Categorization/Classification: the process through which a newly tested function's profile (extracted in Step 3) is categorized against a set of predefined categories defined in Step p2. The profiling vector of Step 3 is passed during Step 5 to the Categorization process. The latter retrieves in Step 6 the centroids created offline during Step p2 and measures the distance of this function profile in Step 7. The resulting categorization is stored in Step 8.

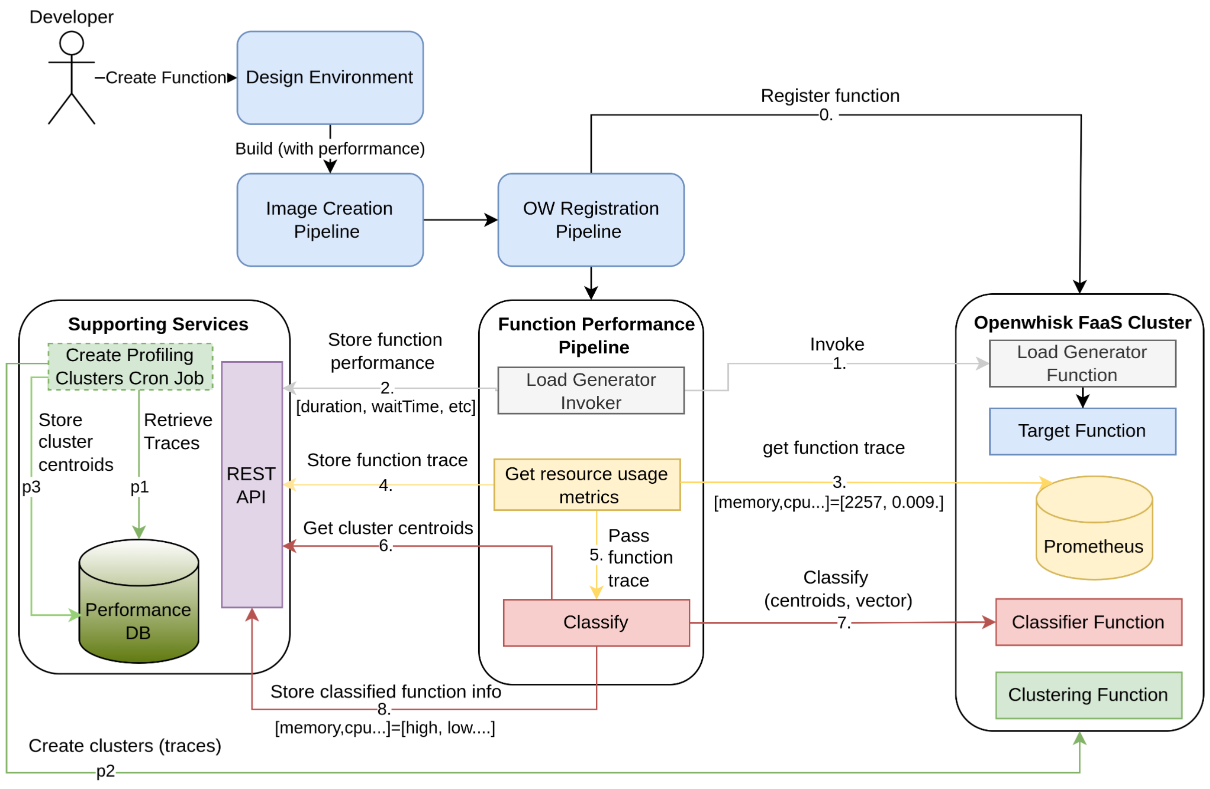


Figure 29: Performance Pipeline Process and Architecture

2.6.5.4. Finetuning the Load Generation Process for Profiling

A significant aspect of load generation needs to be considered, given that it primarily affects how the profiling is performed during the execution. One of the problems when applying a generic process for potentially different functions is how much request rate to use for the load generation. Ideally, in order to profile the usage of resources, one would need to have one fully utilised function container. If the execution time of a function is much lower than the inter-arrival rate of the function invocation, then this would result in high idle periods of the container between the invocations, as seen in the left part of Figure 30.

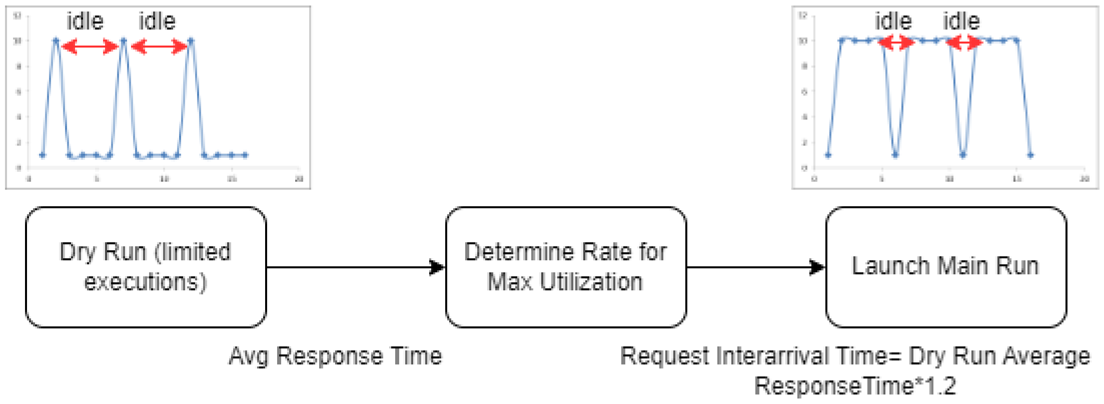

idle
idle
idle
idle
Dry Run (limited executions)
Determine Rate for Max Utilization
Launch Main Run
Avg Response Time
Request Interarrival Time= Dry Run Average ResponseTime*1.2

Figure 30:. Idle Times Correction through a two-stage Load Generation Process

This idle time effectively confuses the profiling process since average resource usage metrics are acquired from the load generation. Monitoring solutions such as Prometheus scrap monitor data every eg. 15-30 seconds, thus no specific resource monitoring can be done directly on the pure function execution time. Thus, these idle gaps should be minimised as much as possible since it would appear as the function is not using the resource, while it is actually the lack of invocations that causes the low metrics. On the other hand, we need to keep requests low enough so that they reuse the same function container (warm execution). This is again needed in order to get statistics from containers that are heavily used and do not include idle time. The same utilisation gap problem would occur even if we used a blocking client, i.e. one that waits for the response before sending the next request. In this case idle periods would appear since the serverless APIs do not allow blocking requests. They immediately return a response for receiving the request, but then the client needs to poll for the result. Thus the difference between the polling period and the function duration would again create idle gaps.

For the above reasons, we applied a two-stage load generation approach, as seen in Figure 30. Initially the first stage runs a load generator with a rather relaxed request rate (i.e. dry or trial run), e.g. 1 request per 30 seconds or 1 minute, in order to get with the same mechanism a baseline average response time for different functions that may have different duration sizes. Once this is acquired, this dry run response time is used as the inter-arrival interval between the requests of the main load generation phase. A 20% safety coefficient is applied in order to cater for any random higher delays. After finishing both stages, only the resource usage results from the second stage are collected. The performance outputs of the load generator are then stored as results.

2.6.5.5. Evaluation of the Two-Stage Load Generation process

To assess the effectiveness of the dry (trial) run adjustment, we configured the dry run settings with a request inter-arrival time of 30 seconds, which is sufficient to cover the execution time of many functions without generating numerous function containers. We then employed the process defined in the previous paragraph to regulate the inter-arrival rate of the main stage. Both stages have a similar duration of 200 seconds. Finally, we compared the outcomes of the dry run and the main run. The results from the FaaS cluster monitoring (Figure 31) demonstrate a clear difference in the obtained metrics.

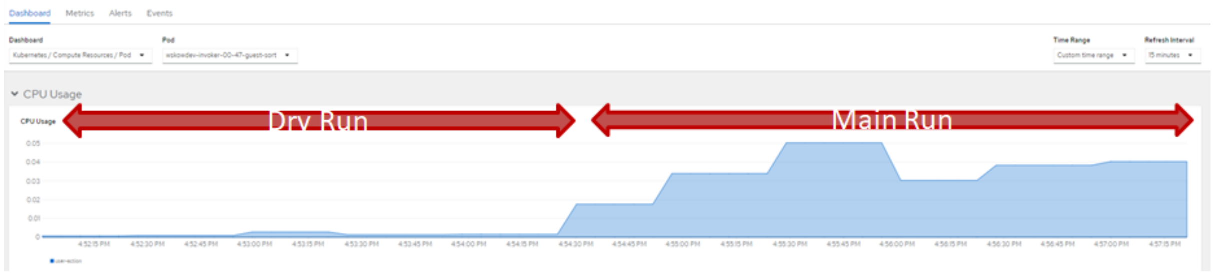


Figure 31: Difference in Observed Metrics between Dry and Main Run

Table 7 below presents the difference for each used metric in the two stages approach for the target function (a sorting function), along with a difference ratio, calculated from the division of the main run result by the dry run one. From this the importance of the adjustment is evident. CPU usage has 16 times more observed load while network usage portrays 45 times larger one when the two-stage approach is used. Only the memory appears to be similar, probably due to the fact that memory refers to the used memory allocation (which is allocated even with 1 function execution).

Table 7: Difference in Acquired Resource Usage Metrics between the Dry and the Main Run

Resource	**Without Dry Run**	**With Dry Run**	**Difference Ratio**
CPU	0.0009480	0.0158765	x16.74
Memory	22573056	27979776	x1.23
Network In	23.15	1042.875	x45.04
Network Out	11.575	520.875	x45.00

Filesystem Use	643072	5808128	x9.03

In addition, Table 8 portrays the generated inter-arrival request delays for 4 different experimental functions (sort, list, fibonacci and fileRW). The pipeline is able to detect the differences in the expected response time of each function and adapt the main run load generation achieving the adjustment goal. In this case the values are augmented since we did not remove the first execution inside the dry run that included a cold start. For further reducing the idle time, this value can also be removed since it appears only in the first execution of the function.

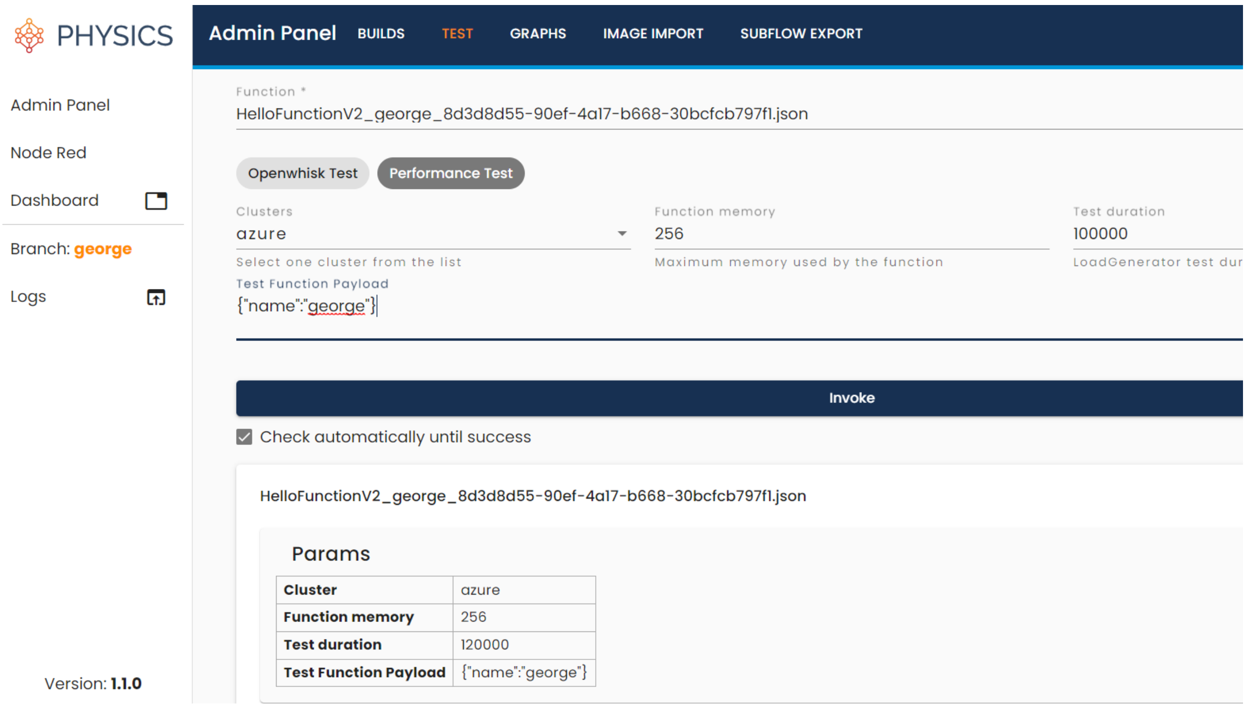
Table 8: Adaptable Determination of Diverse Request Rates for Different Function Types

Function	Determined Request Inter-arrival Rate (milliseconds)
sort	900.39
fibonacci	1314.20
fileRW	677.40
list	925.40

2.6.5.6. Performance Pipeline Execution

The Performance Pipeline is available as an extra tab in the Design Environment, as shown in Figure 32. It has a number of input parameters that relate primarily to function-specific details such as action name, test duration, function memory and test function payload. Once these are defined, no further involvement is needed from the developer side. The performance pipeline is executed inside an Apache Jenkins CI/CD environment (Figure 33), orchestrating the various stages needed (load generation, benchmarking results extraction, low level profiling metrics collection and invocation of the classification process).

The majority of these operations are implemented themselves as functions (load generator, cluster creator, classifier) available from the Performance Evaluation Framework of WP4, so that they can be easily deployable in a FaaS context. The pipeline is supported by a set of small services that aim to store state (benchmarking results, profiling vectors, categorizations and cluster centroids). This information is stored inside T.4.2 PEF and is used within the pipeline as well as maintained for future reference, presentation to the user and collaboration with other platform elements that are responsible for optimised operation including placement and scheduling optimizers. More information about the PEF APIs can be found in D4.2.



Function *
HelloFunctionV2_george_8d3d8d55-90ef-4a17-b668-30bcfcb797f1.json

Clusters: azure — Select one cluster from the list
Function memory: 256 — Maximum memory used by the function
Test duration: 100000 — LoadGenerator test dur
Test Function Payload: {"name":"george"}

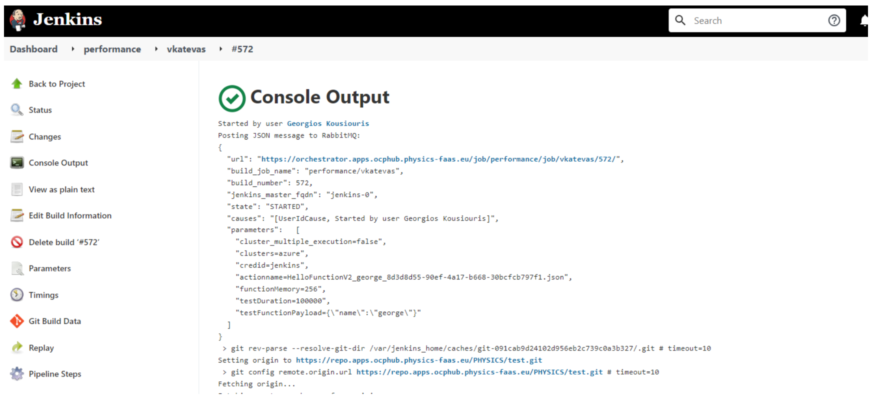
☑ Check automatically until success

HelloFunctionV2_george_8d3d8d55-90ef-4a17-b668-30bcfcb797f1.json

Params

Cluster	azure
Function memory	256
Test duration	120000
Test Function Payload	{"name":"george"}

Figure 32: Performance Test Tab in the PHYSICS Design Environment



Console Output

```
Started by user Georgios Kousiouris
Posting JSON message to RabbitMQ:
{
  "url": "https://orchestrator.apps.ocphub.physics-faas.eu/job/performance/job/vkatevas/572/",
  "build_job_name": "performance/vkatevas",
  "build_number": 572,
  "jenkins_master_fqdn": "jenkins-0",
  "state": "STARTED",
  "causes": "[UserIdCause, Started by user Georgios Kousiouris]",
  "parameters":   [
    "cluster_multiple_execution=false",
    "clusters=azure",
    "credid=jenkins",
    "actionname=HelloFunctionV2_george_8d3d8d55-90ef-4a17-b668-30bcfcb797f1.json",
    "functionMemory=256",
    "testDuration=100000",
    "testFunctionPayload={\"name\":\"george\"}"
  ]
}
 > git rev-parse --resolve-git-dir /var/jenkins_home/caches/git-091cab9d24102d956eb2c739c0a3b327/.git # timeout=10
Setting origin to https://repo.apps.ocphub.physics-faas.eu/PHYSICS/test.git
 > git config remote.origin.url https://repo.apps.ocphub.physics-faas.eu/PHYSICS/test.git # timeout=10
Fetching origin...
```

Figure 33: Example Execution of the Performance Pipeline in Jenkins

If a developer wants to compare function execution duration for different functions or different function versions to identify which one performs the best, load generator data like those shown in Figure 34 can be used. From the examples presented the developer may determine if a specific function may need performance improvements.

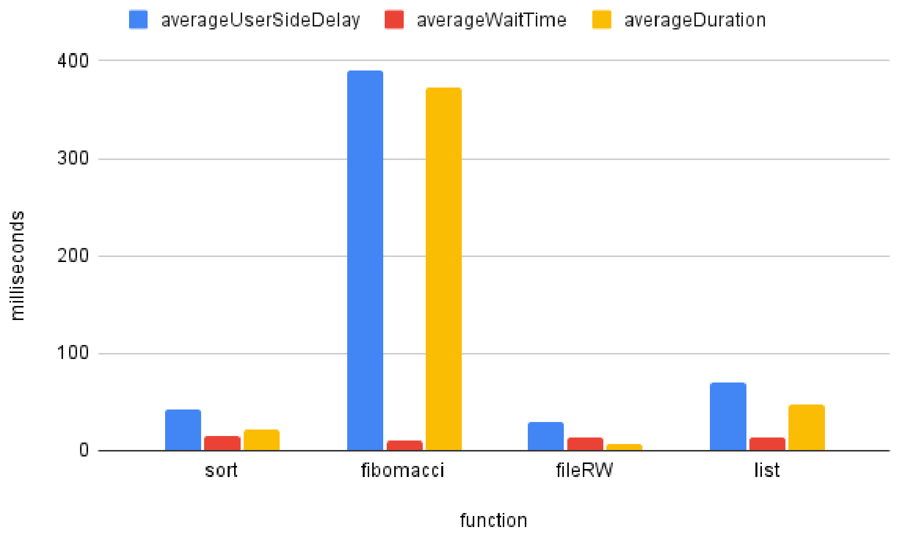


Figure 34: Example output times from the load generation benchmark data

Furthermore, they can apply the process for different memory settings of the same function and acquire a relevant comparative analysis like the one shown in Figure 34 for a sorting function case. From this it can be seen that for the given function, increasing the memory size from 128 to 256 can lead to an improved execution, however further increase does not represent a significant difference. Thus it can be used to optimise the sizing of a function as well as the associated costs, since in a typical FaaS model cost is largely dependent on the execution time and the memory of a function (Figure 35).

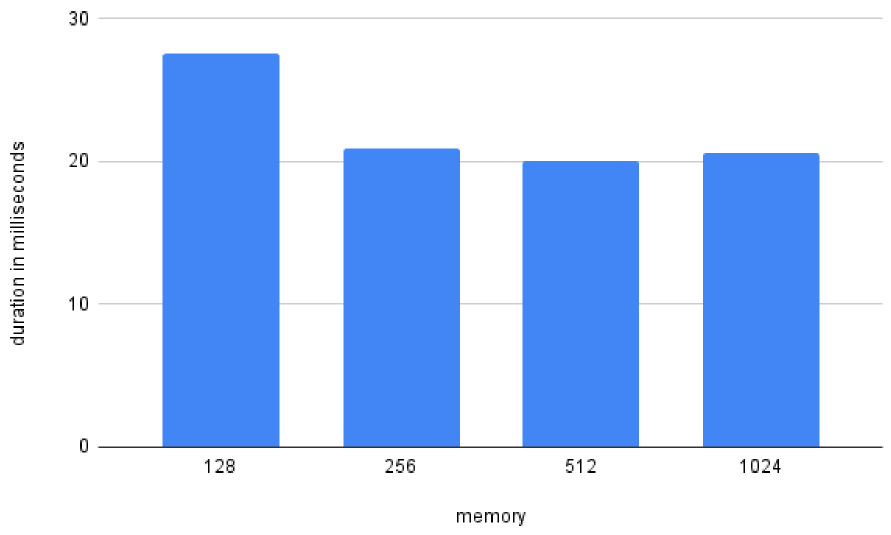


Figure 35: Sort function execution duration for different memory sizes

2.6.5.7. End to End Performance Pipeline Delays

The duration of a performance pipeline is largely set by the developer, since they select the test duration of the main load generation phase (as seen in the previous paragraph). Thus, they can select whether having a more thorough analysis or a faster one is more important. The test runs used in our work had a duration of around 8 minutes each. A chart with indicative durations of these runs can be seen in Figure 36. The dry run phase was statically set to 200 seconds as well as the main run one.
The remaining delays in the aforementioned figure refer to the orchestration needed in the pipeline. Because all the main steps of the pipeline are implemented through external functions, for better manageability and plug & play architecture, the pipeline needs to poll in the intermediate steps for getting the results of the load generator and the classifier function. This is due to the fact that serverless APIs typically do not allow blocking calls, so the caller needs to poll afterwards for getting the execution results. We applied a polling period of 5 seconds for this process. The orchestration delays are in total around 2 minutes in each run, excluding the delays of the dry and main run.

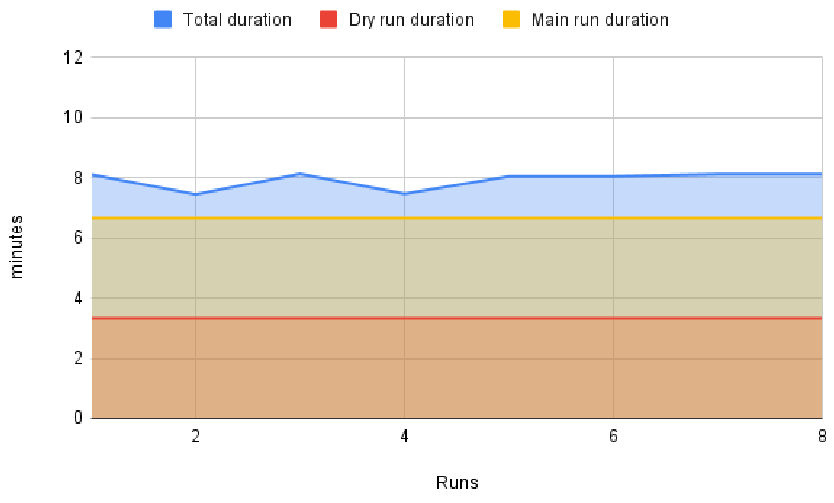
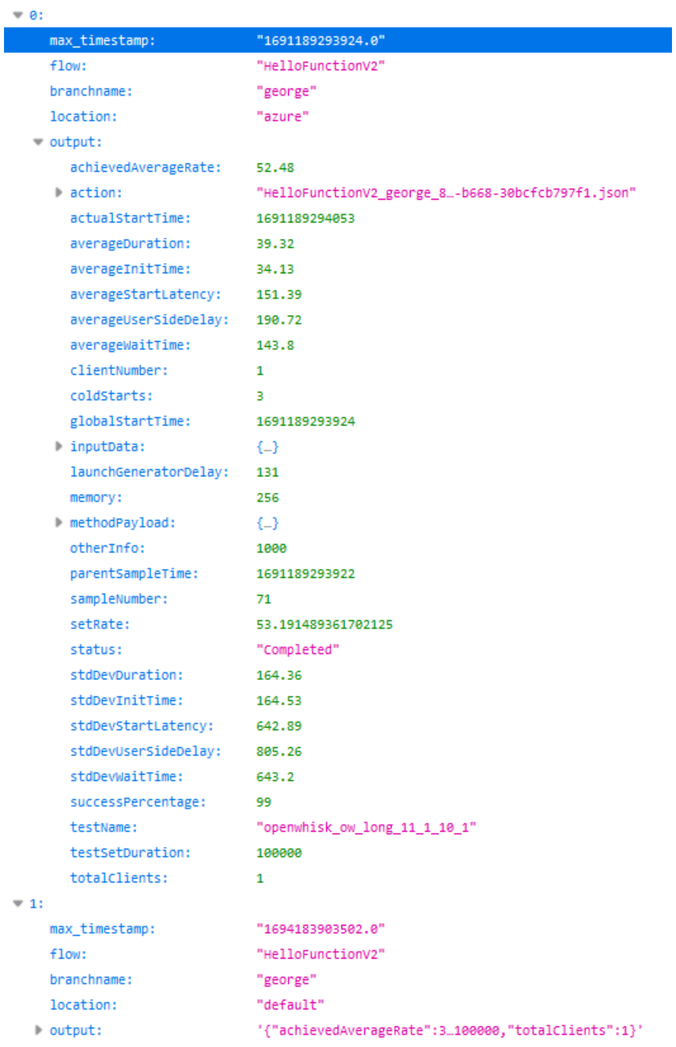


Figure 36: Performance Pipeline Indicative Delays

2.6.5.8. Performance Pipeline Outputs and Multi-cluster Support

The performance pipeline has also been designed to enable the support for benchmarking against multiple clusters. Given that PHYSICS may use more than one clusters, the platform needs to be aware of the performance of a function in each one, which also serves as an input to the Global Continuum Placement process in WP4. Thus the cluster to use is also an input parameter in the process.

Load generation is always performed from the hub cluster central location (AWS in our case). This is a conscious decision since we also want to include any network latencies in the acquired benchmark results, assuming that the remaining application components, invoking the function, will be located centrally. An example of the outputs is included in Figure 37, retrieved from the WP4 PEF API. More details on this are provided in D4.2.



```
▼ 0:
    max_timestamp:          "1691189293924.0"
    flow:                   "HelloFunctionV2"
    branchname:             "george"
    location:               "azure"
  ▼ output:
      achievedAverageRate:  52.48
    ▶ action:               "HelloFunctionV2_george_8…-b668-30bcfcb797f1.json"
      actualStartTime:      1691189294053
      averageDuration:      39.32
      averageInitTime:      34.13
      averageStartLatency:  151.39
      averageUserSideDelay: 190.72
      averageWaitTime:      143.8
      clientNumber:         1
      coldStarts:           3
      globalStartTime:      1691189293924
    ▶ inputData:            {…}
      launchGeneratorDelay: 131
      memory:               256
    ▶ methodPayload:        {…}
      otherInfo:            1000
      parentSampleTime:     1691189293922
      sampleNumber:         71
      setRate:              53.191489361702125
      status:               "Completed"
      stdDevDuration:       164.36
      stdDevInitTime:       164.53
      stdDevStartLatency:   642.89
      stdDevUserSideDelay:  805.26
      stdDevWaitTime:       643.2
      successPercentage:    99
      testName:             "openwhisk_ow_long_11_1_10_1"
      testSetDuration:      100000
      totalClients:         1
▼ 1:
    max_timestamp:          "1694183903502.0"
    flow:                   "HelloFunctionV2"
    branchname:             "george"
    location:               "default"
  ▶ output:                 '{"achievedAverageRate":3…100000,"totalClients":1}'
```

Figure 37: Example Function Benchmark Output of a Load Generation across Different Clusters

3. Semantic Models for Application Characteristics Description

3.1. Introduction- Scope of the Application Characteristics Description

As mentioned in Section 2, the PHYSICS Design Environment is an entry point for the development of an application that will later on be deployed as a PHYSICS app in the context of WP4. In order to detect what needs to be described from an application graph specification, which will be forwarded to the respective deployment process, one needs initially to define what is an application in the context of PHYSICS and from which parts it comprises. An application in PHYSICS is a collection of flows (that consist of functions) and services. More than one element of each category may be included in an application, grouped under the same app ID. Grouping by the same app ID implies that we can set options and annotations at any desired level (e.g. function, flow or application), while managing the overall collection as a unified application (e.g. during deployment). Some further details on the main building elements are as follows.

Functions

The functions are the primary building blocks. Functions can be generic, including any piece of code included by the developer, built-in (or imported from external repositories) Node-RED nodes, as well as external docker images executed upon request. In order for the latter to happen, the images need to be encapsulated around the relevant FaaS platform specification for execution. This includes a set of steps to prepare the image, which is platform-specific. For Openwhisk, the relevant specification needs to include two methods [36]. In essence, any container image that has included a web server and exposed two methods (/init for initialization and /run for execution) can be used as the target of invocation, assuming that a relevant link between these two methods and the internal needed actions has been applied (e.g., triggering of the relevant execution script in the /run case as well as processing of the input arguments in the request in order to pass them to the underlying execution).

Flows

A flow is a group of functions that are linked together in a workflow in order to achieve a specific goal or part of the application logic. These functions receive the initial message, process it based on their inner logic and then propagate it to the next function in the chain. In order to increase reusability and manageability of a flow, specific and reusable subgroups of functions can be grouped around subflows, appearing as one node in the flow.

Inclusion of services

Given that the serverless paradigm is not necessarily a "one size fits all" model, we consider that there will be parts of the application that need to be executed and included as typical services. This not only enables the richer representation of the application, but also facilitates easier porting, limited to the application parts that are expected to benefit the most from the FaaS approach, however still being able to manage the overall application through a single platform. Thus, one of the goals of PHYSICS should also be the ability to include existing services in the context of an application. One distinction for the services inclusion is whether these services should be directly controlled by the PHYSICS platform (i.e. deployed and operated) or they are external services used without control.

Besides this general structure of the application, dedicated semantics should also be defined describing a set of characteristics for each part of the application as annotations, in order to guide the lower layers of PHYSICS management and decision making. These may be deployment or execution options, constraints for operational management, elasticity considerations etc. In essence such an annotation list should exist and be configured through the environment, providing ways for the developer to set them and the framework to retrieve them. Following, details on each step are presented for the use of semantics at the WP3 level, i.e. the semantic descriptions at the application side. Further information on semantics is available in D5.1 for the semantics at the resource side, while the bridging framework is the Reasoning Framework presented in D4.1.

3.2. Relation to project requirements

From D2.3, the list of requirements that the semantic approach should address is the following:

- Req-3.2-WorkflowCoverage
  - Ability to understand and model the structure of a functional workflow
- Req-3.2-RequirementsCoverage
  - Include sufficient attributes for the requirements a workflow component may have regarding hardware, software or location.
- Req-3.2-ConstraintsCoverage
  - Include sufficient attributes for the requirements a workflow component may have regarding QoS, affinity placement (in the cluster) etc.
- Req-3.2-LinkWithVocabularies
  - Links with other ontologies in order to follow the linked data paradigm.
- Req-3.2-ReasoningCapability
  - The created triples must be effectively usable within reasoners.
- Req-3.1-CustomDockerImages
  - Ability to define an arbitrary docker image as the actionable artefact of a function

One general requirement is also the fact that for the deployment of the application, its structure and elements should be appropriately described so that they are understood and managed by the WP4 process.

3.3. Semantics Use Cases

3.3.1. Include Annotations UC

As identified in D2.5, the application developer in PHYSICS may annotate parts of the application created or imported through the Node-RED environment. In order to further specialize the specific use case from a semantic perspective, a more detailed view appears in Figure 38. The annotations are useful for a variety of purposes such as:

- functional needs (e.g. dictate what image is used for which architecture, a specific location constraint for function deployment & execution etc)
- non-functional needs (e.g. QoS constraints, importance of the specific function etc)

These annotations are taken under consideration at all stacks of the PHYSICS platform, according to their scope. For example, placement constraints should be taken into account when deciding the placement of the overall application, importance during scheduling in the cluster etc. In order to annotate flows (Figure 38), the main abilities of Node-RED can be used, including the use of specifically defined nodes, created subflows as well as annotation characteristics inside the environment (such as function groups) that can be used for that purpose. Other indirect annotations can also be acquired, e.g. regarding the application structure, from the JSON specification of a Node-RED flow (e.g. which functions are wired together).

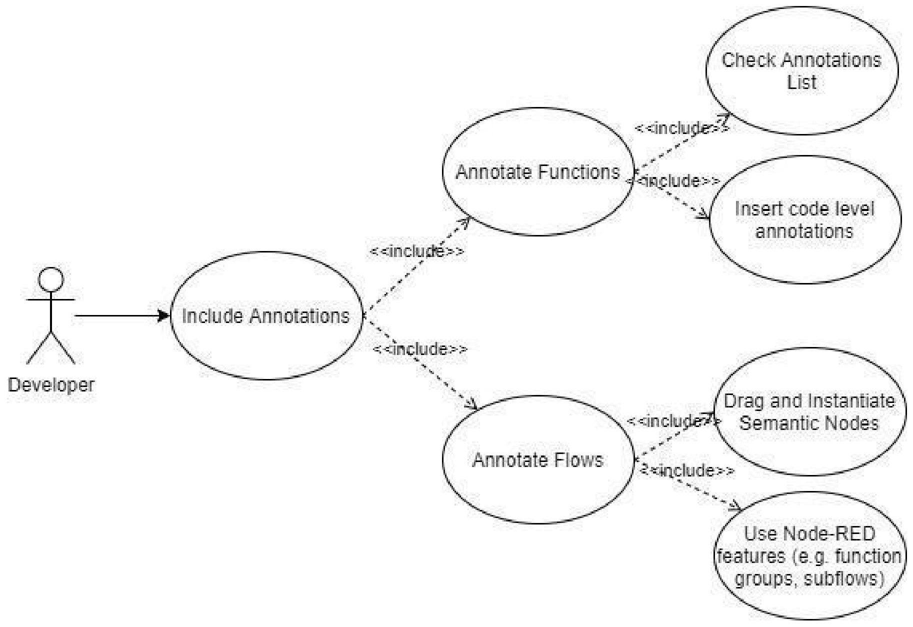


Figure 38: Include Annotations Semantic UC specialization

3.3.2. Create Semantic Graph

During the deployment process, the developer would need to select which flows are part of the application graph and trigger an app graph creation process, as detailed in Section 2.5.1 Step 5.

During this process the semantic block of the environment should retrieve the respective flows, process the JSON specification of Node-RED that includes the various functions and their connections and map them to the ontological concepts defined in the PHYSICS ontology (described in Section 3.5). The relevant use case appears in Figure39. Following, the description of the application structure as well as the elements and annotations it comprises should be forwarded to the Reasoning Framework (Inference Engine) of WP4 for the follow-up actions. In order to enable this use case, a process and component to generate the graph and semantic triples from the available Node-RED flows is needed (Semantic Extractor component defined in Section 3.7).

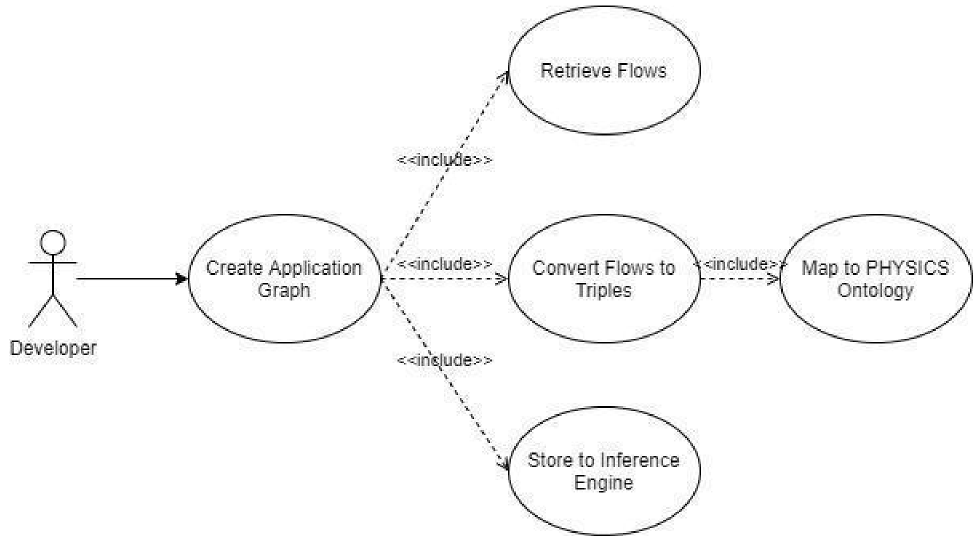


Figure 39: Create Application Graph UC from a semantic perspective

3.4. Annotation Mechanisms Incorporation

In the PHYSICS environment, two ways have been designed to import annotations, in order to adapt to the level these annotations also need to apply (function or flow), a code level process and a semantic node one. Information coming from the DevOps process, regarding the generated deployable artefact per element of the graph is also needed, since this will be finally used by WP4. After this process, the overall application graph is represented by a set of JSON-LD triples under the same app ID in the Reasoning Framework of T4.1. From this location they can be retrieved by the remaining deployment process in WP4. The overview of the annotation mechanisms interactions is depicted in Figure 40. Following, details on the two designed annotation mechanisms are portrayed.

3.4.1. Function level annotations mechanism

At the function level, in-code annotations can be wrapped around a specific syntax, and are put within single-line comments within the simple function node in the Node-RED visual development tool. The inspiration for such an annotation mechanism originates from the Dependency-Aware FaaSifier implementation [37], the special characters used in annotations in common programming languages, as well as the fact that statements regarding the semantic web and graph databases are commonly expressed as triples of object-property-value. The core idea is to have a way to express (a) the property and value wherever the object is the function itself, and (b) the complete triple in cases where there need to be more definitions that are related to the function. The syntax is explained below, along with what each statement means in terms of semantics:

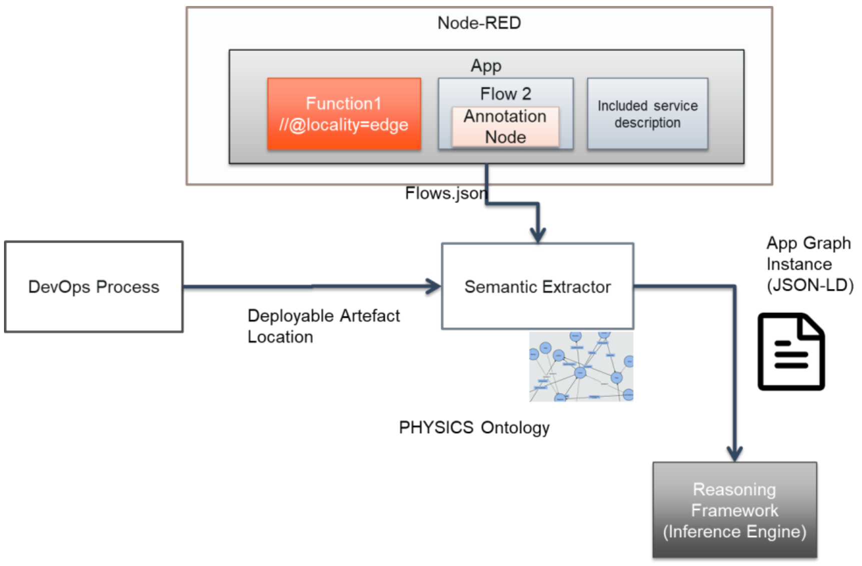
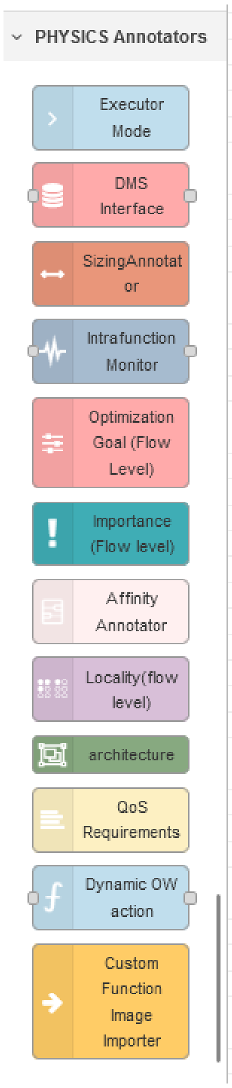


Figure 40: Overview of Semantic Annotations Mechanism Interactions

- Function annotation: //@<property>=<value> This kind of annotation is composed of (a) a single-line comment symbol, (b) the '@' character, (c) the annotation, which is a property from the ontology that has the Function class as domain, (d) the '=' character and (e) the value. Practically such an annotation maps to a triple where the object is the function that contains it. The most important kinds of function annotations are laid out in section 3.4.3, with an explanation for each one.
- Triple associated to the function: //$<object>@<property>=<value> This case is composed of (a) a single-line comment symbol, (b) the '$' character, (c) the object or local definition, (d) the '@' character, (e) a predicate/property from the ontology which has the class of the object as domain, (f) the '=' character and (g) the value. This proves useful in cases where a simple function annotation includes an object property, rather than a data property, and the value is an object that either needs to be locally defined, or needs to have some of its other properties defined within the context of this particular function. This particular scheme is for more advanced usage, and allows for the creation of complex directives for the given function.

In both of the above cases, the key or the property originates from the ontology, and the value is of the type that the range of the property is, and all included objects, properties and values do not have any prefixes. The corresponding triples are created by parsing the code content of the function node and checking for these two syntax schemes. Each object-property-value triple is created only if it is valid, given the ontology and the overall environment of the development tool for the given user/developer, and the appropriate context and prefixes are added to the data. This process is mainly enacted by the semantic extractor, as explained further in Section 3.7, and it results in triples that use a context that has the PHYSICS ontology as the main vocabulary, serialized in the JSON-LD format. These triples are included in the semantic representation of the overall flow, and are sent to the Reasoning Framework for storage.

3.4.2. Flow level annotations mechanism- Semantic nodes



At the flow level, a special set of nodes (semantic annotators) has been created as subflows and included in the Node-RED palette. Node-RED offers the ability to create subflows, in which one can define the required fields (e.g. in a UI format). These fields are included as subflow properties and environment variables. Various nodes have been implemented up to this point in order to address one or more categories of annotations needed at the flow level. A number of indicative ones are presented below. Each of the semantic nodes is also accompanied by a relevant README file accessible in the Node-RED environment. The PHYSICS Annotators palette, available in Node-RED appears in Figure 41.

Executor Mode Node

The Executor Mode is a semantic node used to indicate whether a specific flow will be executed as a service or as a function.

Intrafunction Monitor

This is a helper node that can be used together with a logging system, in order to forward monitoring information from an arbitrary location in a Node-RED flow. Thus with the use of this node the developer can redirect checkpoints or other status/performance information to an external logging service or the DMS service from WP4.

Optimization Goal

This node is to indicate the developer goals in the form of weights (0-100) for the 3 main optimization goals: performance, energy and cost. Thus the developer should include in each flow the percentage interest in each of them, e.g. 30(%) on performance, 30 (%) on energy and 40 (%) on cost. The total should be 100.

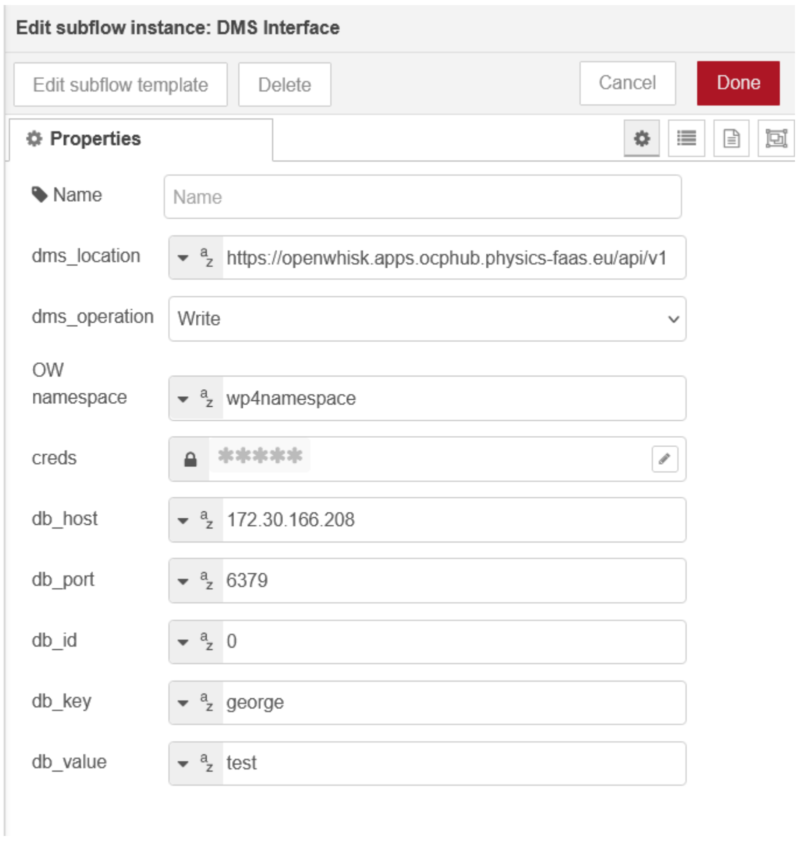
Importance Node

This is a node to indicate the importance of a flow (low/medium/high). This could be useful in the context of scheduling or autoscaling mechanisms prioritization for functions.

Figure 41: PHYSICS Annotators palette available in Node-RED

DMS Interface Node

This is an interface node in order to interact with the Data Management Service functions that are available for interacting (writing/reading) to KeyDB (T4.4). The node can be used in any developer flow in order to read or write data to the DMS. The data to be written (key/value pair) needs to be included in the msg.key (key for storing) and msg.value (value of storing) or set via the UI. The full UI of the node appears in Figure 42 and includes information on the location and configuration for contacting the service. This information can also be passed through according message fields in the Node-RED flow that includes the node. The inclusion of the DMS interface node as a semantic annotation node was performed so that the resulting usage of the DMS is depicted in the annotations forwarded to WP4. In this manner, the platform would understand that a given function is using the DMS service and could also take this into consideration when deciding where to place that function from a data locality point of view.



Name	Name
dms_location	https://openwhisk.apps.ocphub.physics-faas.eu/api/v1
dms_operation	Write
OW namespace	wp4namespace
creds	*****
db_host	172.30.166.208
db_port	6379
db_id	0
db_key	george
db_value	test

Figure 42: Executor Mode Semantic Annotator Node

Affinity Node

This is a semantic node to indicate that a flow has affinity considerations with flows in the same or other app (Figure 43). This information may be useful for the placement and optimization processes of PHYSICS. The target action is referenced through the deployment artefact id (e.g. imageID), so it needs to have been built beforehand. The relevant information can be found in the Build Tab of the Design Environment. The app ID can be used to indicate an action belonging to a different app. Therefore this information needs to be available in the Design Environment (it is under the Graphs tab). If the app ID is left empty, it is assumed that the node refers to this app, since the app ID is given during application creation and it is not available at design time.

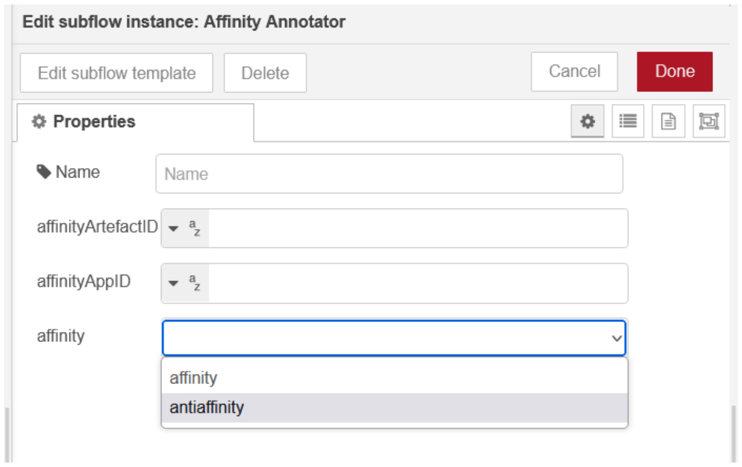


Figure 43: Affinity Semantic Annotator Node

Locality Annotator Node

This node is used to indicate the locality, if needed, for a given function. The node includes logic to dynamically retrieve the available locations, which are then offered as a dropdown for the developer to choose from.

Architecture Node

This is a node in order to include details of the needed h/w architecture for a given function (e.g. GPU capabilities, specific CPU architecture capabilities etc)

QoS Requirements Node

This node is available in order for the developer to specify the desired QoS metrics for their function execution. This information may be used by other components like the autoscalers or the routers available. The metrics available are the same offered by the Monitoring Pattern described in Section 4, i.e. average duration, initialization or wait time for a given function and the window of time this should be calculated upon. An example from the node UI appears below (Figure 44).

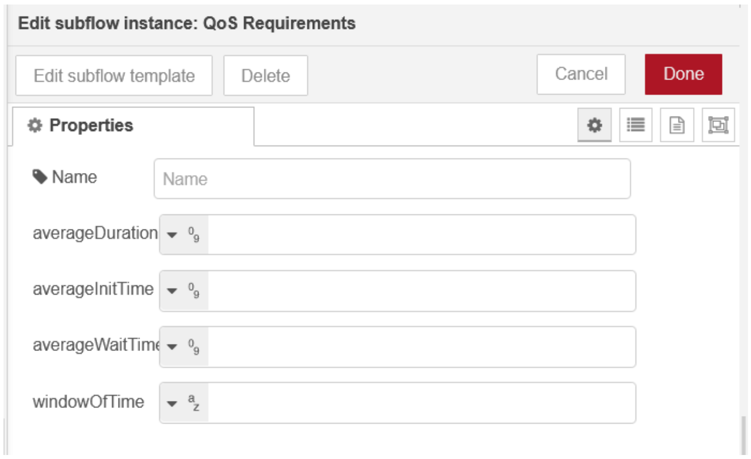


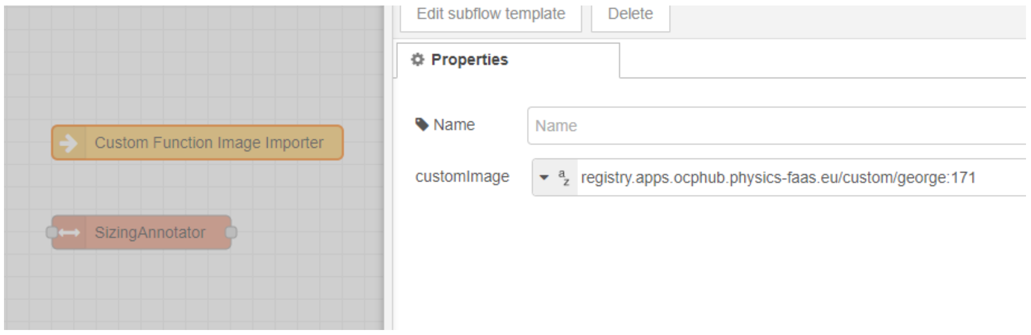
Figure 44: QoS Requirements Semantic Node

Dynamic OW Action Node

This is the node to use dynamically placed actions, whose placement and location is decided after design time. Due to the importance of this process, more details are included in Section 4 under the Dynamic Orchestrator Pattern.

Custom Function Image Importer Node

This node is to be used in the case a custom image has been used as the basis for a function, as mentioned in Section 2. After importing the image from the aforementioned functionality, the main intention is to use it in the context of a PHYSICS application, thus to include it in a collection of flows and functions to be deployed at the typical PHYSICS production environment. However, given that this function has not followed the typical stages of a PHYSICS function, a relevant declaration process needs to be followed, including creating a semantic annotation for this custom image used. In order to support this, the PHYSICS DE provides a relevant semantic node, the "Custom Function Image Importer". The user needs to create a new flow in the DE, in which they will drag and drop that node and populate it with the name of the custom image in the relevant field (Figure 45). In this flow they can also include other needed annotations from the PHYSICS available ones (e.g. sizing, locality etc.).



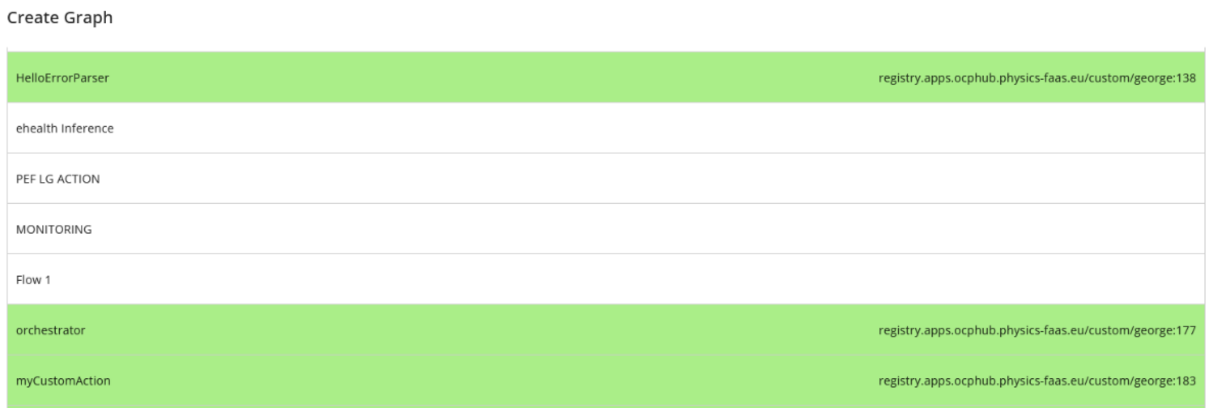
Name: Name

customImage: registry.apps.ocphub.physics-faas.eu/custom/george:171

Figure 45: Custom Function Image Importer Semantic Node

Finally, although there is no relevant function logic inside this flow, they need to build it through the DE. The reason for this is that only built flows are allowed to be included in an app graph in the next stage. So the system needs to have this build documented. Once the flow is built it can then be added to a PHYSICS App like any other "green" flow (Figure 46).

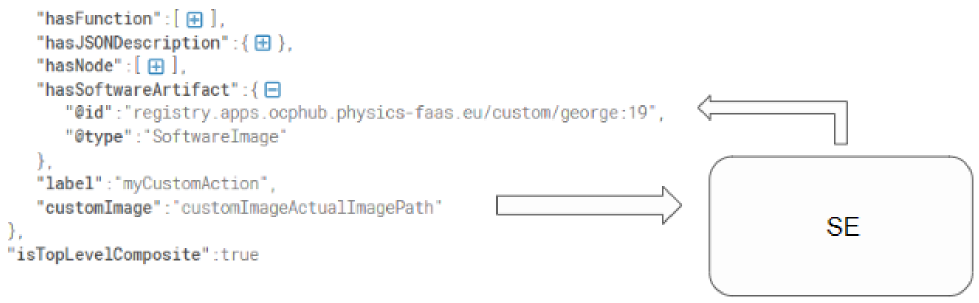
However one consideration here is that the deployable artefact in this case (registry.apps.ocphub.physics-faas.eu/custom/george:183) will be the image built from the DE (thus this flow that has only the semantic nodes) and not the manually imported image that is intended to be used. However this is solved by the PHYSICS Semantic Framework (Semantic Extractor component). When the created app graph will be fed into the SE, the latter will detect that the specific graph has a custom image (from the existence of the "customImage" tag) and will replace the "hasSoftwareArtifact" tag with the value of the "customImage" key value field included in the flow description. This way the app graph will have the correct info when forwarded to the PHYSICS platform for deployment and most importantly it will follow the same process as any other deployable PHYSICS function. An excerpt from the app graph and the replacement rationale appears in Figure 47.



Create Graph

HelloErrorParser	registry.apps.ocphub.physics-faas.eu/custom/george:138
ehealth Inference	
PEF LG ACTION	
MONITORING	
Flow 1	
orchestrator	registry.apps.ocphub.physics-faas.eu/custom/george:177
myCustomAction	registry.apps.ocphub.physics-faas.eu/custom/george:183

Figure 46: Custom Function Inclusion in App Graph



```
    "hasFunction":[ ⊞ ],
    "hasJSONDescription":{ ⊞ },
    "hasNode":[ ⊞ ],
    "hasSoftwareArtifact":{ ⊟
      "@id":"registry.apps.ocphub.physics-faas.eu/custom/george:19",
      "@type":"SoftwareImage"
    },
    "label":"myCustomAction",
    "customImage":"customImageActualImagePath"
  },
  "isTopLevelComposite":true
```

Figure 47: Software Artefact Replacement Process for Custom Function Image

Sizing Annotator Node

The Sizing Annotator node (Figure 48) is used for defining the function container size for execution on OW. Based on this node, the developer may dictate the memory size this container should have, as well as the set timeout. This information needs to be exploited by the platform layer when the respective flow will be declared as an action in the Openwhisk environment.

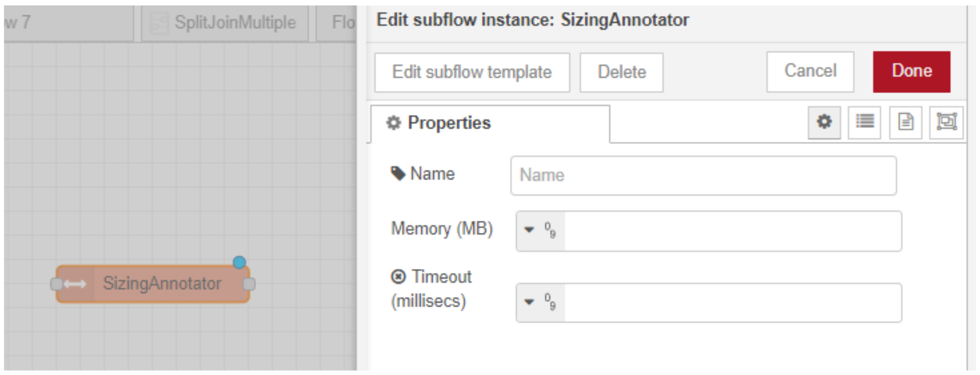
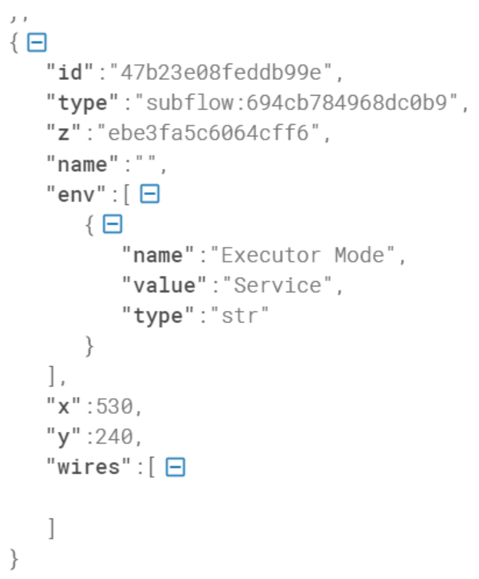


Figure 48: Sizing Annotator Semantic Annotator Node

Ways of processing the semantic node inputs

Selected values in annotation nodes remain in the JSON export of the flow, thus are retrievable and processable. The relevant JSON description of the node includes the name as well as the unique id of this subflow type ("id":"694cb….") and any included fields. Once this node is dragged and dropped in a flow, it is instantiated and the relevant selection (Service in this case) by the developer is an extra section included (Figure 49). The type of the node includes the unique id of the Executor Mode subflow ("id":"694cb….") as well as an id of the node instance itself ("id":"47b23…."). Through the correlation of the generic subflow ID or by the subflow name, the post processing can correlate this node to the Executor Mode type, while retrieving the set value of this specific instance.



```
,,
{
   "id":"47b23e08feddb99e",
   "type":"subflow:694cb784968dc0b9",
   "z":"ebe3fa5c6064cff6",
   "name":"",
   "env":[
      {
         "name":"Executor Mode",
         "value":"Service",
         "type":"str"
      }
   ],
   "x":530,
   "y":240,
   "wires":[

   ]
}
```

Figure 49: JSON Export of an instantiated Executor Mode Semantic Annotator

3.5. PHYSICS Core Application Graph Description Ontology

An ontology has been created in order to accommodate reasoning over the structure, operational parameters, characteristics and requirements of an application that is designed within PHYSICS. Such an application is deployable on the continuum that encompasses multiple heterogeneous resources, and different components may be instantiated and executed in different resources. The PHYSICS application ontology is devised in order to (a) guide the application design, (b) help define further characteristics that guide the deployment process, (c) provide the majority of the metamodel definitions for the reasoning engine of T4.1, (d) formalise the overall vision of the PHYSICS project in a widely understandable and interoperable way from the application perspective. The core concepts of the Application Graph Description ontology are explored in the subsections of this section, while the subsequent subsections explore the ontology as it extends beyond the core application description, as well as the way semantics are extracted from the defined flows.

3.5.1. Included concepts description and background

The core application ontology revolves around the concepts that have to do with (a) the data model of the Node-RED flows, (b) workflows as parts of an application or as a combination of applications, and (c) the interfaces between the parts of the workflow, commonly referred as nodes. As such, the core ontology mainly revolves around the hierarchy of the different classes of application components, based on core ideas related to the "Workflow" term, and applying them to the node-red flow model. The idea is to express all terms in a hierarchy that enables the inheritance of properties, as well as the ability to reason over application definitions. Additionally, the core ontology includes the "Pattern" term, while the different design patterns explored in section 4 are laid out as its subclasses.

The high-level modelling of the application description begins with the "Process" term, in the highly abstract domain of workflow descriptions, rather than the more specialised term in the domain of operating systems. The "Application" and "Workflow" terms are the direct children of the high-level "Process" class. The "Workflow" class is also abstract, and the specialisation used in PHYSICS is the "Flow" class, which encompasses flows as they exist in Node-Red and are deployable in the PHYSICS platform. The "Sub-Flow" class signifies the packaging of flows as reusable sub-flows in other flows, as well as the mandatory message wire inputs and/or outputs they have to include. The "Application" class encompasses any piece of logic that is deployable to the continuum, regardless of its size and characteristics. An "Application" individual is either a member/instance of the subclasses of "Application", or a composition of more individuals like this. The domains and ranges, which are the classes/types of the objects and values, of the properties related to "Application" are important, as they signify the ways in which simple or complex applications are composed, and by extension imply the possible or desired placements and deployment strategies.

As such, the PHYSICS "Flow" class is a descendant of both the abstract "Workflow" and the PHYSICS "Application" class, since one flow can by itself be an application. On a similar note, and because of the use of the FaaS model, the "Function" class, which encompasses function nodes, is a subclass of "Application". Functions can be used in Flow compositions, with most of their relationships and attributes being mostly the same as the Node-RED representation, and are considered to be the most basic atomic deployable entities. There are also Applications that do not fall in the "Flow" or "Function" classes, which are packaged as services, and are also used in Flow compositions. The "Service" class encompasses these applications, and serves as a connection between FaaS and other paradigms. The "External Service" is a direct subclass of "Service", which encompasses Services that are used in PHYSICS Applications, but are not controlled by the same entities, and may not even be part of the PHYSICS ecosystem. The deployment of an application may be handled differently, based on the external services it uses, as their operation is not under the control of the platform, and the application may need to be optimised so that parts of it have more performant and reliable access to those services. The "Standalone Container" is another subclass of "Service", which signifies the reuse or deployment of a complete containerized service as part of an application. A "Standalone Container" individual is controlled by the same entity that deploys the application, and the PHYSICS platform, enabling the deployment and reuse of applications that do not fall into the FaaS domain and are possibly monolithic or legacy or resource-heavy processes.

Since Flows are based on the Node-RED flow representation, a flow is composed of different nodes, and are all encompassed in the PHYSICS “Workflow Node” class. This class is a disjoint union of multiple classes, mainly “Function”, “Software Library Node”, “Service”, and “Sub-Flow”, because each Workflow Node can be a member of only one of these classes. Ready nodes that can be used in the visual composition tool (Node-RED) which utilise, at least a part of, a library are members of the “Software Library Node” class. Moreover, the members of the “Pattern” class are essentially pre-loaded subflows, created based on a software or cloud design pattern, and imported into the visual environment, as further explained in section 4. As such, Patterns are inferred to be part of the “Workflow Node” disjoint union. A Pattern, however, is to be handled differently from a mere Sub-Flow, in reasoning, optimization and deployment processes, as each pattern has its own characteristics, requirements and use-cases, with many pre-set properties, and additional configuration properties specific to each Pattern type.

Similarly, the interfaces between applications, and/or their components, are encompassed in the “Application Interface” class. Application Interfaces exist in an Application or Flow in various different forms. The topmost subclasses are “API”, “Connectivity Protocol”, and “Message Wire”, each with different properties and class hierarchies. Message Wires are essentially the connections between Node-Red nodes in Flows, and can be interpreted differently by the PHYSICS platform depending on the situation. APIs and Connectivity Protocols are found anywhere, and signify the way with which the applications and components interact and communicate, either in the same execution environment, or over the network, or over even the internet.

This overall class structure is the core of the PHYSICS Application ontology, and is meant to be able to describe the overall characteristics and structure of an application. It is also the skeleton upon which different properties are based, in order to be able to create descriptions that can be processed by the combination of OWL reasoning, custom rules reasoning, and further decision-making and optimization processes of the PHYSICS platform, so as to make optimal placements and deployments on the continuum.

3.5.2. Useful external ontologies

In the realm of linked data and open vocabularies, there do not exist ontologies that include terms about the kind of programming workflows that Node-RED and PHYSICS have per se. However, there do exist ontologies that define class hierarchies and characteristics of more generic workflows as dataflows, which most typically are Directed Acyclic Graphs. The modelling of the PHYSICS application class hierarchy is based upon Node-RED, using the following external ontologies as a guide, with the goal of connecting to or reusing terms from related ontologies wherever possible.

The Visual Modelling tool Model (*vmm* [38] for short) defines a vocabulary that includes the characteristics of a modelling tool, and originates from the study of UML software modelling tools.

Although *vmm* falls outside the scope of the application description, it provided a guide for modelling terms related to Node-RED in a more abstract and reusable way. The Abstract Workflow Description [38] (*Wfdesc* for short) ontology is an abstract description of workflows and their structures. *Wfdesc* is meant to be an upper ontology to be used for more specific workflow definitions, and as a way to express abstract workflows. The workflow description structure is useful as a basis for the modelling of the PHYSICS applications as workflows. The *Wfprov* [2] ontology is linked with the workflow descriptions of *Wfdesc*, in order to form a provenance trace of the execution of a workflow. However, *Wfprov* does not align with the modelling requirements of PHYSICS. Nevertheless, some of its terms can be used as a connection point with this ontology, in order to define which entity runs the workflow, what artifacts may be the results of the workflows, and how and where they are invoked on. The invocation of the steps of a workflow execution are described by the Workflow Invocation Ontology[3], which provides useful insights to the representation of an actual workflow execution, but does not align with PHYSICS application modelling. The different kinds of data-intensive activities that are common in data operations, and the ways in which each activity is implemented within a workflow, are found in the Workflow Motif Ontology[4]. However, this ontology is only usable as a guide for imprinting common observations in the application metamodel, and is not suitable to be reused.

In general, linking related ontologies with occurrences of equivalent terms in the manifests of the FaaS platform and the visual workflow programming tool of PHYSICS has proven beneficial in the semantic modelling of applications in PHYSICS. However, the only ontology that made a serious contribution to the PHYSICS application ontology is *Wfdesc*, which is imported in its entirety. Using its abstract Classes, a Workflow-related class hierarchy is organised into the core application ontology, as seen in the previous sub-section. After a search for terms useful to the application metamodel, no other ontology was found as capable to be linked or imported with the core application ontology. There is a possibility that *Wfprov* may play a more important role in the evolution of both the core and extended application ontology, as other entities are introduced, in order to better identify the involved people and platforms in the application definitions. In the end, the core ontology will have to be refined based on the evolution of the PHYSICS platform.

3.5.3. Domain model

In this subsection, some key triples of the core application ontology are laid out. Table 9 shows some of the key triples, showcasing the main classes seen in the ontology description, as well as key properties that connect them, and the meaning each connection has. The full ontology has been available online and can be visualised through WebVOWL[5].

[2] Wfprov Ontology, Available at: http://purl.org/wf4ever/wfprov
[3] Workflow Invocation Ontology, Available at: http://purl.org/net/wf-invocation.
[4] Workflow Motif Ontology, Available at: http://purl.org/net/wf-motifs.
[5]https://service.tib.eu/webvowl/#iri=https://drive.google.com/u/0/uc?id=1-9dnKP3Qr0oa9dEarp-hH2QfYvbXuDN4&export=download

Table 9: Key Concepts of the PHYSICS Core Ontology in an Object-Property-Value syntax

Object	Property	Value	Description - Comments
Application	includes Flow	Flow	Properties like this make the Application individuals that have them be compositions of multiple Application components.
Application	includes Service	Service	Similar to above. This is for when a Service is included in an Application. This may more commonly be used with "Standalone Container" individuals as values. Similar properties between Application and other Application components exist.
Flow	has Node	Workflow Node	Shows that a node is part of a Flow. This is a more high-level property.
Flow	has Function	Function	Sub-property of the above. Shows that a Function is included in a Flow. Derived from Node-RED.
Workflow Node	has Input	Application Interface	A Workflow Node, commonly a Function, has an Application Interface, commonly a Message Wire) as input. Derived from Node-RED.
Workflow Node	outputs	Application Interface	Similarly, a Workflow Node has message wires as outputs. Derived from Node-RED. There is an inverse property to this one in a manner similar to the two above. Commonly used with Functions and Message Wires.
Application	has Runtime (*sub-property of* has Dependency)	Runtime	A Function requires a software Runtime to run on. A hard dependency that needs to exist.
Flow	hasJSONDescription	string	Keep the original Node-RED-based JSON description intact, so that components like the orchestrator can do their own interpretation/translation of the Flow.
Template	has Instance	Instance	These are general-purpose, high-level classes, with two inverse properties. They encompass common cases of templates used for instantiation. In programming a class is a Template which has some objects as Instances. In
Instance	has Template	Template	

			virtualization or containerization, a VM or container image is a Template which has a running VM or container as an Instance, and so on. Any individual can be marked as Instance, or Template or both, if such a relationship applies.
Application	is Black Box	*boolean*	Application individual is a Black Box, if true. That means that an Application is not composed of any of its subclasses, and is not an Instance (has no Template). This should be inferred by a rule that finds that an Application is not an Instance of a generally valid and non-Black-Box Template Application, and not a member of any of the Application subclasses. Black Box Applications should not be used as Templates. Certain subclasses, like Flow, Function, should not be black boxes.
Application	is Top-Level & Composite	*boolean*	An Application is a Top-Level package that is a Composition of several individuals that are members of its subclasses, if true. This can apply to members of subclasses like Flow, as long as they are the top-level Application package in their case. This should be inferred by a rule that finds the top-level composite Application.
Deployable Software Artifact	has Execution Mode (*swArtifactExecutionMode*, different from the one below called *appExecutionMode*)	Execution Mode	A Deployable Software Artifact individual is a URI that targets the file/artifact in question directly. A Deployable Software Artifact individual has possible Execution Modes for some Resource types/classes (or Specific Resource individuals). The Execution Modes supported in PHYSICS by default are: ● Node RED Action ● OpenWhisk Sequence
Deployable Software Artifact	is Deployed On	Resource	

Deployable Software Artifact	artifact Type	Software Artifact Type	• Service Execution The Software Artifact Types supported in PHYSICS by default are: • Service Description File (e.g. compose file etc.) • Source Package • Docker Image
Application	has Deployable Software Artifact	Deployable Software Artifact	An Application Individual has a Software Artefact that can be deployed on some kind of resource.
Application	has Execution Mode (*appExecutionMode*, different from the one above called *swArtifactExecutionMode*)	Execution Mode	Mode of Execution for an Application individual. There can be 1 or more supported Execution modes for an Application individual. Like other properties, a definition like this on a sub-component overrides the generic definition on a composite.
Standalone Container	has Image (*sub-property of* has Template, means that individuals are also marked as Instance and Template respectively)	Software Image (*subclass of* Deployable Software Artifact)	A standalone Container (subclass of Service) is instantiated from a Software Image. The Software Image can be either a reference to an image from a repository (preferable), or reference to a Dockerfile.
Function *or* Flow (depending on the annotation)	*annotation* (as a Property)	*value*	Every annotation shown in Table 2 of section 3.4.3 is included in the core ontology as a triple, where each annotation is mapped to a triple like this.

3.6. PHYSICS Application Extended Ontology

This section proceeds with the overview of the extensions over the core application graph descriptions. There are various added terms that have to do with Resources, Quality of Service, Deployment Targets and various additional parameters, that (a) make a more direct connection between the application and the target resources and (b) provide a holistic view of the PHYSICS platform, from the perspective of the application design.

3.6.1. Included concepts description and background

The extensions of the application ontology go beyond the definition of an application as a workflow and the functions as the main workflow nodes. In order to facilitate the correspondence between applications and resources, the ontology includes high-level definitions of (a) resources for deployment, such as Cloud services, compute nodes and platforms, (b) locality constraints, (c) Raw Computational Resources, such as CPU and memory, as resource requirements, (d) QoS constraints to be matched to the Performance Benchmarks done on Resources or SLAs, and (e) other related terms that can be used as Application parameters and requirements.

The extended ontology is made with just one concern in mind: Finding the appropriate resource to deploy the application on. The Workflow-based "Application" class, its hierarchy of classes, and the entire core ontology in general are part of the extended ontology, with the core term of interest for the matching being the "Application" class. From the side of the deployment targets, the "Resource" class is the core term of interest, and it has its own class hierarchy. The majority of the parameters to be matched between an application and resources can be summed up into three axes, which are modelled as distinct class hierarchies, namely (a) "Raw Computational Resource", (b) "Locality" and (c) "QoS". These constitute the disjoint union of the "Deployment Attribute" class, which has "Application Requirement" and "Resource Attribute" as subclasses. An "Application" individual defines requirements which are later matched to resource attributes through reasoning, and through optimization whenever reasoning does not find exact matches, or whenever the matches are more than one.

Moreover, properties of the classes that constitute the disjoint union of "Deployment Attribute" may be sub-properties of the properties of this class in many cases. For these deployment attributes, there can be different ways to express a specific value or many possible values. A "Raw Computational Resource" member is a computational resource with processing power or storage of some sort, with examples including a processing unit like CPU, GPU, TPU, or temporary storage such as RAM, or Persistent Storage like HDD or SSD, or a MicroController like an Arduino, or a specialized card, or FPGA, or ASIC. The "Locality" class encompasses the different ways to either advertise the location of a Resource, or the desired deployment location or area of an application. This implies that the location of a Resource is easily retrievable, either by (a) annotating an available resource with its geolocation, (b) by including an expression of Location Information through DNS, so long as the resource is addressable by a domain name, following a standard such as RFC 1876[6] or reverse DNS[38], or (c) by means of a third-party service that may provide a semi-precise location based on IP address, utilising complex methods such as the landmark-based one seen in [39].

As such, "Locality" individuals can be expressions of geo-locations or geographical areas as GeoJSON[7], which may be combined with location names based on regions and/or reverse DNS. Big geographical regions, such as continents and countries can be named in advance in order to be able to match the Locality of Applications and Resources by simple string values directly. The "QoS" individuals that are Resource Attributes may be (a) defined by the Resource directly, (b) defined by the SLA of the Service, or (c) provided by a "Performance Benchmark", given a "Benchmark Scenario", for a given metric. For an Application, a "QoS" requirement may define preferred values or ranges of acceptable values for a given metric.

[6] A Means for Expressing Location Information in the Domain Name System (IETF RFC 1876), available at: https://datatracker.ietf.org/doc/rfc1876/

[7] GeoJSON specification, available at: https://geojson.org/

The “Resource” class mentioned in this section, along with its hierarchy, is a more abstract view of the resources, from the point of view of the application. The “Resource” class hierarchy includes many different kinds of deployment targets, which may stem from different domains. As such, different kinds of Cloud Services, Compute Nodes, Serverless Platforms and Storage Nodes can be described based on that, and this constitutes the connection with the PHYSICS Resource ontology created in WP5.

Additionally, there are Classes which express the performance profiling of Application instances, namely “Performance Benchmark”, “Load Generation Data” and “Performance Profile”. One Application may be associated with exactly one “latest Performance Benchmark” instance, one “Load Generation Data” object, and one “Performance Profile” object, for one location. An application can have one or more than one instance of these properties, each pertaining to a specific location. There is also a property for old performance benchmarks of an Application individual, in case historical data is kept in the knowledge base. The core idea is that an application undergoes benchmarking and performance analysis in order to produce a performance profile. That profile denotes how high the resource requirements of the application are, compared to the resources available in the deployment environment in each tested location. There are also accompanying metrics gathered from benchmarking the application during load at each location, which are expressed as data properties of a “Load Generation Data” object.

There are cases where information that follows the structure the ontology dictates will need to be defined in simpler ways than a, sometimes large, set of triples, as they are meant to be more common in usage than others. As such, the semantic annotations explained in section 3.4 are properties with “Function” or “Flow” as their domain and can sometimes be “shortcuts” into more complex descriptions, by defining rules. The resolution of such descriptions through rules are to be implemented as part of the integration of the components of the entire semantic block, in WP6.

3.6.2. Useful external ontologies

In order to create the overall structure of the PHYSICS ontology, including the connection with other domains, as well as the resource descriptions of T5.1, some external ontologies have been investigated. Most of the external vocabularies were used as guides for modelling parts of the ontology, while some connection points and inclusions are to be considered.

When it comes to application requirements that have to do with resources, i.e. the devices or platforms an Application can be deployed on, there exist some ontologies that encompass related terms. Some useful ontologies from the cloud computing domain, in that regard are the Ontology for Cloud Computing Instances[8] and the Ontology for Service Level Agreements[9]. However, these are minimal ontologies which only provide some ideas for the modelling of resources. When it comes to

[8] Ontology for Cloud Computing Instances, available at:
http://cookingbigdata.com/linkeddata/ccinstances/
[9]Ontology for Service Level Agreement for Cloud Computing.
http://cookingbigdata.com/linkeddata/ccsla/

Resources that are part of the Edge Computing and IoT domains, the W3C Web of Things recommendation provides insights into their hierarchy, relationships and usability guidelines. The Web of Things is a concentrated effort by W3C to create standards that will reduce the fragmentation of the IoT domain, by unifying many terms into a small set of ontologies. In the WoT architecture[10], the way to describe, expose, as well as consume a "Thing" are laid out. As such, terms from the WoT Thing Description[11] and Binding Templates[12] may be useful for laying out application requirements that are satisfied from special IoT equipment at the Edge. The inclusion of a subset of these ontologies is to be considered during the integration of the overall PHYSICS ontology, containing both the Application and Resource metamodel.

Since the continuum that PHYSICS is considered overlaps with multi-cloud service compositions (MCSC), and for cases where specific services or unique requirements (e.g., object stores etc.) are included in the application description, it is best to explore further works about MCSC on the semantic web. In [40], a unified cloud service description is proposed, which is meant to be able to accurately represent any cloud service, by consolidating all common characteristics and organizing them in nine dimensions/sub-ontologies:

- Service sub-ontology, representing the general information about a cloud service. (type, deployment model, category, evaluation, service reusability, etc.).
- Functional sub-ontology, which represents the overall functionality, as the set of operations offered by a cloud service.
- Technical sub-ontology, represents the technical aspects, that is, the way a cloud service is accessed.
- Participant sub-ontology defines the different actors (e.g., providers, consumers) participating in the description, composition, and invocation of cloud services.
- Interaction sub-ontology describes the services' behavioural aspects, and how cloud providers and consumers interact with services.
- Service-level sub-ontology comprises the QoS capabilities of each service (e.g., security, reliability, compliance).
- Legal sub-ontology, which represents the legal aspects and restrictions of the cloud service's usage.
- Pricing sub-ontology, which represents the fees and pricing models for consuming a cloud service.
- Foundation sub-ontology, which represents the general concepts (e.g., artifact, resource, location, time). Additionally, it addresses the resources control and visibility, the dynamic changes in the environment, and the environmental constraints.

This description can be used for automated service selection, based on characteristics and QoS constraints, expressed in Semantic Web Rule Language. In the same work, an algorithm for selecting

[10] Web of Things (WoT) Architecture, available at: https://www.w3.org/TR/wot-architecture/
[11] Web of Things (WoT) Thing Description, available at: https://www.w3.org/TR/wot-thing-description/
[12]Web of Things (WoT) Binding Templates, available at: https://www.w3.org/TR/2020/NOTE-wot-binding-templates-20200130/.

a combination of clouds is introduced, which considers both service semantics and multi-cloud settings, optimizing the usage of Multi-Cloud Environments. This work may prove useful to the whole PHYSICS project in general. A semantic engine for porting applications to the cloud and among clouds is presented in [40], which later became part of the mOSAIC framework. In order to aid in modelling cross-cloud deployments and optimize cloud deployments based on QoS metrics, requirements, prices and other SLA parameters, CloudPick [41] was introduced as an operational platform, which is based on ontologies. CloudPick may diverge from the PHYSICS project, however, it provides insights on both modelling resources and optimizing their usage.

Another important extension domain is that of locality requirements. The GeoJSON-LD[13] vocabulary can be used to encode coordinates or geographical areas, by just semantically annotating valid GeoJSON data with the vocabulary and indicating the appropriate class. In addition to that, the Vocabulary for Regions and Zones on Cloud Computing[14] is used to specify the suitable regions of Data Centres where application components can run and can further help model a Locality class hierarchy.

The approach of the creation of connections of Application descriptions with other domains, was done from the perspective of the requirements that an Application will have in order to be deployed and operational. Many of the connections and inclusions are to be further considered and introduced, in tandem with tasks T4.1 and T5.1. The goal is to connect the work of the Application ontology, along with its extensions, with the resource descriptions.

3.6.3. Domain model

This subsection proceeds to show some key extensions of the PHYSICS Application Ontology, in a manner similar to section 3.5.3. The extensions of the ontology provide connections between the Core Ontology Classes and other related domains, in order to enable the matching of an Application with the Resources it can or will be deployed on, through reasoning and optimization processes. The key extensions are laid out in Table 10 below and are also included in the WebVOWL link of the main ontology.

Table 10: Key Concepts of the PHYSICS Extended Ontology in an Object-Property-Value syntax

Object	Property	Value	Description - Comments
Application	requires	Application Requirement (*subclass of* Deployment Attribute)	Express a requirement of an Application, or application component.

[13] GeoJSON-LD format, available at: https://geojson.org/geojson-ld/
[14]Vocabulary for Regions and Zones on Cloud Computing. http://cookingbigdata.com/linkeddata/ccregions/

Resource	has Resource Attribute	Resource Attribute (*subclass of* Deployment Attribute)	Express the offering of a Resource.
Deployment Attribute	with Value	*string or number or boolean*	One specific value for a given Deployment Attribute individual. The subclass commonly used on the application descriptions is the “Application Requirement”. The attribute has to always has to also be part of one and only one of the following classes: ● Locality ● QoS ● Raw Computational Resource
Deployment Attribute	with Variance	*number*	Applicable only if the above exists, with a number value. It expresses the acceptable variance of the attribute.
Deployment Attribute	with Min Value	*number*	Minimum value, when expressing a numerical range.
Deployment Attribute	with Max Value	*number*	Maximum value, when expressing a numerical range.
Deployment Attribute	with Possible Value	*string or number*	One or more possible values for the given attribute.
Deployment Attribute	with Prefered Value	*string or number*	Preferred value, but not the absolute target.
SLA *or* Performance Benchmark	has Derivative QoS	QoS (*inferred subclass of* Deployment Attribute)	A QoS attribute derives from the SLA of a Resource, or Performance Benchmarks done on the resource.
Application	has Owner	User	The owner of the application. Extracted by the user id used in the call to deploy the application.
Application Management Entity (*subclass of* Management Entity)	manages Application	Application	These properties express Management Entities that manage Applications or Resources. Management Entities include management systems, orchestrators, operations support platforms, etc. for Applications or Resources. The common characteristic of these management entities is their (softwarized) configurability via APIs.
Resource Management Entity (*subclass*	manages Resource	Resource	

of Management Entity)			
Application	has Platform Dependency (*sub-property of* has Dependency)	Operating System *or* Management Entity *or* Raw Computational Resource *or* (some specific Resource *subclasses*:) Storage Node *or* Serverless Platform *or* Cloud Service	Property to express hard platform dependencies. This goes beyond runtimes, libraries and frameworks, as it has to do with software or hardware platforms, as well as specific hard dependencies of those platforms that may affect the Application (component). Applications cannot be deployed on Resources if their dependencies are not fulfilled.
Pattern	enhances	QoS	Patterns can change some QoS parameters. The QoS parameter and its metric and values are an expression of the expected difference to baseline. (Further modelling regarding Pattern and QoS relationships to be done.)
Pattern	deteriorates	QoS	
Deployable Software Artifact	has Artifact Dependency	Operating System *or* Management Entity *or* Raw Computational Resource *or* (some specific Resource *subclasses*:) Storage Node *or* Serverless Platform *or* Cloud Service	Property to express hard platform dependencies, but for an Artifact that is built from an Application definition. This goes beyond runtimes, libraries and frameworks, as it has to do with software or hardware platforms, as well as specific hard dependencies of those platforms that may affect the specific Deployable Artifact.
Application *(Instance or Component)*	has Performance Profile	Performance Profile	This property assigns Performance Profiles to an Application Instance or Component. Each Performance Profile for an Application consists of the Location it applies to, and the Resource Requirements Classification.
Performance Profile	has Location	Location or Locality	The classification of the requirements in terms of various resources, based on the overall resources available, and conditions present, at a location. An Application may be classified to have *"low"*, *"medium"* or *"high"* requirements for computational resources: - *CPU* - *Memory* - *Network Traffic Reception* - *Network Traffic Transmission*
Performance Profile	defines Resource Requirements	Resource Requirements Classification (Various data properties regarding the requirements or usage of	

		computational resources)	- *File System Reads* - *File System Writes* - *GPU (optional, if GPU is, in fact, required)* Each classification results from performance analysis done on the specific Location or Locality the performance profile is created for.
Application *(Instance or Component)*	has Load Generation Data	Load Generation Data (Collection of data properties)	Various data properties with scalar values, regarding the results, and some inputs, of the latest benchmark done during generated load, at a specific location. These properties include resource usages, averages and standard deviation values of various metrics, such as latency, duration and delay of the application's operation.

3.7. Semantic Extractor Process and Implementation

The Semantic Extractor is the component tasked with (a) extracting different annotations from Flows and Functions and (b) transforming the JSON representation of Application components from Node-RED into a JSON-LD representation compliant with the PHYSICS Ontology (as seen in Section 3.5) as well as its extensions (as seen in Section 3.6). It now supports the representation of different Application structures, requirements and annotations, and is also made to be extensible in terms of the supported annotations. The outputs are fed to the Inference Engine of WP4, as well as the deployment pipeline that follows. The workflow of the semantic extractor can be summed up in the following steps:

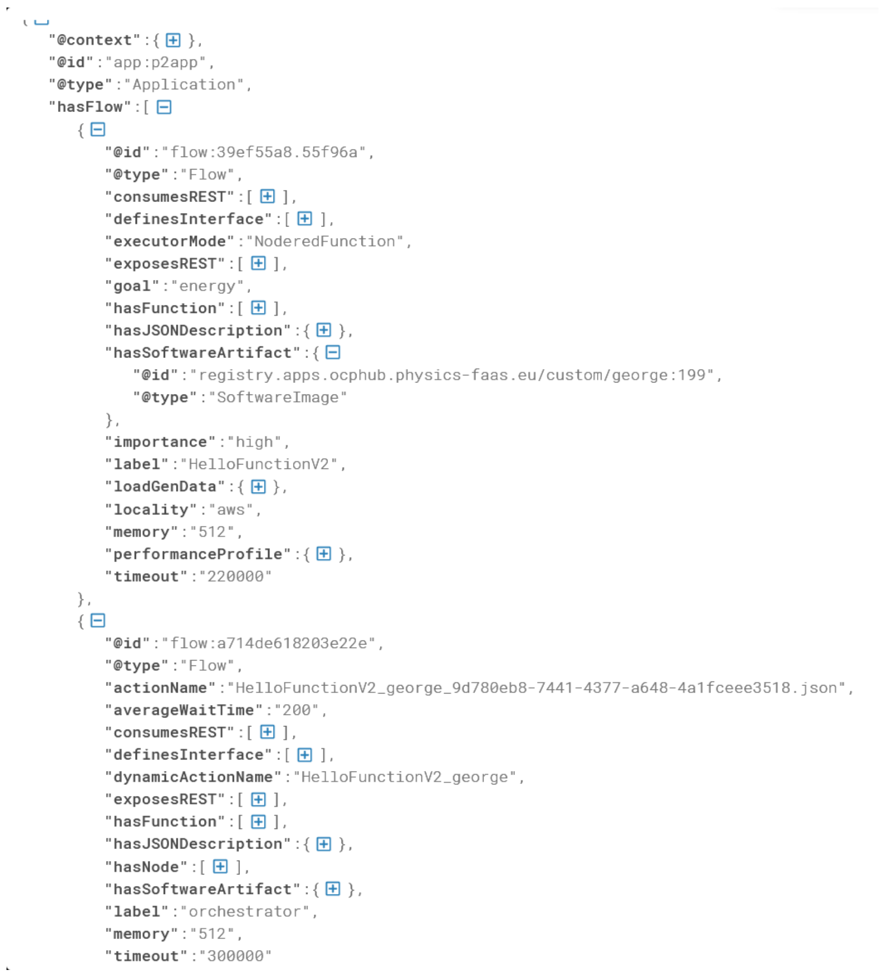
- The Design Environment (DE) triggers the Semantic Extractor (SE) (Step 4 of the process described in Section 2.5.1) to begin its workflow, on the flows from Node-RED with specific IDs, and the corresponding deployable artefact locations. This is done by calling the */extract* endpoint with an HTTP POST, with the body including all Node-RED flows (main logic) and subflows (re-used logic or annotator nodes) selected together. The body also includes the display name and branch name of the application, the ID of the deployable artefact to be created, the type of the artefact, and the corresponding deployment action name. The semantic extractor responds with an acknowledgement that the extraction trigger has been received.
- The SE creates the top-level JSON-LD representation of the application based on the contents of the DE output with some basic properties. It also segregates the different parts of the DE output in order to properly analyze them with different pieces of logic, especially flows and subflows.
- The included semantic annotations (more details on their types in Section 3.4) are extracted from each Flow as well as the Functions included.
    - For the Flows, the annotations are extracted by recognizing annotator nodes, which are represented as Node-RED subflows in the DE output, and setting the values that are set in their instance properties for the corresponding properties.

  - For the Functions, the single line comments are parsed, in order to create the corresponding annotations, as explained in section 3.4.1. The extraction is done by recognizing the explained syntax within the single-line comments. The processing of the single line comments within the code follows the example of this project[15].
- The overall structure of the Flow is parsed and transformed in order to be compliant to the ontology. First, the context that introduces the ontology and the various addressing schemes are introduced as a base structure, as pre-computed in step 2. Then, the flow is filtered, and each node and component is classified as an entity that belongs to a class defined by the ontology, and the properties that connect those entities are added in order to form the proper triples (in RDF terms).
- Once the flow is fully transformed, the SE attempts to include additional data by making the appropriate requests to the Performance Evaluation Framework (PEF) of T4.2. As long as the developer has gone through the Performance Pipeline process described in Section 2, PEF will have been enriched with the results of that execution. Thus it will provide the SE with raw data regarding the performance profile, as well as load generation results and input data, as a response, given combinations of flow and branch IDs as inputs, for the various available OW cluster locations. The SE then parses this data in order to create the appropriate ontology-compliant objects, as well as property-value pairs for the corresponding flows. As a result the flow objects include the relevant results of the PEF where applicable, in the JSON-LD structure.
- The structure that is created up until that point is validated via a json-ld processor, in order to ensure the proper data structure is created, by cleaning and normalizing the document, and to enforce the JSON-LD standards. The result is a JSON-LD document in array-notation, that complies with the ontology, and can thus be introduced as it is in a triple store or reasoning engine. This description also includes the original representation as it was taken from the DE, for the cases where the raw representation is needed for orchestration and/or deployment purposes.
- Each JSON-LD document in the aforementioned array-notation structure that is a flow representation is pushed to the Inference Engine, once it is processed. The reasoning engine then continues with its own process. The inference engine will build upon the representation the extractor provided, by imposing an OWL inference / entailment regime, and using further inference capabilities that build upon it.
- The Application ID, and different component IDs that are ready are sent to the Design and Control UI. The user is notified that the selected Flow and the corresponding Application components are ready.

The data transformations are done either manually or with the help of the JSONata library[16], and the JSON-LD processing is done via RDF.js[17] and mainly its JSON-LD context parser[18] in order to provide a valid, compact and compatible representation of the Application description. The semantic extractor has been developed and tested as a Node-RED flow that processes the data outputs of DE

[15] Dependency Aware FaaSifier project, available at: https://github.com/qngapparat/daf
[16] JSONata library, available at: http://docs.jsonata.org/overview
[17] RDF.js library, available at: https://rdf.js.org/
[18] https://github.com/rubensworks/jsonld-context-parser.js

and PEF, regarding flows from the instances of Node-RED used in a PHYSICS DE. It is packaged either as a Node-RED extension or as a standalone Node.js service. Since the way the internals of the extractor operate are heavily dependent on the structures that the ontology imposes, the extractor and the ontology have been developed in parallel, in order to ensure that the ontology is well-suited for the project's needs, and that the extractor is fully compliant with the expectations of the dependent tasks.



```
{ ⊟
  "@context":{ ⊞ },
  "@id":"app:p2app",
  "@type":"Application",
  "hasFlow":[ ⊟
    { ⊟
      "@id":"flow:39ef55a8.55f96a",
      "@type":"Flow",
      "consumesREST":[ ⊞ ],
      "definesInterface":[ ⊞ ],
      "executorMode":"NoderedFunction",
      "exposesREST":[ ⊞ ],
      "goal":"energy",
      "hasFunction":[ ⊞ ],
      "hasJSONDescription":{ ⊞ },
      "hasSoftwareArtifact":{ ⊟
        "@id":"registry.apps.ocphub.physics-faas.eu/custom/george:199",
        "@type":"SoftwareImage"
      },
      "importance":"high",
      "label":"HelloFunctionV2",
      "loadGenData":{ ⊞ },
      "locality":"aws",
      "memory":"512",
      "performanceProfile":{ ⊞ },
      "timeout":"220000"
    },
    { ⊟
      "@id":"flow:a714de618203e22e",
      "@type":"Flow",
      "actionName":"HelloFunctionV2_george_9d780eb8-7441-4377-a648-4a1fceee3518.json",
      "averageWaitTime":"200",
      "consumesREST":[ ⊞ ],
      "definesInterface":[ ⊞ ],
      "dynamicActionName":"HelloFunctionV2_george",
      "exposesREST":[ ⊞ ],
      "hasFunction":[ ⊞ ],
      "hasJSONDescription":{ ⊞ },
      "hasNode":[ ⊞ ],
      "hasSoftwareArtifact":{ ⊞ },
      "label":"orchestrator",
      "memory":"512",
      "timeout":"300000"
```

Figure 50: Application Graph Semantic Output towards the Reasoning Framework of WP4

An example of a translated application graph appears below, consisting of two functions. The semantic extractor is able to understand the DE input (Node-RED flows of these functions) and translate them to the concepts of the ontology, feeding that information into WP4 and the Reasoning Framework. A number of different annotations are included in the specific example and are transferred to the app graph, as this appears in Figure 50, including sizing, importance, execution mode, locality etc. The extractor has also retrieved and included performance profile and benchmarking information (loadGenData tag) that was available for one of the functions in the defined application.

4. Design Patterns

4.1. Introduction

The concept of patterns is introduced in order to aid in the creation of reusable partial implementations that can be dragged and dropped in the developer environment, and thus speed up the application creation process as well as minimize the knowledge barrier for the developer. Patterns have been a very useful tool for dictating design principles as well as driving abstract implementations for a specific domain [42]. Patterns, or architectural styles, are considered a key element of cloud software application development in order to increase code reuse and ensure a number of issues such as fault tolerance or performance in distributed environments [43]. Especially for developers coming from more typical programming backgrounds, the use of asynchronous, function-oriented programming can be considered challenging. Therefore, the grouping of even simple sequences in generic, reusable collections can prove useful in this transition.

Numerous definitions are included in the existing literature with relation to patterns [44]. The one followed in the context of PHYSICS is that a pattern is "*a proven series of activities which are supposed to overcome a recurring problem in a certain context, particular objective, and specific initial condition*" [45]. For this reason, the defined and implemented patterns in the context of PHYSICS come with a complete documentation around these fields. However, this is not sufficient, since as can be found in typical detailed pattern documentations [3], patterns may come with specific requirements for their application, as well as drawbacks and limitations. Patterns do not fit to all cases, typically have a set of parameters to be determined, weak points and strong points, limitations and can be compensated by other patterns. Therefore, this information is also included on a pattern description level. This is needed in order to inform the developer about these issues, as well as enable the combination of complementary patterns or the avoidance of using conflicting ones.

In the context of this work, the concept of a pattern fits very well with the generic ability of Node-RED to group entire workflows in subflow nodes and add properties and configurations on top of them. This functionality then is included in the environment palette as a single node, hiding all underlying complexity. The developer can directly drag and drop that node in their flow and use it directly, a feature that significantly speeds up development. It is necessary to stress that a pattern implementation may be executed in either a service or function (sequence) mode, may reside on the client or on the server side, but they need to be configurable, reusable and parametric.

Sources of pattern definitions

The main sources from which relevant pattern definitions have and will be extracted in the context of PHYSICS include (Figure 51):

- cloud and FaaS specific patterns [46] that need to address specific domain issues (e.g. minimize invocations, improve some features like state, cold starts etc.)

- the project's research and innovation scope and objectives
- the project's use cases and their specific problems and considerations

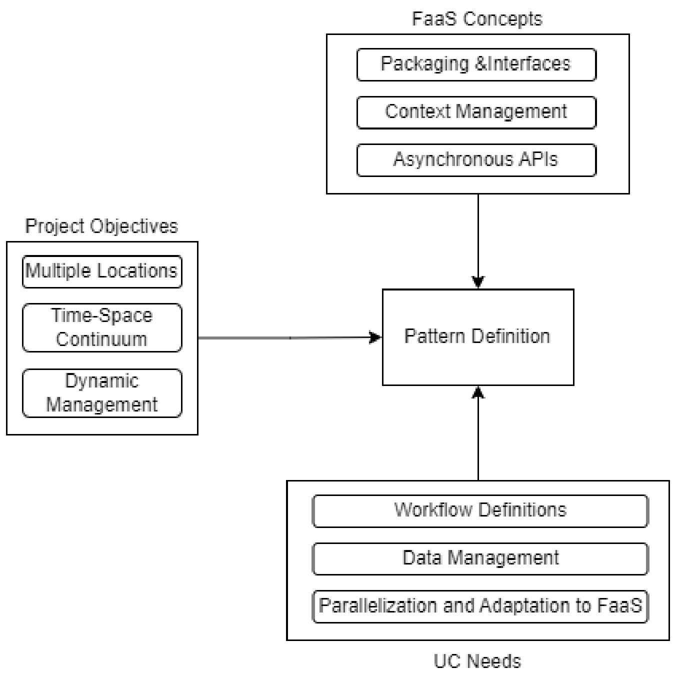


Figure 51: Pattern Definition Sources

Following, further details per pattern are portrayed, including their relation and usage within PHYSICS, their relation to requirements, as well as implementation and experimentation details where appropriate. Initially a brief description is given on a table template basis, whereas further descriptions are provided in subsequent sections.

4.2. Patterns usage and inclusion

4.2.1. Pattern Use Cases

The specialisation of the use case of pattern inclusion appears in Figure 52.

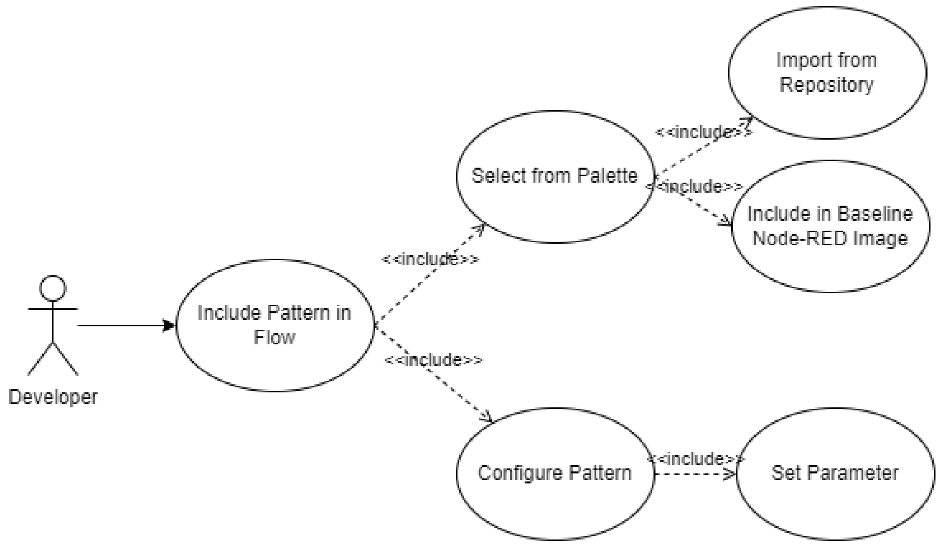


Figure 52: Patterns Inclusion Use Case

In order for the developer to include a pattern in their application flow, they need to drag and drop it in their flow. Following its incorporation, the developer needs also to instantiate any input parameters needed by the pattern. These parameters may regulate usage and application of the pattern or pure functional aspects.

This functionality is related to the generic requirements expressed in D2.2 and more specifically:

- Req-3.3-PatternApplication: The exposed patterns need to be seamlessly integrated into the application structure. To this end, they need to be exposed in the Visual environment and directly integrated into the application workflow, while configurable, if applicable, through the environment. The mechanism for enforcing the pattern should be seamlessly deployed along the application in the FaaS platform.
- Req-3.3-PatternDocumentation: The exposed patterns from T3.3 need to be completely documented so that the application developer is aware of their structure, operation, effects and outcome.

4.2.2. Patterns Incorporation

In order to enable the aforementioned incorporation, the patterns need to be available in the environment (Figure 53).

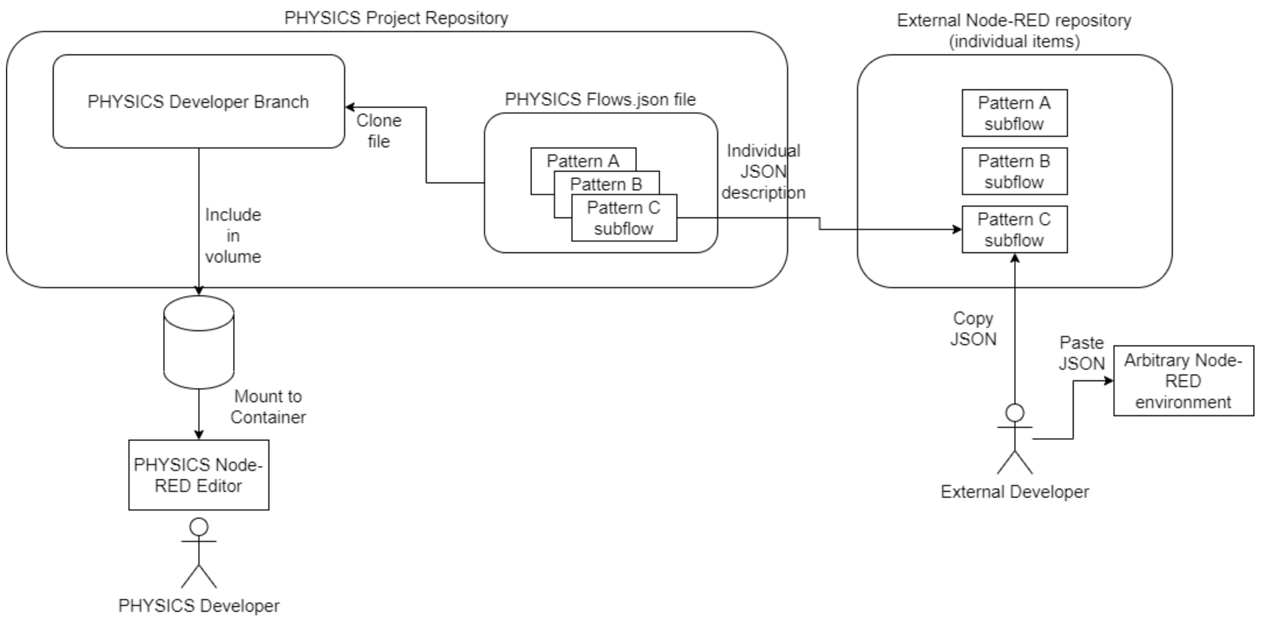


Figure 53: Incorporation of a pattern in PHYSICS and beyond

To achieve this, the pattern description subflow (in JSON format) needs to be available in a reachable repository and included in the environment of the Baseline Node-RED image created by PHYSICS. For cases of patterns that are deployed as separate services, a relevant description node should exist in order to point to a template docker file for the service. However, in order to extend their usage outside PHYSICS, since many of these patterns will cover generic usages, the specific individual JSON descriptions of the patterns will also be made available to external Node-RED repositories[19] so that they can reach wider audiences. In this case, any external developer, not belonging to the PHYSICS ecosystem, or not having incorporated the overall PHYSICS functionality, will be able to cherry pick and select an individual subflow that may prove useful.

Following, details on the implemented patterns for this period are included in the following paragraphs. For each pattern, a template table for pattern description is included, followed by subchapters that describe details on a number of needed information such as implementation details, examples of usage, limitations, variations etc.

4.3. Node-RED flow as function pattern (OWSkeleton)

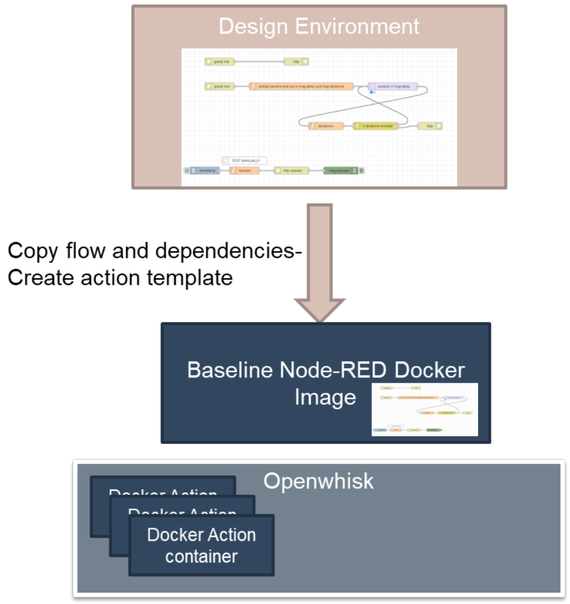
4.3.1. Pattern template description

The Node-RED flow as function template description is provided in Table 11.

Table 11: Template Description for Node-RED flow as function Pattern

Pattern name	Node-RED flow as function (OW Skeleton)

[19] https://flows.nodered.org/

Relation to Requirements	Req-3.1-WorkflowDef, Req-3.1-CustomDockerImages, Req-4.2-FaaSBenchmarking (to check delays inserted by the runtime), Req-6.1-stateless, Req-3.3-PatternApplication
Source of pattern	Development abstractions for all UCs, Execution mode defined in PHYSICS
Pattern Context	Use Node-RED as an extra and more abstract runtime for serverless functions
Pattern Underlying Problem	Asynchronous and functional programming style is not very intuitive for developers with different backgrounds. Workflow definitions of current FaaS platforms limited in terms of abilities and large flows complexity of description
Pattern Usefulness/ Objective	The pattern enables the usage of Node-RED as a generic programming environment for functions and workflows, exploiting the abundance of available nodes as well as easy local testing. By creating the flow in the editor one can debug most significant errors. Then by utilizing also the PHYSICS DevOps processes, to create the deployable artefact and deploy the flow as an executable function in a relevant Node-RED runtime image. Through this runtime, the developer may create in a more user friendly manner complex flows, as well as exploit the baseline functionality of Node-RED and the community repositories and nodes. They are also not limited by constraints of FaaS platform workflow specifications.
Schema	 Design Environment Copy flow and dependencies- Create action template Baseline Node-RED Docker Image Openwhisk Docker Action Docker Action Docker Action container
Prerequisites	Adaptation to the I/O specification of an Openwhisk action.
Condition of application	Need to include readymade logic from the Node-RED ecosystem in an easier manner than native function creation in one of the available Openwhisk runtimes
Input parameters	flows.json file of the flow

Output parameters	deployable container image to be registered at the target OW platform
Included functionality	Wrapping, Context Reuse functionalities
Pattern limitations	Higher startup times compared to native runtime environments, context reuse issue
Linked to Pattern	Node-RED orchestration
Indicative Domains of Applicability	Generic means of writing function logic

4.3.2. Pattern implementation details

The Openwhisk specification indicates that any docker image can be included as a function provided that it has a REST interface with two endpoints (a POST /init and a POST /run), listening on port 8080. The /init method is used to initialize the environment of the function (if needed) while the /run is used for the actual execution of the function and for passing any input arguments. To this end, a relevant subflow has been created (OWSkeleton in Figure 54) that includes the two methods and an example hello world function. The hello world function can be replaced by one or more functions linked in a flow, as long as the wiring is maintained (first node in the flow to be linked to the /run endpoint and final node of the flow linked to the http response node). Any available Node-RED node can be used (or externally imported) while creating the inner flow, thus enabling the incorporation of arbitrary logic from the developer. The same applies for function wirings, that can follow any type of wiring and workflow orchestration logic supported by the Node-RED runtime or empowered by the patterns described in this document. One key feature in this process is the Jenkins pipeline mentioned in Section 2.6, that guarantees that whatever dependency (e.g. extra node or library used in the Node-RED editor environment, settings or credentials files) is also included in the template Docker image that is generated as a result of the DevOps process. The OWSkeleton node appears in Figure 54.

One key consideration in this case is the need to have separate function names for each flow. So each flow , in order to be generated, starts from the same baseline Node-RED template image, the specific flow file as well as other node dependencies are copied in the new image from the development environment and the image with a discrete name is registered as the one to trigger on the specific action activation. This is the only way since fetching dynamically the flow would not work for cases of paused containers reuse in Openwhisk. The respective generic Node-RED template image would be treated as a single runtime image by Openwhisk. This means that once OW had executed an Action of the Node-RED type, a subsequent call to another Node-RED action (but of a different flow) could reuse the same paused container that also includes the context of the initial action. Any context created in the first execution would be visible in the second one. This is of course dangerous from a security point of view but also from a functional point of view if the two different instances share any context variable name that is not initialized as part of the flow logic. In case different flows were meant to be executed, this again will not work since the init method, that could potentially be used to fetch the new flow, is not re-executed in the warm execution case (container re-use) by Openwhisk.

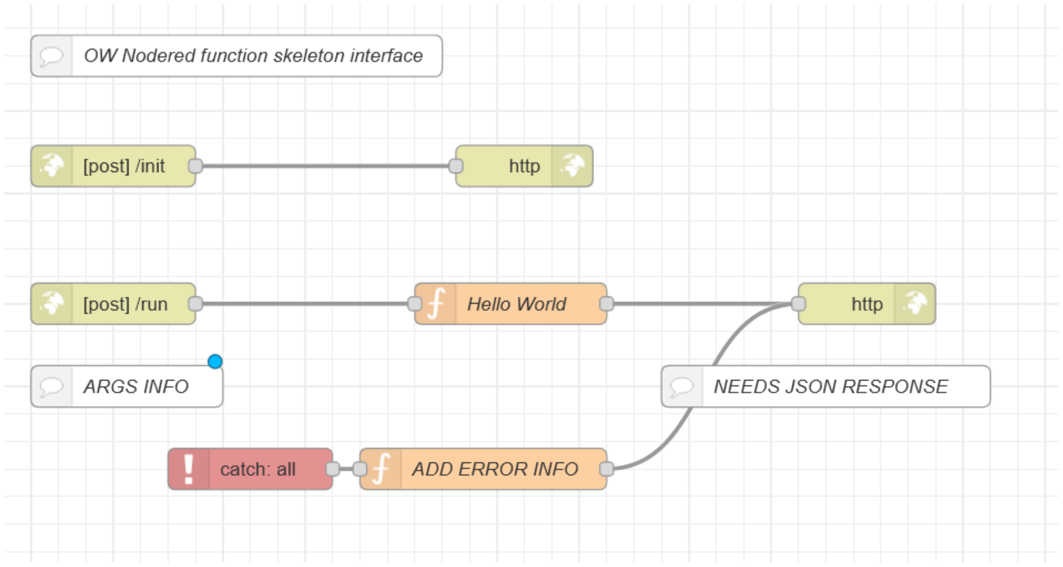


Figure 54: OW Skeleton Pattern

Improved Logging through an Error catching process

Another point of attention is the need to include detailed error capturing for function related errors as well as propagation of these to Openwhisk following the according specification. This is especially critical for solutions like PHYSICS, in which many intermediate layers (e.g. WP3, 4, Openwhisk, Kubernetes) are intertwined. We also need to avoid the fact that different errors may occur during an invocation and the possibility of one internal error to cause other cascading ones. In the specific example (Figure 55), the logs display a top level error of a function timeout:

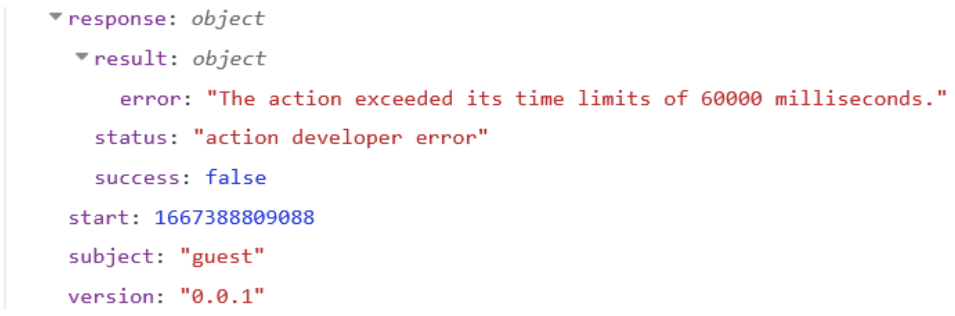
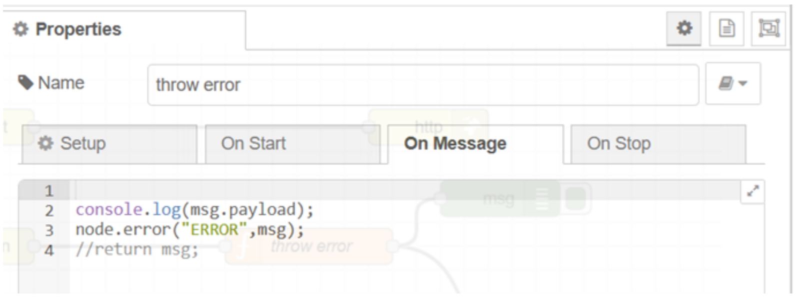

```
▾response: object
  ▾result: object
      error: "The action exceeded its time limits of 60000 milliseconds."
    status: "action developer error"
    success: false
  start: 1667388809088
  subject: "guest"
  version: "0.0.1"
```

Figure 55: Indicative example of Timeout Error Report

However if we examine the execution carefully, we may observe that the reason for this timeout is the fact that the function execution has stalled inside the Node-RED runtime for another reason (e.g. wrong input argument). This is in principle an error on the function creation itself, which is twofold. Initially it should catch that error and return a relevant error message to Openwhisk, indicating that the execution has finished in an erroneous way. This way one can avoid waiting until the function timeout to detect the erroneous execution, leading to faster debugging. The Openwhisk interface dictates that the response object should contain an error field in order to indicate this. In that message it could also indicate more details of the internal function error. If editing the function from Node-RED, then a suitable insertion of such an error would be through the proper configuration of the msg.payload before returning the error response object, by including a relevant error field in the msg payload (msg.payload={'error':'Some Error Message'}).

If the flow needs to capture an error from any node it needs to use the built-in Catch node of Node-RED and then add the msg.payload.error field in a subsequent function node. According to (https://nodered.org/docs/user-guide/writing-functions) if the function encounters an error that should halt the current flow, it should return nothing. To trigger a Catch node on the same tab, the function should call node.error with the original message as a second argument, as depicted in Figure 56.

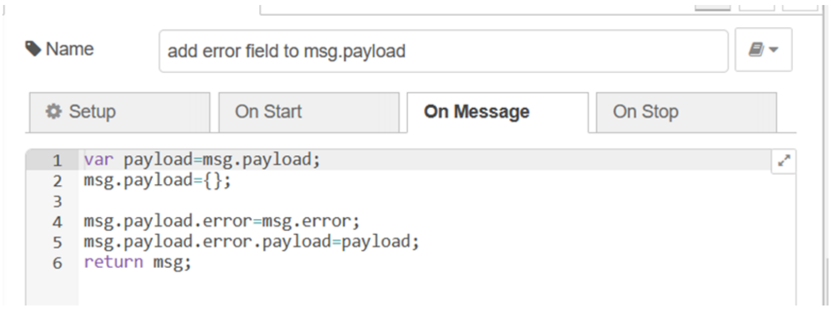


Name: throw error

```
console.log(msg.payload);
node.error("ERROR",msg);
//return msg;
```

Figure 56: Error Throwing Structure

We can then write some generic code to include that information in the msg.payload, which is the response object towards Openwhisk (Figure 57).



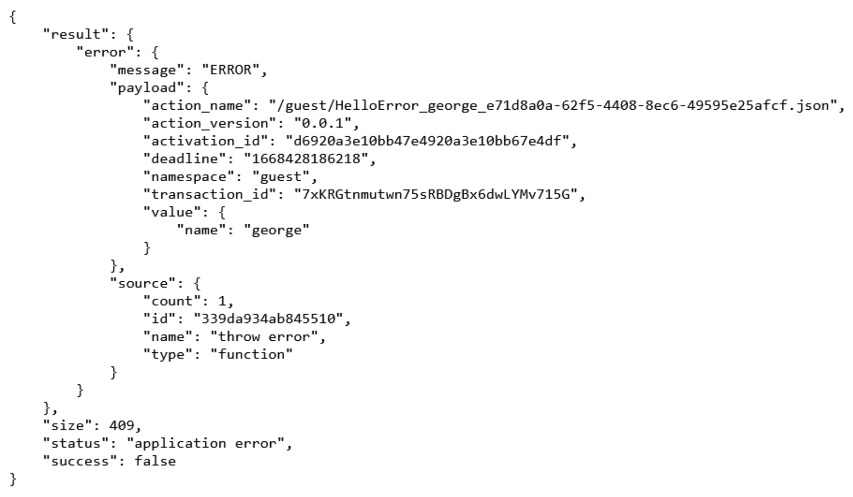
Name: add error field to msg.payload

```
var payload=msg.payload;
msg.payload={};

msg.payload.error=msg.error;
msg.payload.error.payload=payload;
return msg;
```

Figure 57: Preparation of Response For Detailed Error Report

In this way we will get a lot of information regarding the error including the payload of the msg that triggered the error in the first place as well as the node in which it occurred. The according output in the activation result appears in Figure 58.



```
{
    "result": {
        "error": {
            "message": "ERROR",
            "payload": {
                "action_name": "/guest/HelloError_george_e71d8a0a-62f5-4408-8ec6-49595e25afcf.json",
                "action_version": "0.0.1",
                "activation_id": "d6920a3e10bb47e4920a3e10bb67e4df",
                "deadline": "1668428186218",
                "namespace": "guest",
                "transaction_id": "7xKRGtnmutwn75sRBDgBx6dwLYMv715G",
                "value": {
                    "name": "george"
                }
            },
            "source": {
                "count": 1,
                "id": "339da934ab845510",
                "name": "throw error",
                "type": "function"
            }
        }
    },
    "size": 409,
    "status": "application error",
    "success": false
}
```

Figure 58: Full Activation Result including Error Details

This ensures that accurate logging information is propagated both to the Openwhisk environment and therefore to the final user of the function. All the aforementioned functionality has been included in the updated Skeleton template of the pattern, in order to assist developers in the process.

4.3.3. Pattern examples of usage

This skeleton is used across many different patterns in this document, especially the Node-RED Split Join pattern. In a typical standalone usage external to PHYSICS, for parsing the above hello world flow, one should create the flow file in a relevant Node-RED server environment, export the flows file as a JSON from the environment and then build a relevant image. Extra Node-RED packaged nodes included in the palette by the developer should be manually inserted in the dockerfile, in a manner similar to the docker file presented in Code 3. Then the image should be built, pushed in a docker registry and registered with Openwhisk with the –docker flag (Code 5). Now the action is accessible for execution either via wsk cli or HTTP.

```
docker build --no-cache -f Dockerfile -t username/action_name .
//The no-cache argument is needed since it should retrieve the flow each time from the repo in case of //changes


docker push username/action_name


wsk action create action_name --docker username/action_name --web true
//--web true should be used if one wants to invoke the action directly from an HTTP endpoint

//examples of invocation
OW CLI mode: wsk action invoke action_name
HTTP: GET http://serverIP:3233/api/v1/web/<namespace>/default/<action_name>.json
```

Code 5: Example build and registration externally to the PHYSICS platform

However in the context of the PHYSICS design environment, this creation is performed by the DevOps processes described in Section 2, that includes an extraction of the node dependencies, the inclusion of the relevant flows file and the publication and registration of the image automatically.

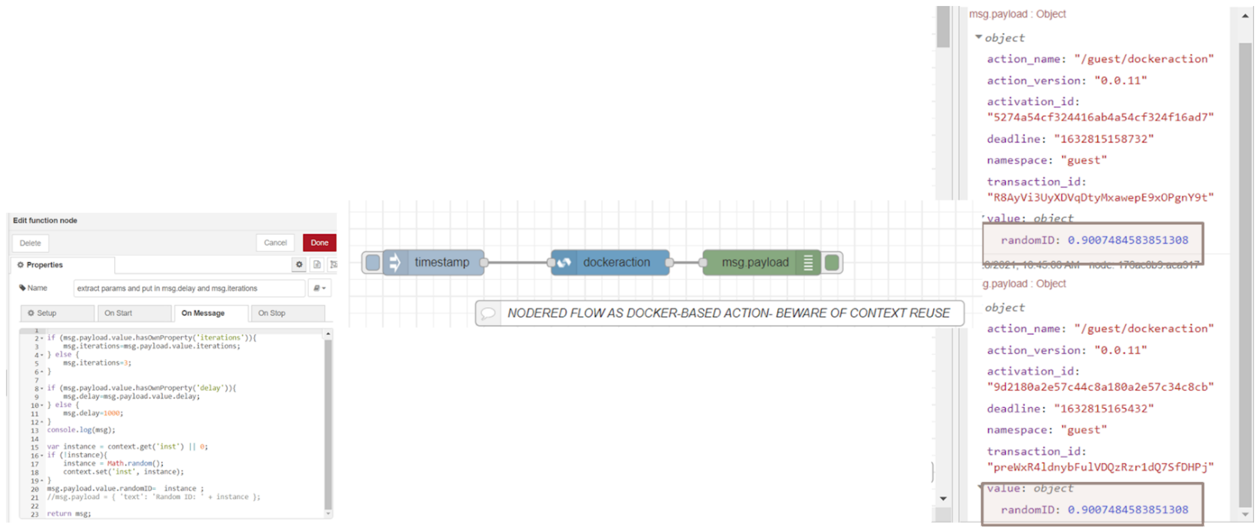
4.3.4. Pattern necessary adaptations (application side)

The main adaptation from the application side in the flow refers to the way the arguments are passed from Openwhisk to the inner function logic during runtime. The incoming parameter (and any other parameters) are wrapped around a specific JSON structure by Openwhisk before passing them in the inner functions. Thus any input arguments that are included in the POST body of the call are automatically wrapped around a msg.payload.value field when executing in the Node-RED flow function. This needs to be treated accordingly by the developer, so that they can obtain and process the values. The same applies for the testing process inside the Node-RED editor. The developer should directly use this structure in the testing environment in order to avoid errors during deployment of the action in OW. One final point of attention is the fact that the logic needs to return a JSON object according to the OW specification.

4.3.5. Pattern variations

As mentioned previously, one of the issues relates to context reuse. Typically executed function containers remain in the Openwhisk environment in a paused mode for a period of time after execution (~12 minutes). If during that time another function invocation comes for the same function, the paused container is reused. This means that any variable that is initialized in the first invocation will maintain the value if not re-initialized. In order to demonstrate this aspect, an experiment was performed in order to check whether the context variables of a previous execution were available in a subsequent execution. The process appears in Figure 59. Initially we created a flow that is executed as a Node-RED docker action. In the flow, a random id is assigned to a context variable, if it does not exist. Then this random id is returned as part of the response to the caller. Indicatively, triggering twice the execution of the action, with a sufficient delay so that the first one has finished before the second request arrives but below the paused container lingering period, has a result of returning the same random id in both calls.

This context reuse example creates opportunities however for creating more than one pattern variations with relation to context and state reuse, in order to cover for all possibilities and use cases (needing or not the pre-existing context). The three variations implemented or foreseen are listed below.



Name: extract params and put in msg.delay and msg.iterations

```
if (msg.payload.value.hasOwnProperty('iterations')){
    msg.iterations=msg.payload.value.iterations;
} else {
    msg.iterations=3;
}

if (msg.payload.value.hasOwnProperty('delay')){
    msg.delay=msg.payload.value.delay;
} else {
    msg.delay=1000;
}
console.log(msg);

var instance = context.get('inst') || 0;
if (!instance){
    instance = Math.random();
    context.set('inst', instance);
}
msg.payload.value.randomID=  instance ;
//msg.payload = { 'text': 'Random ID: ' + instance };

return msg;
```

```
msg.payload : Object
▾object
  action_name: "/guest/dockeraction"
  action_version: "0.0.11"
  activation_id:
  "5274a54cf324416ab4a54cf324f16ad7"
  deadline: "1632815158732"
  namespace: "guest"
  transaction_id:
  "R8AyVi3UyXDVqDtyMxawepE9xOPgnY9t"
  value: object
    randomID: 0.9007484583851308
g.payload : Object
object
  action_name: "/guest/dockeraction"
  action_version: "0.0.11"
  activation_id:
  "9d2180a2e57c44c8a180a2e57c34c8cb"
  deadline: "1632815165432"
  namespace: "guest"
  transaction_id:
  "preWxR4ldnybFulVDQzRzr1dQ7SfDHPj"
  value: object
    randomID: 0.9007484583851308
```

Figure 59: Example of Context Reuse issue in warm containers

Opportunistic Context Reuse

The default way presented above, that includes the observable context from a subsequent execution of a new function invocation on a paused container, could be the default way in which functions can be written. In this manner the developer would need to be careful regarding needed initializations of parameters and variables to happen in the flow logic, however they could also benefit from finding existing context from previous executions from a performance point of view. For example, if a function needs a token in order to access an external storage, the call to obtain the key would need to be performed in each function invocation. However if the function finds the key in the context from the previous execution this could speed up its execution, if no security issues arise. For example, if a developer creates a serverless API that uses some input argument to retrieve a different token per user, while all users invoke the same endpoint, this pattern would not be suitable. However if the invocations are expected to be performed by the same user, then the token could be reused. This pattern is noted as opportunistic since it is not guaranteed that in all cases the functions will find a paused container and thus context from the previous executions.

Deterministic Context Purge (OW Skeleton Tabula Rasa pattern)

In case the developers consider that finding context from previous executions can be dangerous or harmful, this variation will guarantee that any pre-existing context will be purged. In this case, the context preservation is shifted to the local filesystem option of Node-RED, given the fact that it can be easily deleted within the function flow. For this reason, caching of the context in memory is also set to false, while flush interval is set to 0.05 seconds on change. A counter is set during the /INIT method to indicate whether this is the first execution. If not, in the use of the paused container the /INIT method is not re-executed by OW. Hence the main /POST flow includes a switch based on the value of the counter. If it is not zero, then the flow is redirected to delete the local file that includes the context. To support this manner of operation, a new pattern has been created, the OW Skeleton Tabula Rasa, that appears in Figure 60.

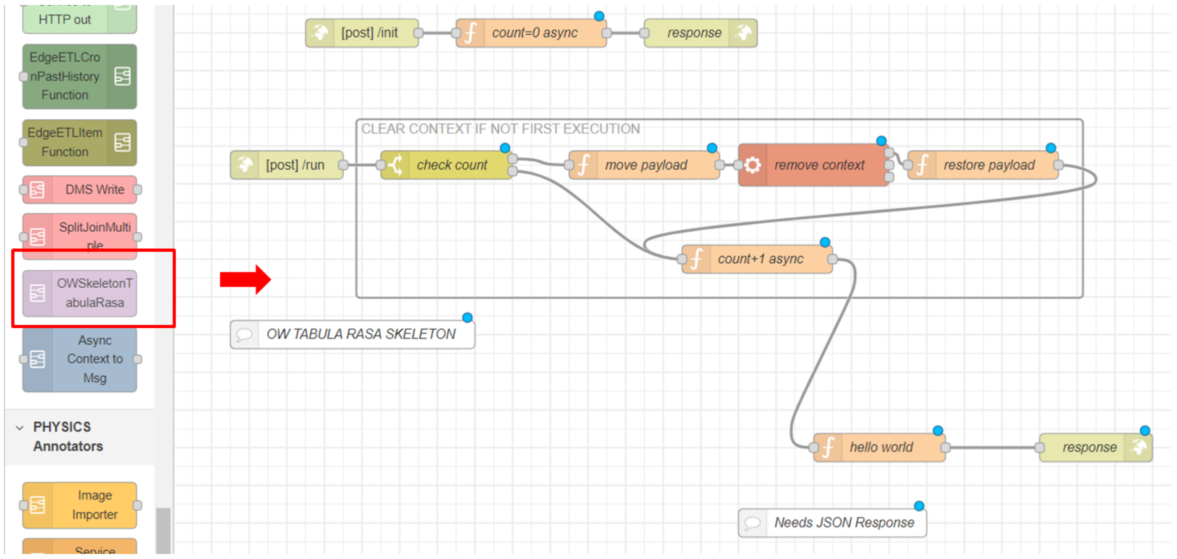
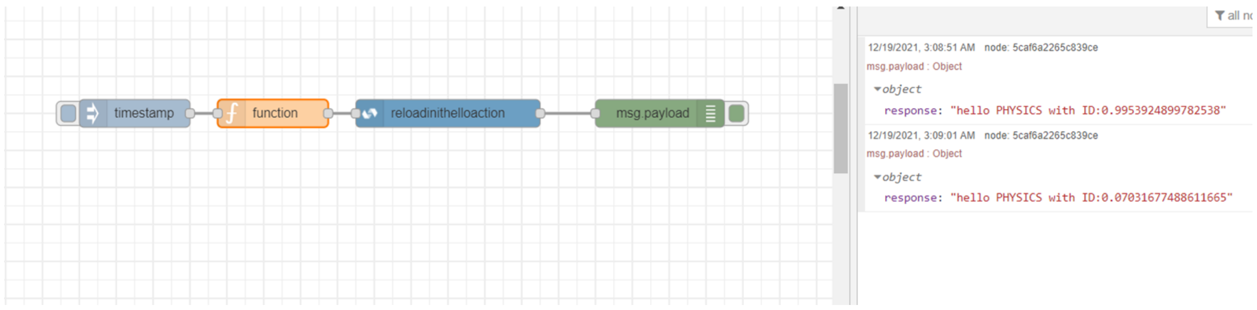


Figure 60: Deterministic Context Purge (OW Tabula Rasa pattern) Skeleton flow

In this case, repeating the previous experiment with the random IDs appears in Figure 61, indicating a successful context purging and a new random ID value.



```
12/19/2021, 3:08:51 AM   node: 5caf6a2265c839ce
msg.payload : Object
▾object
  response: "hello PHYSICS with ID:0.9953924899782538"
12/19/2021, 3:09:01 AM   node: 5caf6a2265c839ce
msg.payload : Object
▾object
  response: "hello PHYSICS with ID:0.07031677488611665"
```

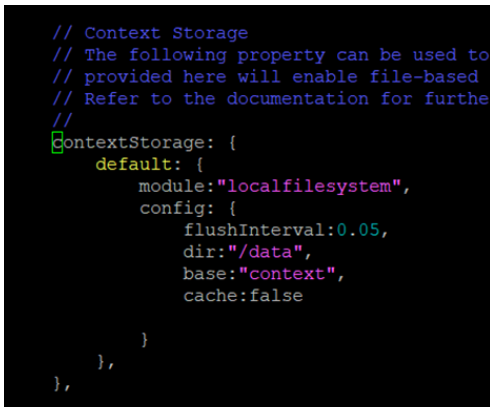
Figure 61: Successful context purging in an OW Tabula Rasa Action

What also needs to change in this case is the way Node-RED handles context. The settings needed in the respective settings.js file appear in Figure 62. Initial performance measurements (included in T4.2) do not indicate a significant performance drawback of this approach, however one disadvantage is that now reads and writes to the flow or global context need to be wrapped around asynchronous functions (example appears in Figure 63) which is a significant increase of complexity compared to the straightforward get/set synchronous methods used in the initial OW skeleton.

In order to abstract more the way developers use the flow and global variables, we have provided a get/set subflow that can be configured via the incoming message fields values in order to perform this operation. The node is configured via the following message attributes:

- msg.get_set= "get" or "set" for which method to choose

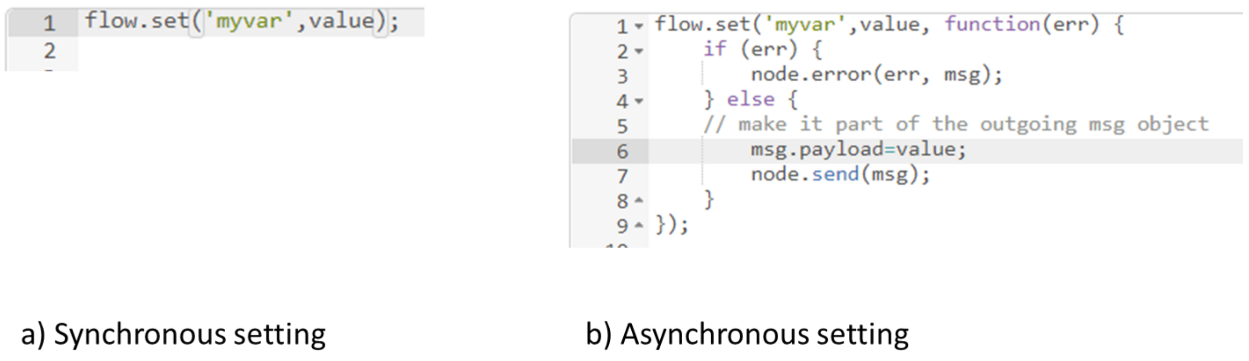
- msg.contextLevel="flow" or "global" for which context level to choose
- msg.variable="<variable_name>" for which variable name to use
- msg.payload for the value to be set



```
// Context Storage
// The following property can be used to
// provided here will enable file-based
// Refer to the documentation for furthe
//
contextStorage: {
    default: {
        module:"localfilesystem",
        config: {
            flushInterval:0.05,
            dir:"/data",
            base:"context",
            cache:false

        }
    },
},
```

Figure 62: Updated settings of Node-RED for supporting the Tabula Rasa pattern variation



```
flow.set('myvar',value);
```

```
flow.set('myvar',value, function(err) {
    if (err) {
        node.error(err, msg);
    } else {
    // make it part of the outgoing msg object
        msg.payload=value;
        node.send(msg);
    }
});
```

a) Synchronous setting

b) Asynchronous setting

Figure 63: Change from synchronous to asynchronous variable settings in OW Tabula Rasa

In case of getting, the variable value is sent in the outgoing msg.payload. An example of usage of the node appears inside the flow of Figure 64. The code of the example (asynchelloaction) is available at the project repository while the docker container image is available on Dockerhub[20].

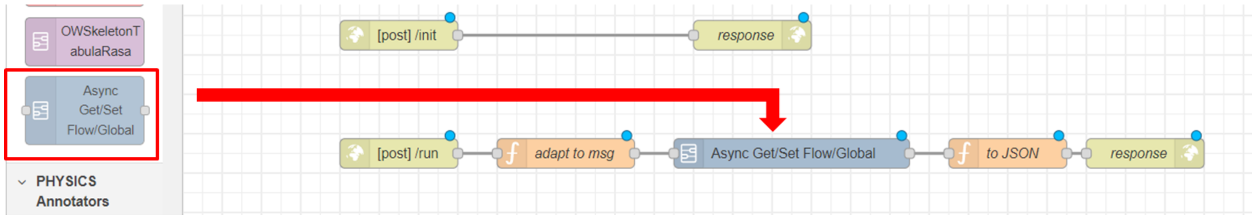


Figure 64: Example Usage of the Async Get/Set subflow

[20] Async Hello Example Image, available at: https://hub.docker.com/r/gkousiou/asynchelloaction

Deterministic Context Reuse (Externalized State pattern)

In the case of Opportunistic Context Reuse, if the next invocation comes within the paused container duration they will find the existing context. But if they arrive after that, a new container will be created. Given that in some cases one may be interested in reusing context, a third pattern can exist in order to guarantee that in all circumstances. For example, this enables the implementation of other patterns such as the function chain, i.e. the ability to execute functions for longer time than the maximum allowed by the FaaS platform. In order to achieve that, one should include in the function a timer to measure the remaining time, and when this timer is near the max function execution time, it could store the state to the external store and trigger a next function of the same type which would reload the state and resume. Alternatively, this could be done in the stopping section of a function upon an external timeout event. In order to separate between different function executions this could be used with a key based on the activation id of a function. This id would feed as an input argument to the second, extended execution of the same function instance.

Node-RED can be configured to store the context in an external store. Custom plugins can also be created in order to use whatever external storage service to store the state. This gives the opportunity to define a way to always use that state, loading it from the external store on initialization and saving it on finalization. This pattern can exploit plugins that use the DMS of T4.4 in order to store the state, or any other alternative external storage services.

Node-RED orchestration as a function

As mentioned in Section 2, orchestrating an application through Node-RED may provide significant advantages, in terms of abilities to define arbitrary workflow structures. Typical FaaS platforms provide very limited ways of describing large and complex workflows, either without support for workflow primitives or with complex text/yaml syntax that can not scale to large numbers of functions. Through this variation, one can exploit Node-RED's arbitrary wiring and runtime support for message manipulation in order to construct arbitrary workflow orchestrators, along with the control structures provided as patterns in PHYSICS (e.g. SplitJoin or BranchJoin patterns). In this case, the inner actions invoked in the workflow should already be registered with the platform or at least included in the overall application graph. A prerequisite in this case is the Openwhisk node client of Node-RED, for invoking the inner actions. The user should also use the Semantic Executor mode in order to define that this flow will be executed as a service. A generic example of such a flow appears in Figure 65, while more detailed and complex examples are included in the SplitJoin Pattern section that utilizes this pattern. The blue node is the Node-RED Openwhisk interface node that invokes a given function (or action in the OW terminology).

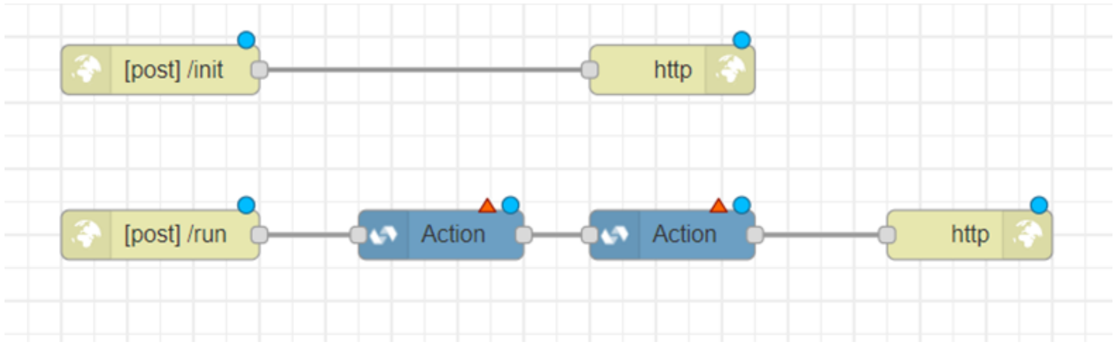


Figure 65: Example Orchestration flow as function

4.3.6. Pattern experimentation outcomes

The performance measurements for the Node-RED runtime as well as differences in orchestration modes have been benchmarked in the context of T4.2 and are included in D4.1.

4.3.7. Pattern publication means

The OW skeleton pattern and its variation is included in the Node-RED palette as a subflow. The nodes and example flows and images are also available in the project Node-RED flows repository[21] as well as dockerhub registry.

4.4. Split and Join Pattern

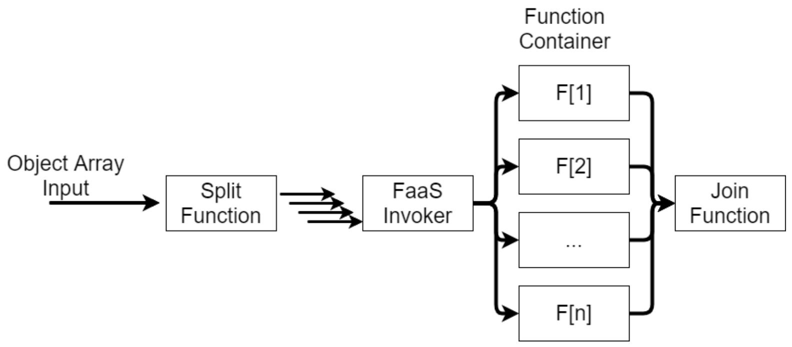
4.4.1. Pattern template description

The template description of the Split Join pattern appears in Table 12.

Table 12: Template Description for Split Join pattern

Pattern name	Split And Join (ForkJoin)
Relation to Requirements	Performance, Scalability, Req-3.3-ParallelContainerExecution, Req-4.2-FaaSBenchmarking (to check effectiveness of approach)
Source of pattern	Parallel computations from Smart Agriculture and Smart Health UCs
Pattern Context	Problems that can be easily parallelized based on different data inputs that need to be executed on the same program (Single Program Multiple Data-SPMD)

[21] PHYSICS Node-RED flows repository: https://flows.nodered.org/collection/HXSkA2JJLcGA

Pattern Underlying Problem	A problem size might be large, leading to significant execution times for centralized execution. Parallelization of execution would lead to performance benefits.
Pattern Usefulness/ Objective	The pattern aims to exploit the inherent parallelization of the serverless paradigm, following the ability to launch separate containers based on incoming requests. To this end, it takes an array of inputs and splits it into different messages, each of which triggers the invocation of a function. The Join function waits until all executions are finished and joins the results in an output array.
Schema	 Function Container F[1] F[2] ... F[n] Object Array Input Split Function FaaS Invoker Join Function
Prerequisites	Array Input of whatever type (e.g. objects, sources of data etc.)
Condition of application	N/A
Input parameters	Name of the inner parallel function to invoke, array of inputs to the inner function
Output parameters	Merged response object from the inputs
Included functionality	The split is performed on each element of the top level array of the JSON object. Therefore if one needs to apply another tradeoff (e.g. include more than one data inputs for one function), a multilevel array will need to be created before the application of the pattern flow. Furthermore given the burst of requests for this specific function runtime, context reuse should be expected between function invocations. This can prove useful in terms of performance, but should be taken under consideration when creating the main function.
Pattern limitations	The maximum benefit applies for the maximum concurrency factor of the function in the FaaS platform. If the number of invocations caused by the split exceeds that limit, then the pending requests will wait until a relevant function container is available. Furthermore, contention due to multiple container executions on the same node should be measured.
Linked to Pattern	Node-RED flow as function pattern (including orchestration variation)

Indicative Domains of Applicability	Parallel computations, Single Program Multiple Data (SPMD) pattern with output barrier (Fork-Join)

4.4.2. Pattern implementation details

The created split (fork)-join pattern to parallelize parallel computations based on the Single Program Multiple Data (SPMD) pattern (Figure 66a) appears in Figure 66b. These computations are in many cases based on typical parallel technologies like MPI, which however are more difficult to develop and support.

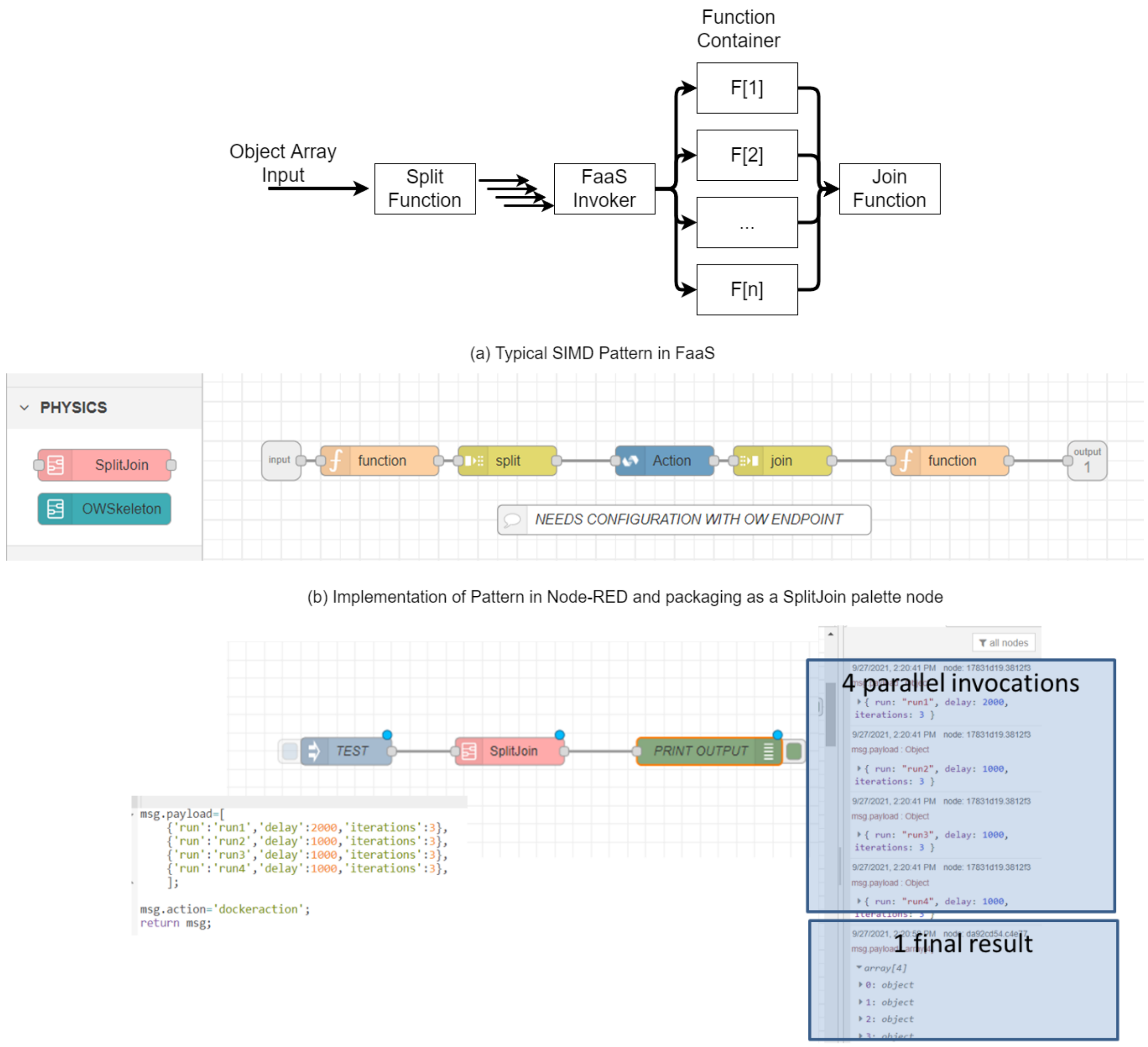
In a function programming style, the parallelization may be performed by splitting the initial message that contains an array of rows upon which the same computation will be applied on each row. Each split message then triggers a respective function execution on a FaaS platform, while a Join node waits for all the respective partial messages created from the original message. Contrary to the MPI case, we do not need to have declared in advance available and static MPI nodes to be used for the parallelization. The invocations are forwarded to the FaaS platform and are executed based on the typical FaaS execution model. Grouping at the output is performed based on the unique original message id that characterizes all partial messages as well as a msg.parts field that indicates the position in the initial sequence of each partial message.

The Split and Join nodes are built-in Node-RED nodes. The Action node is a Node-RED packaged node available on npm to interface with an existing Openwhisk platform[22], while the "function" nodes include custom code used to adapt incoming and outgoing message fields (e.g. enforce a convention that the array to be split is included in the msg.payload field). The overall functionality is included in a subflow (SplitJoin node in left-side palette), and can be used directly in an example flow (Figure 66c) with minimum effort, indicating only the Openwhisk endpoint to be used. It can thus be included in any Node-RED server flow.

4.4.3. Pattern examples of usage through a function orchestrator

In order to fully exploit the functionality of the environment, the flow can be executed in a function mode, thus enabling higher scalability of the orchestration mechanism, exploiting the Node-RED Orchestration variation of the Node-RED flow as function pattern presented in this document. In order to enable this, the subflow needs to be wrapped around the Openwhisk Skeleton flow. Arguments are passed by the Openwhisk runtime as the body of the POST /run method.

[22] Node-red Openwhisk Interface Node Description. Available at: https://flows.nodered.org/node/node-red-node-openwhisk.



(a) Typical SIMD Pattern in FaaS

(b) Implementation of Pattern in Node-RED and packaging as a SplitJoin palette node

```
msg.payload=[
    {'run':'run1','delay':2000,'iterations':3},
    {'run':'run2','delay':1000,'iterations':3},
    {'run':'run3','delay':1000,'iterations':3},
    {'run':'run4','delay':1000,'iterations':3},
    ];

msg.action='dockeraction';
return msg;
```

```
9/27/2021, 2:20:41 PM   node: 17831d19.3812f3
msg.payload : Object
 ▸{ run: "run1", delay: 2000,
iterations: 3 }
9/27/2021, 2:20:41 PM   node: 17831d19.3812f3
msg.payload : Object
 ▸{ run: "run2", delay: 1000,
iterations: 3 }
9/27/2021, 2:20:41 PM   node: 17831d19.3812f3
msg.payload : Object
 ▸{ run: "run3", delay: 1000,
iterations: 3 }
9/27/2021, 2:20:41 PM   node: 17831d19.3812f3
msg.payload : Object
 ▸{ run: "run4", delay: 1000,
iterations: 3 }
9/27/2021, 2:20:59 PM   node: da92cd54.c4e77
msg.payload : array[4]
 ▾array[4]
  ▸0: object
  ▸1: object
  ▸2: object
  ▸3: object
```

(c) Abstract Usage of SplitJoin subflow in other flows

Figure 66: Split Join Pattern Concept, Implementation and Example

The needed arguments are two, initially the name of the inner Openwhisk action to invoke (the function responsible for processing each input chunk) and the initial array of input data that are split and sent for processing in chunks. The resulting flow appears in Figure 67.

The latter also includes a couple of testing flows that can be used while the developer implements and tests the flow inside the Node-RED editor environment. These can prove very useful for debugging and fixing a number of Javascript or data passing related errors without having to deploy to the actual Openwhisk environment. Indicatively, during the tests for the creation of this flow, approximately 70-80% of the errors were fixed without the need to deploy to the final FaaS platform.

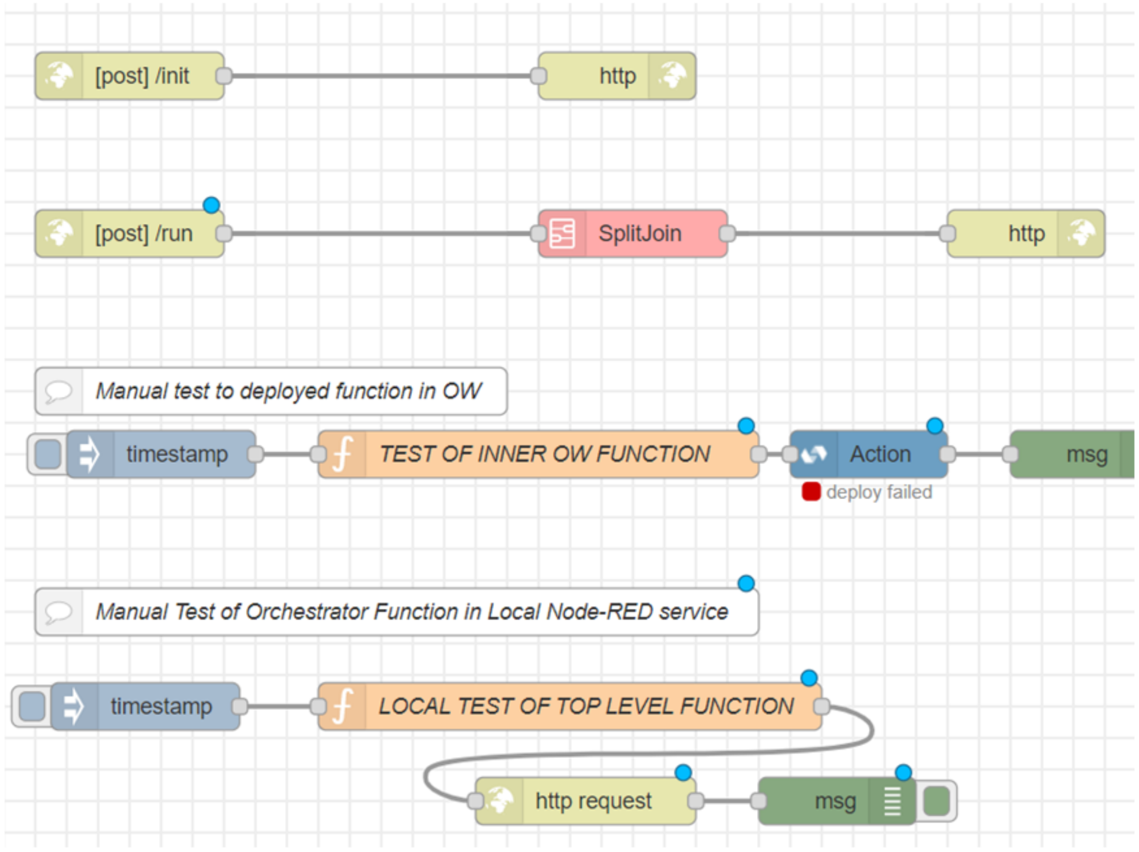

[post] /init
http
[post] /run
SplitJoin
http
Manual test to deployed function in OW
timestamp
TEST OF INNER OW FUNCTION
Action
deploy failed
msg
Manual Test of Orchestrator Function in Local Node-RED service
timestamp
LOCAL TEST OF TOP LEVEL FUNCTION
http request
msg

Figure 67: Example Orchestrator Flow for the Split Join Usage

4.4.4. Pattern necessary adaptations

In order to use this pattern, initially the application needs to have adapted the inner computation to be performed in a containerized manner, as well as a function manner. Therefore, they need to have registered the inner function in the Openwhisk environment, in whatever manner (e.g. native openwhisk function, or, if embedded in a Node-RED flow, the Node-RED flow as function pattern. For having a more flexible execution, and for supporting variations of multiple split sizes (see next section for SplitJoin variations), this inner function should be able to handle arrays of inputs.

4.4.5. Pattern limitations

Given the creation of one function container for each request, one question when applying this pattern relates to the stress caused in the backend. While parallelization would enable the faster execution of the application, the existence of vast container numbers in the background would mean that contention arises during execution, especially in resource constrained environments such as edge locations or small clusters. Furthermore, raising a container in that manner in order to do a rather small computation (e.g. in the Smart Agriculture UC each individual simulation takes approximately 1-2 seconds) would result in having to tolerate overheads for container creation or reuse.

Another limiting factor is the concurrent actions limitation by the Openwhisk environment, which is a limit to the number of the same function instances that may execute concurrently in the cluster. If that number is reached, then the extra requests start to fail. Therefore, a careful selection of the split size should be performed, which in turn would regulate how many requests would be generated towards the backend. This led to the creation of a relevant pattern variation (SplitJoinMulti) in order for the developer (or the environment) to regulate the split size, presented in the following section.

4.4.6. Pattern variations

As mentioned above, the ability to define an arbitrary split size of the input array could prove critical in order to ensure a smooth and flexible execution. For this purpose, the SplitJoinMultiple variation was created that appears in Figure 68.

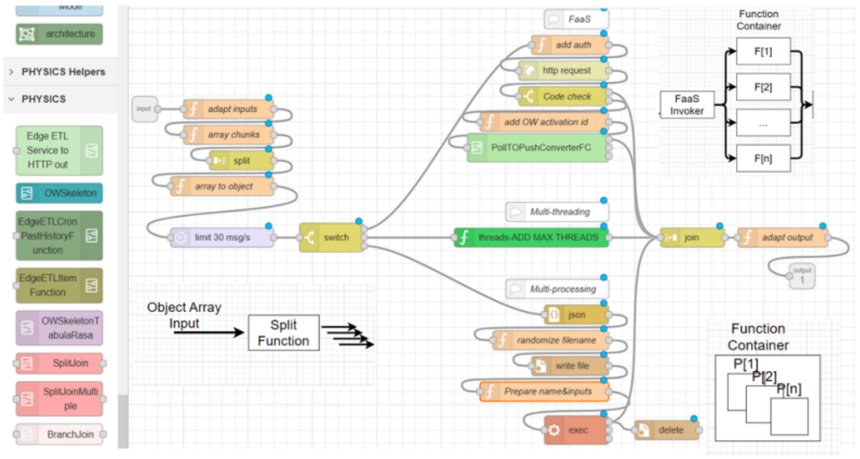
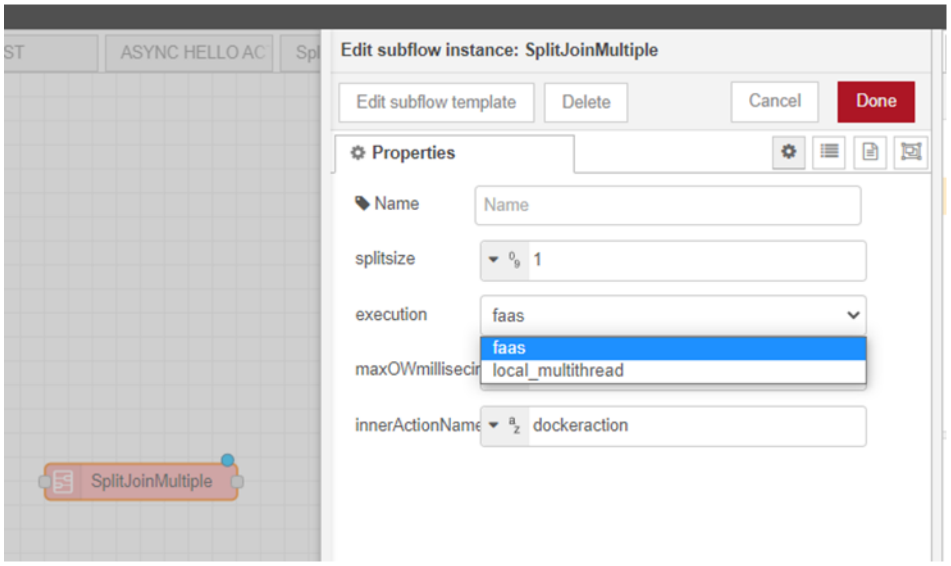


Figure 68: SplitJoinMultiple Variation for dynamic regulation of split size

In contrast to the simple SJ pattern that breaks it into single messages, this pattern gets the initial message and chunks it down based on the *msg.payload.value.splitsize* value. Therefore if an array of 1000 rows (included in the *msg.payload.value.values*) is inserted and the splitsize is set to 10, it will create 100 inner calls of the *msg.payload.value.action*, each of which will undertake 10 inputs. The solution of the *msg.payload.value.splitsize* was chosen since node properties of the built-in Node-RED Split node allow only environment variables to be used for setting the split size. These can be configured only at startup and thus can not be used for dynamic management of the pattern during runtime. Furthermore, with this dynamic setting we can also pass the split size as an argument when we incorporate this subflow around an HTTP endpoint and the split size can be an input argument. However it needs to be noted that now the functions used in the inner parallelized action need to be able to process arrays as inputs.

The flow includes also a rate limiter, in order to be aligned with any Openwhisk options regarding maximum invocations per minute. By setting the rate limiter to the according OW limit, we can avoid failures of action invocations due to this limit. The parameter can be set in the millisecond interarrival time (*msg.paylod.value.maxOWmillisecinterarrival*). Furthermore, the flow supports a local multithreaded execution, if the respective function needs to be executed within the top level function without spawning inner level external functions. In this case, the incoming value of *msg.payload.value.execution* needs to be set to "local_multithread". If it is set to "faas'' then the Openwhisk Action that is contained in the *msg.payload.value.action* is used.

The data format of the *msg.payload* is specifically designed so that it is compatible with the way Openwhisk passes arguments so that this flow can be directly combined with the OW Skeleton node and executed as an orchestrator function. The configuration parameters can also be set through the node menu (Figure 69), however the values in the incoming message will override any set values during node initialization.



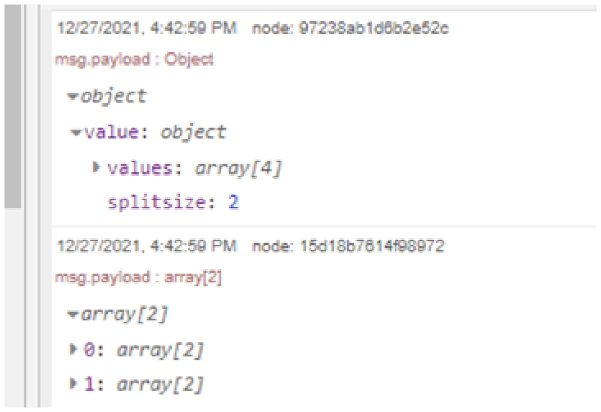
Edit subflow instance: SplitJoinMultiple

Name: Name
splitsize: 1
execution: faas
maxOWmillisecir
innerActionName: dockeraction

Figure 69: SplitJoinMultiple Node and Parameters

4.4.7. Pattern experimentation outcomes

One of the relevant questions for the operation of the SplitJoin pattern (especially the Multiple Variation) is how much time it takes for the rearrangement of inputs in order to be split into lower level arrays. For the dynamic split size to take effect, the initial N rows array needs to be broken down into N/S arrays (where S is the split size) and pushed down the hierarchy of the message object. An example of splitting a 4 element array with a split size of two appears in Figure 70, depicting the message object just before the entry to the SplitJoinMultiple node and the message just after the "array chunks" function of the pattern. Therefore, one should try to experiment with a number of different array input sizes and relevant split sizes in order to detect significant delays or memory limits when applying this pattern.



```
12/27/2021, 4:42:59 PM   node: 97238ab1d6b2e52c
msg.payload : Object
▾object
 ▾value: object
   ▸values: array[4]
    splitsize: 2
12/27/2021, 4:42:59 PM   node: 15d18b7614f98972
msg.payload : array[2]
▾array[2]
 ▸0: array[2]
 ▸1: array[2]
```

Figure 70: Example of Needed Input Resizing in SplitJoinMultiple

To this end, the following testing flow was created (Figure 71), for testing out the time delays needed inside the pattern when having a number of different setups. In order to measure only the specific time segments in the flow, the calls to the OW node were bypassed, since we are interested primarily in the delay due to the inner pattern management activities. Furthermore, the rate limiter was bypassed, since this is again a limitation of the OW setup. So this measurement includes the delay for restructuring the array as well as delays for splitting, propagating and recollecting the messages. The flow was executed within the Node-RED environment running as a server.

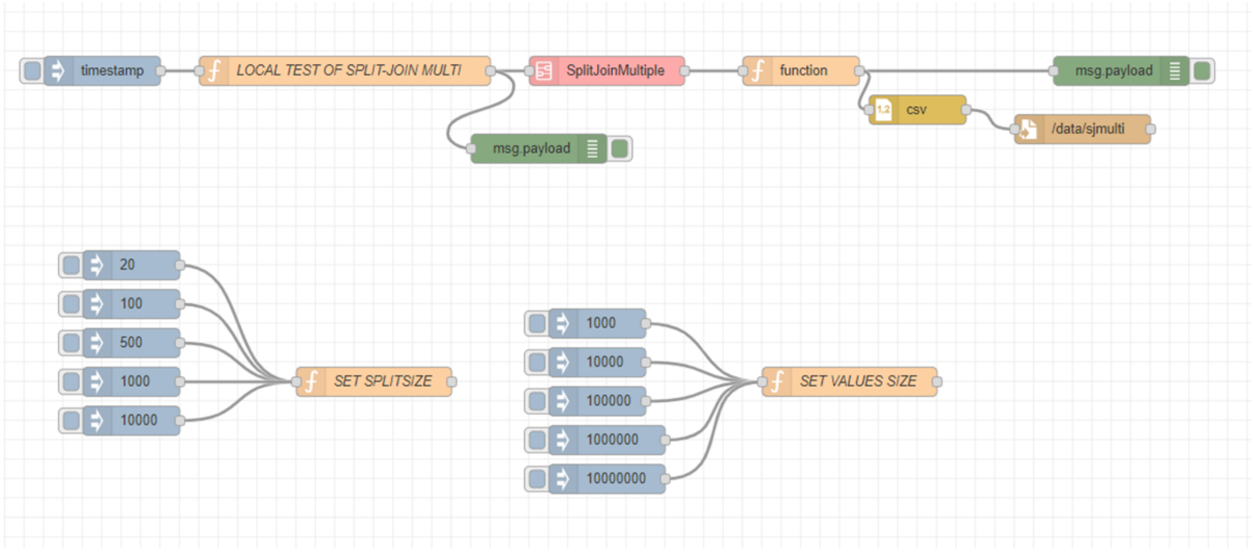


Figure 71: Testing flow for experimentation data of SplitJoinMultiple message management overheads

Different split sizes were used (20,100,500,1000,10000) for different sizes of inputs (1000, 10000, 100000,1000000,10000000). While for small sizes the delay is negligible (in the range of 20 milliseconds), for cases of large arrays (e.g. 1,000,000 inputs) (Figure 72) the split size significantly affects the speed, primarily due to the fact that a large number of messages needs to be created and propagated within the environment for small steps, increasing delays from 8.6 seconds (when split size is 10,000) to 102,888 seconds (when split size is 20).

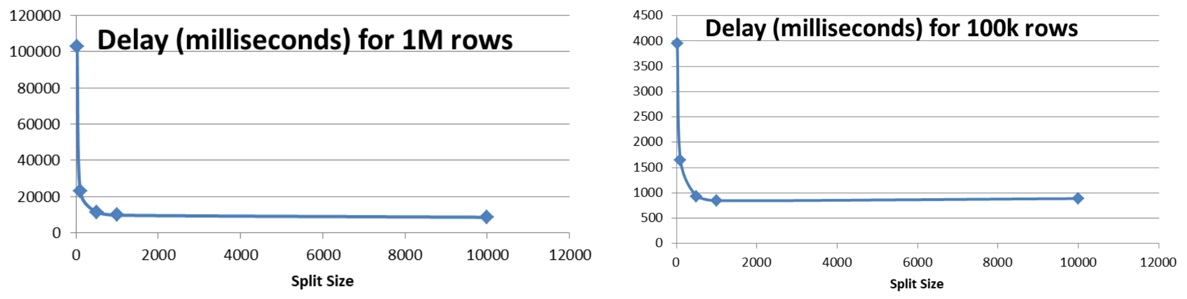


Figure 72: Indicative delays for SplitJoinMultiple message manipulation for different split sizes

However also the size of the messages affects the delay, although to a smaller extent, as Figure 73 indicates. In each plot, different input size variations that create the same number of inner messages (as derived from the input size/split size ratio) indicate larger delays as the total input size increases. Given that the number of messages is constant in this case, what changes is the size of each message, containing more rows.

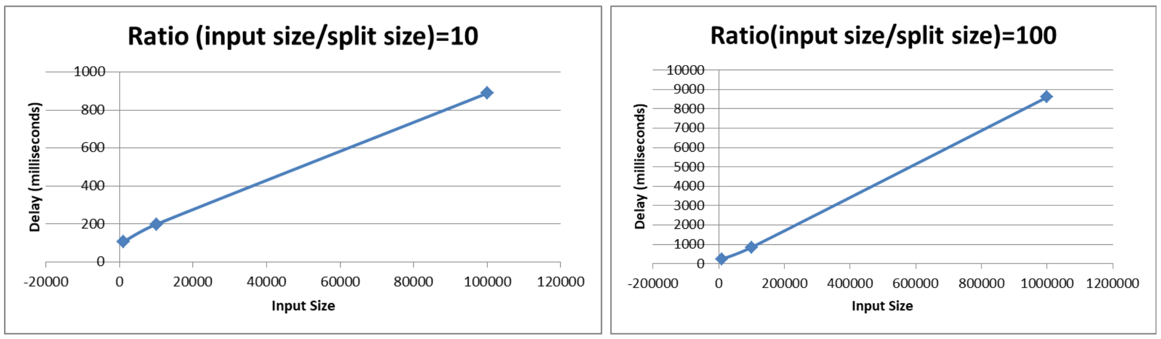


Figure 73: Indicative delays for larger messages with the same input to split ratios

When adding the flow in a function wrapper, according to the Node-RED orchestrator pattern, we anticipate a larger delay due to the intermediate layer. Also we would like to check the limits of Openwhisk, which relates to the size of the input message. Indicatively, by using the default values of the input message, we get the following error from as early as 100k rows:

"POST http://10.100.59.182:3233/api/v1/namespaces/guest/actions/splitjoinmultiplegrounded?blocking=true Returned HTTP 413 (Payload Too Large) --> "Request larger than allowed: 4300027 > 1048576 bytes.""

indicating that based on the specific row size, we can have a maximum of 25K rows approximately as top level input. This is evidence that a bypass should be followed in order to enable larger input sizes.

In terms of timing delays, it can be observed in Figure 74 that there is a significant increase in the case of the function execution (again referring only to pre and post processing delays of message handling). The function executions were warm ones. However, even though the difference seems large, the benefits of a function execution (e.g. no dependency from the Node-RED throughput limit of a single server) favor for this mode of operation, especially given the fact that the delay difference in the orchestration would be small compared to the overall time of the application.

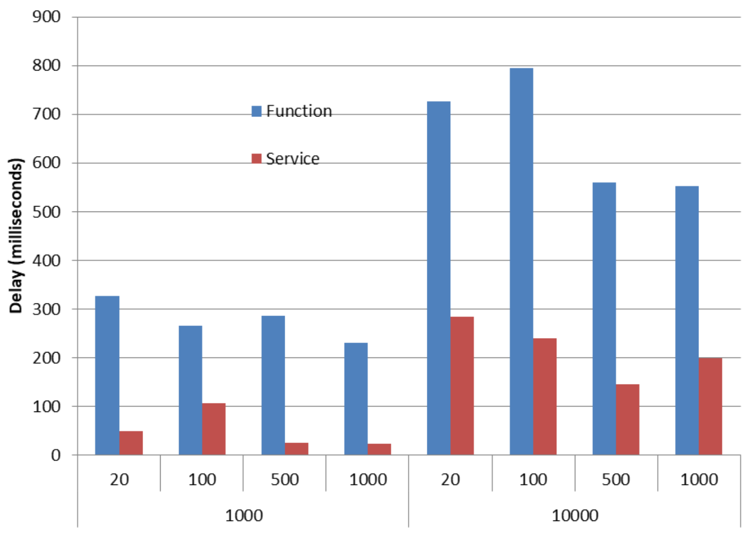


Figure 74: Performance Difference between Function and Service mode for the SJ pattern

Experimentation on the number of containers used (i.e. split size used) with relation to performance of the function execution and total application time has been performed and is included in D4.2.

4.4.8. Pattern publication means

All relevant artefacts have been created and included in the project repository. These include:

- testing flows
- SJ and SJmultiple flows, included in the PHYSICS palette as well as standalone flows[23]

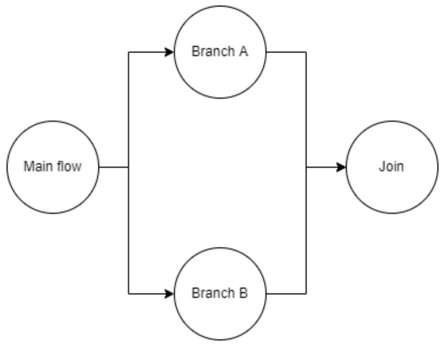
[23] https://flows.nodered.org/flow/7a5acfc999b1ad47bb32b5d37419c777/in/HXSkA2JJLcGA

4.5. BranchJoin Pattern

4.5.1. Pattern template description

The template description for the BranchJoin pattern appears in Table 13.

Table 13: Template Description for the BranchJoin Pattern

Pattern name	BranchJoin
Relation to Requirements	Req-3.1-WorkflowDef
Source of pattern	Generic Workflow construction needs
Pattern Context	The applications need to implement specific workflow constructs that can aid them in capturing the business logic. Specifically, at some point in the workflow, they may need to merge messages coming from two branches in one single message.
Pattern Underlying Problem	Typically in a combined use of a split join node, the split node takes care of annotating the message with information used by the join (such as *msg.parts*, index in the message sequence, id based on which to filter various messages coming in the join node), in order to properly reassemble the initial message in the join node. However when one needs to just merge the outputs from two partial branches, without the need to use a split in the beginning, the info needed to correctly reassemble the message does not exist.
Pattern Usefulness/ Objective	The pattern automates and abstracts the process of structuring annotations in messages coming from different branches in order to support the developer to create easily such joins in the flow logic.
Schema	
Prerequisites	The developer uses one such node in each branch as well as a default Node-RED join node
Condition of application	Need to merge messages coming from different branches
Input parameters	Message from branch A and message from branch B

Output parameters	One combined message at the output (as array of payloads)
Included functionality	Adds msg.parts logic to support the Join node
Pattern limitations	No cloning should be performed in the messages in the branches, same initializing point of the two branches
Linked to Pattern	Node-RED flow as function
Indicative Domains of Applicability	Generic Workflow primitives needs

4.5.2. Pattern implementation details

The default Join node of Node-RED can distinct branch messages coming from different initial messages, however a relevant annotation needs to exist (Figure 75). This annotation, typically included by a split node in a combined split/join usage, is necessary for the join node to reassemble the messages. It includes an id, assigned during the split of this particular message, the number of elements of the original message as well as the index (location) of this partial message with regard to the original position in the message. With this information, the join node waits until all the partial messages of the same id are assembled and then recompiles and sends the compiled overall message. If multiple splits are performed, each one pushes down the stack of the parts object the relevant information. Each Join node then pops the last inserted item and performs the join based on this information.

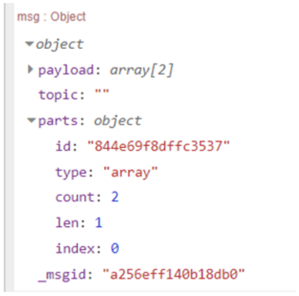

```
msg : Object
▾object
 ▸payload: array[2]
  topic: ""
 ▾parts: object
    id: "844e69f8dffc3537"
    type: "array"
    count: 2
    len: 1
    index: 0
  _msgid: "a256eff140b18db0"
```

Figure 75: Example of msg.parts structure normally used in the join node reassembly

However, when the initial split node is not needed, alternative ways can be used such as using a counter of messages in the Join node for the reassembly of the message. The problem then appears in Figure 76.

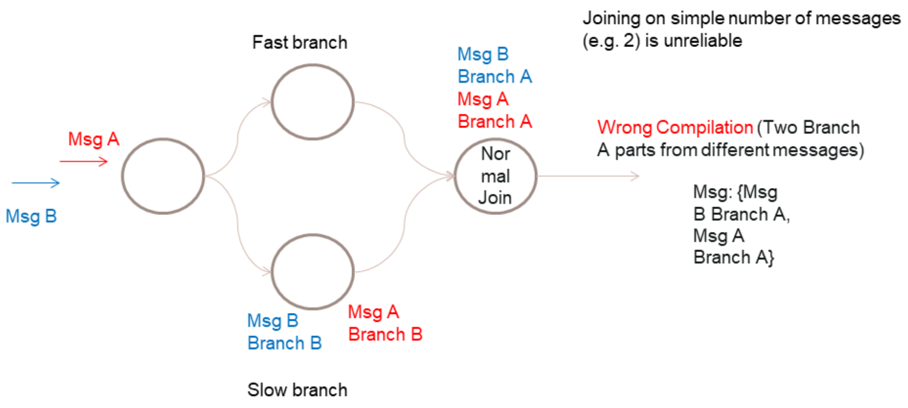

Fast branch
Joining on simple number of messages (e.g. 2) is unreliable
Msg B Branch A
Msg A Branch A
Wrong Compilation (Two Branch A parts from different messages)
Msg A
Msg B
Nor mal Join
Msg: {Msg B Branch A, Msg A Branch A}
Msg B Branch B
Msg A Branch B
Slow branch

Figure 76: Erroneous message reassembly based on simple message count

The timing of message arrival as well as delay in processing in the branch will lead eventually to errors in the reassembly of the message. The intermediate BranchJoin node can be used in order to insert the aforementioned level of needed annotation (Figure 77). Given that a unique message id exists for each incoming message in the Node-RED environment, this id is used in the inserted annotation to correctly assemble the structure. Other points such as the total needed branches and the position of this branch in the final output are defined by the user and inserted based on the needed format. The implementation pushes the annotation down the msg.parts object, if that exists, in order to respect any previous setting of this information by other nodes.

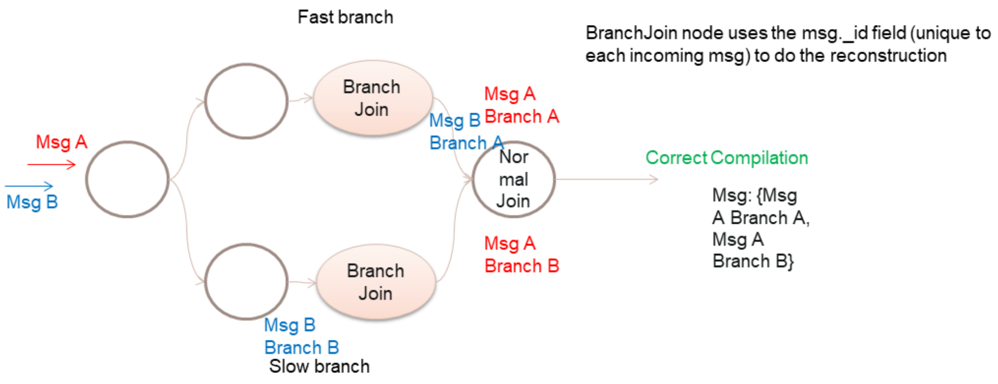
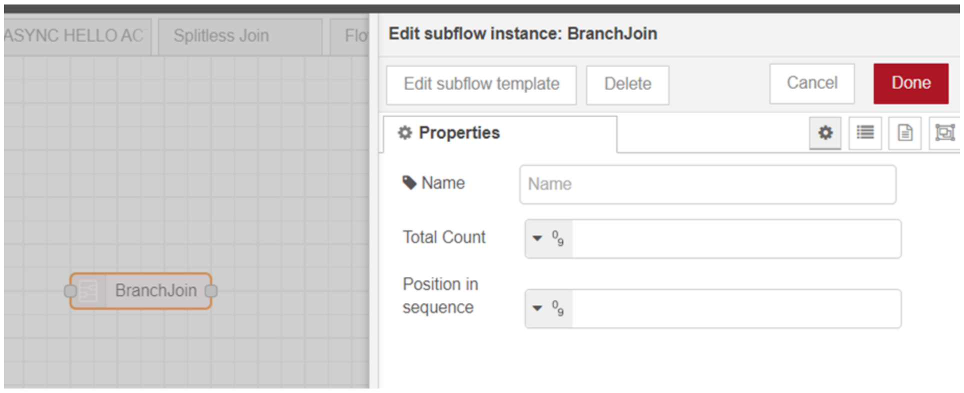


Figure 77: Correct message reassembly based on the BranchJoin Node

4.5.3. Pattern examples of usage

The BranchJoin node UI appears in Figure 78. The user should use one such node in each branch, just before the usage of a default join from the Node-RED palette. They should also define how many branches exist (Total Count) and what is the position of this branch in the sequence (from 1 to total count).



Edit subflow instance: BranchJoin

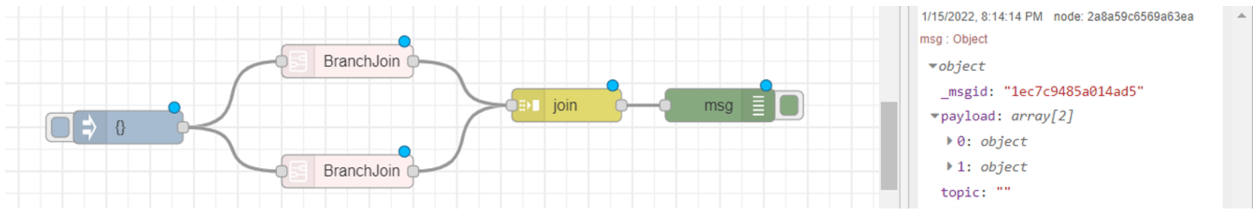
Name: Name

Total Count

Position in sequence

Figure 78: Parameter Setting in the BranchJoin Node

An example usage in a flow with two branches appears in Figure 79. The Join node should be used in the default automatic mode. The debug output indicates the merged message object in the output.



```
1/15/2022, 8:14:14 PM   node: 2a8a59c6569a63ea
msg : Object
▾object
  _msgid: "1ec7c9485a014ad5"
 ▾payload: array[2]
   ▸0: object
   ▸1: object
  topic: ""
```

Figure 79: Example Usage of the BranchJoin in two branches

The node also respects any existing msg.parts attributed in the message, so that it may not disrupt any previous setting of this property (if other split-join combinations of nodes are included in the flow).

4.5.4. Pattern necessary adaptations

One point of attention is that both branches should send a message, otherwise the accumulated message will never be completed. Thus if a branch may include cases that do not typically send one for some reason, a dummy one should be inserted to indicate that that branch has finished.

4.5.5. Pattern limitations

The pattern relies on the two routes maintaining the common msg._id attribute of the original message, which is the filter for joining them. Thus if two routes are completely independent, not sharing a common starting point, the messages will not be joined. Furthermore, some (limited) nodes in Node-RED may clone a message, meaning create a new one in the context of their operation and forward this to their output. In this case the msg._id changes and the Branch Join node will fail to find a match. Therefore the developers are advised to include debug nodes in the two points prior to the Branch Join node in order to detect whether this happens in their flow.

4.5.6. Pattern publication means

The pattern is available on the PHYSICS bundle of nodes, as well as a standalone flow in the Node-RED repository.

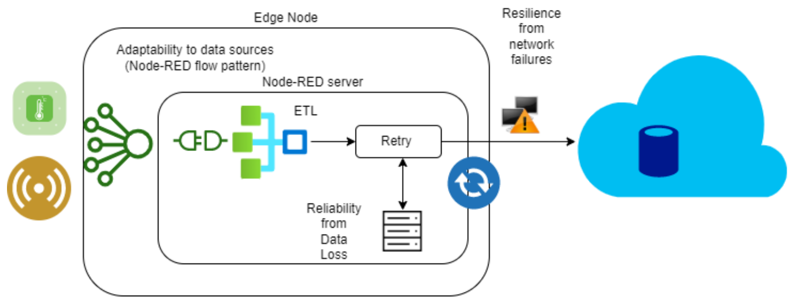
4.6. Safe and Flexible Edge ETL Pattern

4.6.1. Pattern template description

The Edge ETL Pattern template description appears in Table 14.

Table 14: Template Description for Safe and Flexible Edge ETL Pattern

Pattern name	Safe and Flexible Edge ETL Pattern
Relation to Requirements	Resilience (in terms of network failures), Reliability (in terms of data loss) , Adaptability (in terms of linked IoT systems)
Source of pattern	Smart Agriculture UC
Pattern Context	IoT and Edge environments responsible for data collection and forwarding (example of greenhouse nodes responsible for plant sensors data collection)
Pattern Underlying Problem	Especially in edge environments in which a large part of the data collection process is performed, intermittent network failures may lead to data loss. In many cases, application logic only buffers a limited number of data points, therefore if the failure lasts for a larger period of time than the buffer size, the intermediate data is lost. This creates considerable consequences in the operation of services, especially of estimation and modelling services that need accurate and complete time series of data in order to reach a safe conclusion. Therefore a pattern based approach to maintain locally these data points and retry to push them in the main storage service should be in place in order to enhance the reliability and robustness of the application since with the application of this methodology data loss is expected only for cases of internal network collapse. Furthermore, in typical IoT environments data may come in from different sensors, different protocols or different communications channels. Therefore an adaptable and abstracted manner of retrieving this information and adapting it to the needed data model for storage and further processing is needed. Potential filters and data clean-up may also be performed in this step.
Pattern Usefulness/ Objective	The goal of the pattern is initially to enable the developer to exploit Node-RED abundance of IoT related protocols, clients and systems in order to create flows that retrieve the data and prepare them for ingestion, easily and in a more decentralized

	manner. To this end it also exploits the Node-RED flow pattern. Once the data item has been finalized, it needs to be pushed towards the central storage service. However if the call fails, the pattern should store the data item locally and apply the retry pattern (or the circuit breaker one) for a limited number of retries, in order not to stress the system endlessly. The pattern periodically retrieves the locally stored data items that have not been sent and pushes them to the central service and if the call succeeds deletes them from the local store. The pattern may be applied either as a standalone Node-RED service or as a Node-RED based function. In the former case the data may be stored in the Node-RED container (e.g. through a locally deployed sqlite mini DB), in the latter an accompanying data service element (e.g. from T4.4) needs to be used for local (edge) data storage.
Schema	 Edge Node Resilience from network failures Adaptability to data sources (Node-RED flow pattern) Node-RED server ETL Retry Reliability from Data Loss
Prerequisites	Edge FaaS environment, lightweight k8s version on the edge or standalone Node-RED server, installed sensors and a mean/interface to obtain their values
Condition of application	On each trigger of a new data point Periodically to repush the local values to the main Cloud DB
Input parameters	Values from data sensors, ETL needs, IoT interface
Output parameters	Type of external system for pushing, transformed values stored in central Cloud storage
Included functionality	Retry mechanism, local store mechanism, retrieval and deletion of past values
Pattern limitations	In case the pattern is applied as a function, complementary data services are needed on the edge in order to maintain the state as well as a lightweight OW instance such as Openwhisk Light.
Linked to Pattern	Retry/Circuit Breaker Generic Pattern, Node-RED flow pattern

Indicative Domains of Applicability	Smart Agriculture, Smart Ehealth

4.6.2. Pattern implementation details

The pattern is implemented as a Node-RED flow and packaged as a subflow (Edge ETL Service). The implementation appears in Figure 80. Initially an input is provided so that the developer can plug in any kind of means to obtain the primary data value, encapsulated in the payload field of the message. Then any custom ETL logic can be applied through one or more functions and apply any needed transformation, filtering or other operation on the data. Once this is finalized, the generic part of the pattern begins. Given that any output nodes, such as the HTTP out node used in this example, may substitute the contents of the msg.payload field with the results of the call that pushes the data to the central system, it is necessary to maintain the original data for future use (in case the transmission fails). For this reason, these are moved in the "Keep contents'' function in the *msg.originalpayload* field. This function is also responsible for inserting a retry counter in the message, as well as differentiating the origin of the message (new data that have arrived or past data that have failed and have been stored locally).

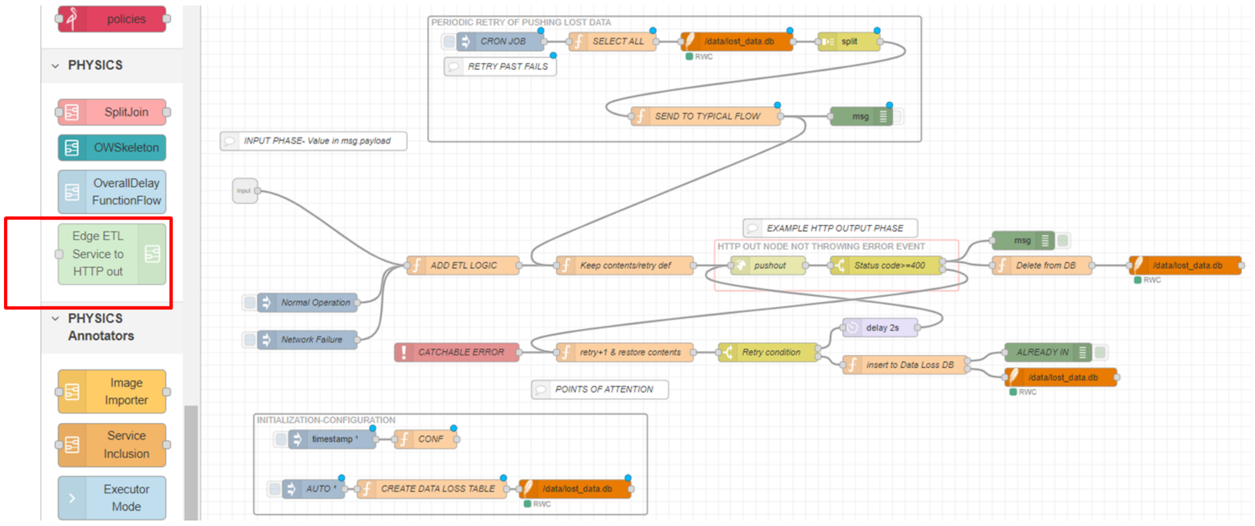


Figure 80: Implementation Flow for the Edge ETL Pattern

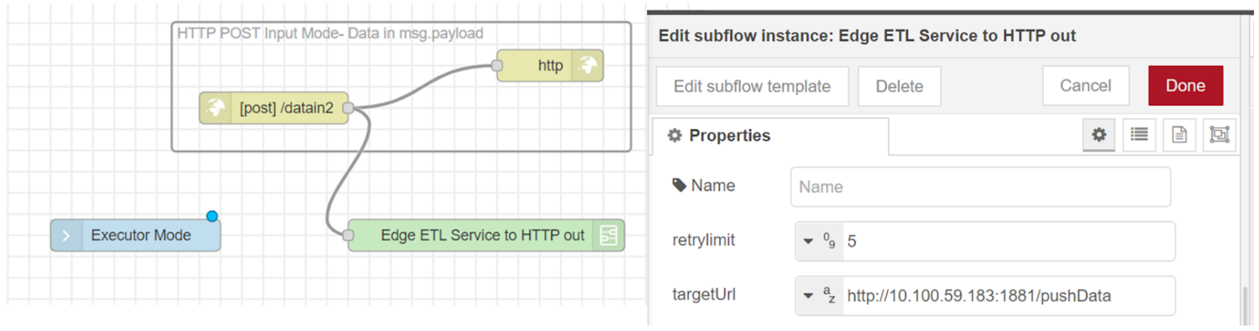
The call to the external system is performed and if this fails a configurable number of retries is attempted immediately. If all of these fail, the data item is forwarded for storing in a local database based on sqlite, embedded in the Node-RED environment. The data item is stored as is, in a string format, along with a timestamp collected from the Node-RED environment and an optional field on a used format for the string inputs. Periodically, e.g. every 30 minutes, 1 hour etc, the top box of the flow is activated in order to retry past failures. In this process it selects all data items from the DB, splits the message into one individual message per data item and retries the overall process. The split on a data item granularity is performed in order to maintain data integrity on the DB, since for each message if a success in transmission is observed a deletion from the local DB is performed to avoid duplicate values sent over the network. The pattern includes also self-initialization abilities (Initialization-Configuration Box), in order to create the DB tables if they don't exist and to configure the number of retries and target url. However the latter parameters have also been included in the subflow configuration parameters for more ease, as will be detailed in the next section.

Some points of attention include:

- The retry field in the message needs to be created as different with comparison to msg.retry, since the http node creates such a msg field in each request. Therefore the pre-existing field for retries in this flow will be overwritten by the http node (similarly to the msg.payload field).
- Some nodes, like the http request node, do not throw a nodejs error event when the call fails, but instead the error is included in the message. Therefore the CATCHABLE ERROR node is not able to detect the failed call. For these nodes, the HTTP OUT box logic can be used as an example of how to detect per case the error condition. An extra step is the direct link between the switch node and the "retry+1" node, which is not needed in a typical catch case of an output node that throws an nodejs error event on failure.
- At the moment, deletion from the DB is based on msg.payload and msg.timestamp comparison, which implies that inside the payload there should be a unique identifier. However, typically this is the case in IoT related sensing, whereas the existence of timestamps of collection can also serve as a distinguishing factor between records with similar other fields.

4.6.3. Pattern examples of usage

An example of the pattern usage appears in Figure 81. In this case we have exploited the abstraction offered by the subflow in order to create the overall flow. The input node is assumed to be an HTTP Post endpoint, in which the new data are pushed in, which is directly linked to the Edge ETL pattern node. An http response node is also needed (top right corner of the HTTP Input Box) in order to return the response to the caller. It is necessary to stress that this successful response should be interpreted not as a success to forward the data item to the central DB service but as a response for the successful receipt of the data item by the pattern. The flow shown in the previous section has been packaged as a subflow and in the node properties the main configuration parameters of the pattern can be observed, including the retry limit count and the target URL of the central HTTP OUT service. These are instantiated as environment variables at the instantiation of the node. The main pattern includes also a set of testing flows that simulate a local receipt service and generate a HTTP in call for simulating inputs.



Edit subflow instance: Edge ETL Service to HTTP out

Name: Name
retrylimit: 5
targetUrl: http://10.100.59.183:1881/pushData

Figure 81: Example of Inclusion of the Edge ETL pattern in a service mode

4.6.4. Pattern necessary adaptations

Points of adaptation/configuration in new flows include:

- the input protocol used for obtaining the data item. Node-RED has an abundance of ready made nodes that can be used to e.g. create an HTTP endpoint, listen on an MQTT or AMQP broker etc. This input layer needs to be created by the developer depending on their system and linked to the input of the pattern
- the output protocol is currently an HTTP POST call, including the data item in the payload. If another protocol is needed, the EXAMPLE HTTP OUTPUT box needs to be replaced with the according logic (similarly to the input)
- the interval of the cron job for lost data retry (configurable in the CRON JOB node set interval)
- the number of initial retries as well as the target URL of the HTTP OUT service

4.6.5. Pattern limitations

At the current time, the pattern stores the data in a local sqlite mini DB, for portability reasons and especially given that the number of data sizes is not expected to be large in a normal operation. Therefore in order to reduce dependencies from external systems the local mini DB option was applied. If one needs to decouple the local execution (e.g. if the pattern needs to be run as a function) then the target DB path should not be inside the local container environment since there is no guarantee that the same container will be reused, given that containers in FaaS case are ephemeral and created on demand. Therefore the specific setup can only be used for the service mode as is, unless the DB interface nodes target an external to the container DB.

4.6.6. Pattern experimentation outcomes

For testing the pattern, two aspects are important. Initially how the pattern scales in terms of stored messages depending on the failure scenario and finally whether the application of the pattern results in no lost data. In order to test these aspects, the following experimental setup was used:

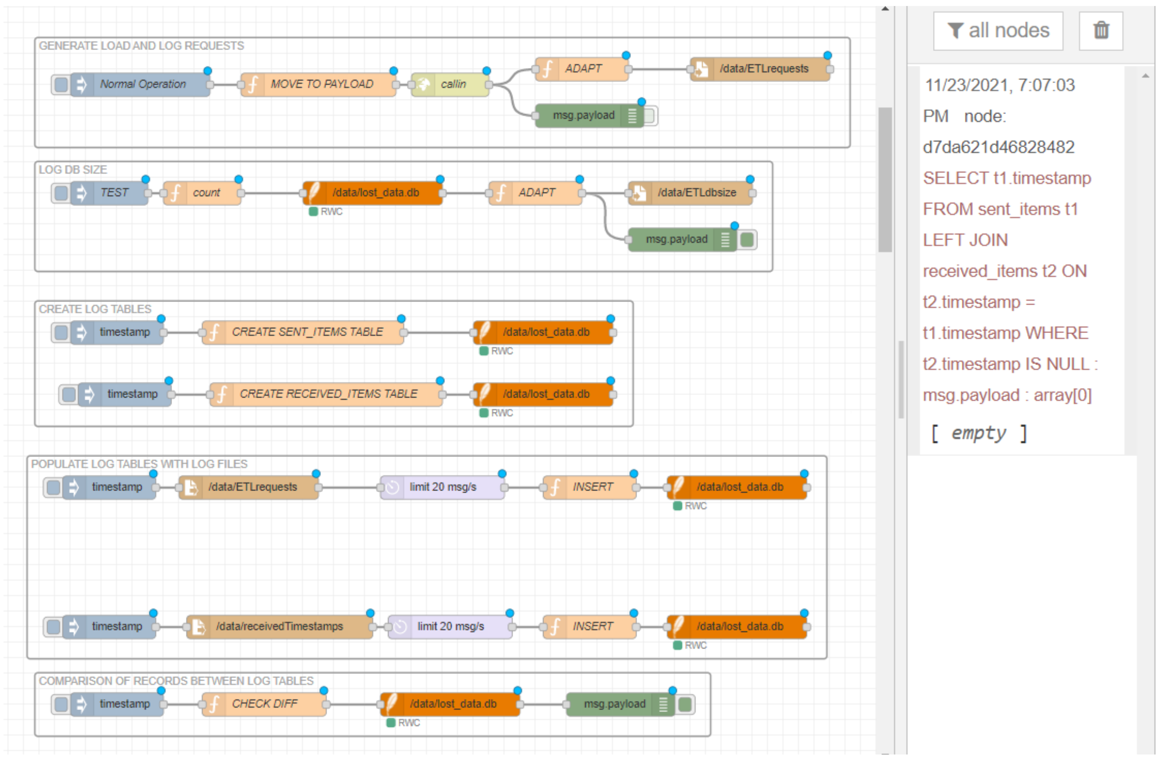
- Initially the target URL was set to change every 5 minutes to an existing endpoint and every 3 minutes to a non-existent one. The difference in the intervals thus generates a condition simulating the network outage that is more oriented towards the fail condition. This was selected in order to stress the system more, given that it would mean higher storage needs and burst messages for pushing the past failed data.
- A client generated 2 calls per second for creating new data items. This is significantly higher than the declared need of the Smart Agriculture case, whose current sampling rate is 1 new data item every 10 minutes, thus resulting to an equivalent of 1200 greenhouses message rate.
- The periodic cron job for past failed data submission was set to 2 minutes.
- The log files included the timestamp of each new data item call as well as a sampling of the database size every 10 seconds.
- The HTTP out endpoint documented the data items in a log file. Thus at the end of the experiment one can compare the log files at the client (included timestamps) with the log files of the HTTP out service to detect whether all samples have been pushed to the HTTP out endpoint.

The helper flows for the test appear in Figure 82. Initially a flow is used in order to generate the client requests and log the generation timestamp of each call. While the calls are performed, the flow below (LOG DB SIZE) retrieves and stores the row count of the DB. Upon experiment completion, the CREATE LOG TABLES box creates the two tables (sent_items and received_items) and the POPULATE LOG TABLES box feeds the logs into the DB. The COMPARISON box then applies the query that appears in the debug window in order to check differences between the two tables. No differences can be found. Thus no single data item is lost in the process, as was determined also by the comparison of the two log files, the generated messages and the received ones in a total of 18500 samples. Indicatively, the Smart Agriculture Use Case reports that the pre-PHYSICS implementation loses approximately 50% of the available data items.

The experiment was left to execute for ~2 hours. The evolution of the db size appears in Figure 83. From this it can be seen that the size fluctuates as expected from the interval difference in the simulated outage, and it also reaches a large number of points (~1400). This generates a burst of retries at the pattern side when the periodic cron job is activated, which however are manageable and result in cleaning the database in each cron job cycle.

4.6.7. Pattern variations

A variation of the ETL service pattern is the ETL function pattern. One may need to execute this pattern as a function. For adapting it, the main modifications are to change from the usage of the local mini DB to a function-external, edge-local storing mechanism as well as split the two processes (individual data item processing and periodic resubmission) to two functions. The second can be directly triggered periodically based on the Openwhisk scheduled tasks process. Two extra subflow nodes have been created for this reason. Further variations may be created by the user in order to adapt to different protocols for input and output.



11/23/2021, 7:07:03 PM node: d7da621d46828482
SELECT t1.timestamp FROM sent_items t1 LEFT JOIN received_items t2 ON t2.timestamp = t1.timestamp WHERE t2.timestamp IS NULL : msg.payload : array[0]

```
[ empty ]
```

Figure 82: Testing Flows for experimenting with the Edge ETL pattern

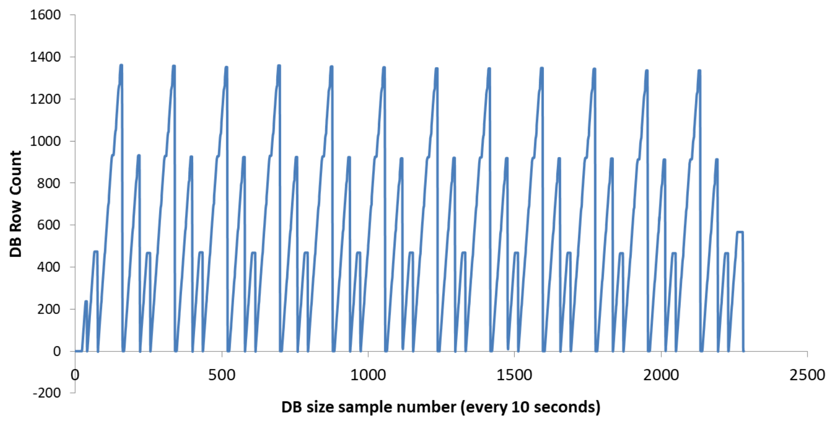


Figure 83: DB Failed Samples Variation over time in testing scenario

4.6.8. Pattern publication means

The pattern is embedded in the main PHYSICS Node-RED environment and offered through the palette. The source code of the subflow is available at the PHYSICS repository and it is self-contained (meaning that it can be directly copied and used in any Node-RED environment).

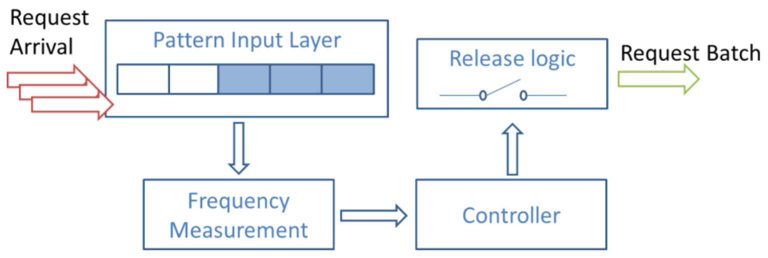
4.7. Batch Request Aggregator Pattern

4.7.1. Pattern template description

The template description for the Batch Request Aggregator appears in Table 15.

Table 15: Template Description for the Batch Request Aggregation Pattern

Pattern name	Heavy Thread Aggregator/ Batch Request Aggregator
Relation to Requirements	Non Functional- Performance & Cost, Req-4.2-FaaSBenchmarking (to check effectiveness of approach)
Source of pattern	Project research and innovation objectives, cost models of FaaS based on number of invocations
Pattern Context	Applies to either microservice or FaaS based implementations that involve high initialization times compared to the actual internal computation
Pattern Underlying Problem	Typical thread-pool server operations rely on threads (1 thread per request) that may need significant resources to launch and operate, while the work internally may be minimal. This is especially true when these functions/services are based on underlying heavyweight frameworks/environments such as AI or numerical computation frameworks. Especially under heavy traffic a significant number of these threads needs to be launched concurrently by the server, leading to further degradation of performance. This is especially augmented in the FaaS domain given that typically functions may be wrapped in containers and for each function invocation a relevant container needs to be spawned. In this case container tail latency (or start-up delay) further degrades performance, as well as queueing time in case the specific function reaches the maximum number of concurrent invocations.
Pattern Usefulness/ Objective	The goal of the pattern is to reduce the number of concurrent threads needed, or container invocations, by aggregating incoming requests in batches, not launching the main server threads (or respective containers) until a threshold is reached and then launch one single thread (or function triggered container in FaaS) that includes all the needed inputs as an array, significantly reducing performance overhead. Given that the main function invocations will be less, as well as less time consuming (due to avoiding multiple startup latencies as well as performance degradation from the concurrent containers), this aspect is also expected to improve the associated costs in the FaaS business model, since this is highly related to number of function invocation and time of execution. Expected results:

	● Private cloud: less concurrent containers, improved performance for the same amount of resources under heavy traffic ● Public cloud: less function invocations-> less cost
Schema	 Request Arrival Pattern Input Layer Release logic Request Batch Frequency Measurement Controller
Prerequisites	Function needs to accept array input, Ability to measure frequency of requests, some form of controlling logic
Condition of application	Frequency of incoming traffic determines whether the Aggregator is beneficial or not. If frequency is too low, waiting until a batch is complete before submitting it to the backend will be counter productive
Input parameters	Name of function to invoke, endpoint type
Output parameters	Response per request needs to be returned
Included functionality	Logic to enable or disable the pattern based on the conditions of traffic
Pattern limitations	Sequential execution of the rows in each batch.
Linked to Pattern	-
Indicative Domains of Applicability	Performance/AI model querying, Numerical simulations

4.7.2. Pattern implementation details

Addressing increased demand in cloud environments is typically achieved through auto-scaling the use of further resources allocated. However, this comes at the trade-off of increased back-end resource stress as well as runtime costs. In many application domains, such as Artificial Intelligence, performance estimation, resource management etc., service implementations are required to serve incoming requests by leveraging heavyweight computational environments and libraries. These are used in order to apply created models to desired request inputs and respond with the according prediction.

When these functionalities are offered as a service, through typical thread-pool server implementations, this implies that for each incoming request a relevant environment needs to be spawned in order to process it. This creates extreme stress on a server for mainly two reasons. Firstly, the initialization time of such an environment thread is typically non-negligible. A performance analysis on such a server implementation [47] using the Octave environment[24] indicated that above 70% of the total serving time (1.2 seconds) for a single client request was due to raising the respective Octave environment thread. Only 10 milliseconds of computation time were needed for the actual computation, indicating an extreme overhead of preparation in relation to the actual computation time. Secondly, when multiple simultaneous requests are sent towards the server, an according number of threads need to be concurrently run and compete for resources such as memory and CPU, while interfering with peripheral elements such as cache memories. The aforementioned competition creates a further increase of.the response times, while concurrency overheads can reach levels of 400% of performance degradation [26].

Similar overheads apply for more modern computing models such as serverless and FaaS, in which containers with designated functions are executed in order to respond to an incoming request endpoint. In this case, the process itself of raising the containerized environment is even worse than raising another thread on an existing server. Overhead inserted by this computing model is heralded as one of the main challenges of the domain [48].

In this pattern a request batch aggregation and management pattern is implemented, especially for cases of computational load that needs a heavyweight environment setup (such as performance prediction and AI services). The pattern acts as a preprocessing layer and accumulates requests, withholding their forwarding to the backend based on the conditions of execution (frequency of requests), in order to regulate back-end resource stress without adding extra resources. Before proceeding to the main pattern description, a comparison to the current typical scaling patterns in serverless environments needs to be performed, in order to indicate the differences of the approach to the current standard mechanisms of serverless platforms.

Scaling patterns in FaaS

Various scaling patterns are available in serverless offerings such as AWS Lambda, Google and IBM Cloud Functions as well as Knative and Cloud Run [49]. Based on the findings, two main modes of auto-scaling connecting the number of containers to function requests are identified, in addition to a third one based on node metrics. Per-request scaling (Figure 84a) raising a separate container per function request (most typical case) and concurrency value scaling (Figure 84b), enabling the concurrent execution of multiple functions in one container. The Request Aggregator pattern appears in Figure 84c. In this case we propose the aggregation or incoming requests before submitting them to the back-end. Release logic can be on batch size, timeout threshold or a combination of both. This would enable reduction of total containers running in the system,

[24] https://www.gnu.org/software/octave/index

compared to case (a), while removing function concurrency overheads that can occur in case (b), especially for computationally intensive workloads.

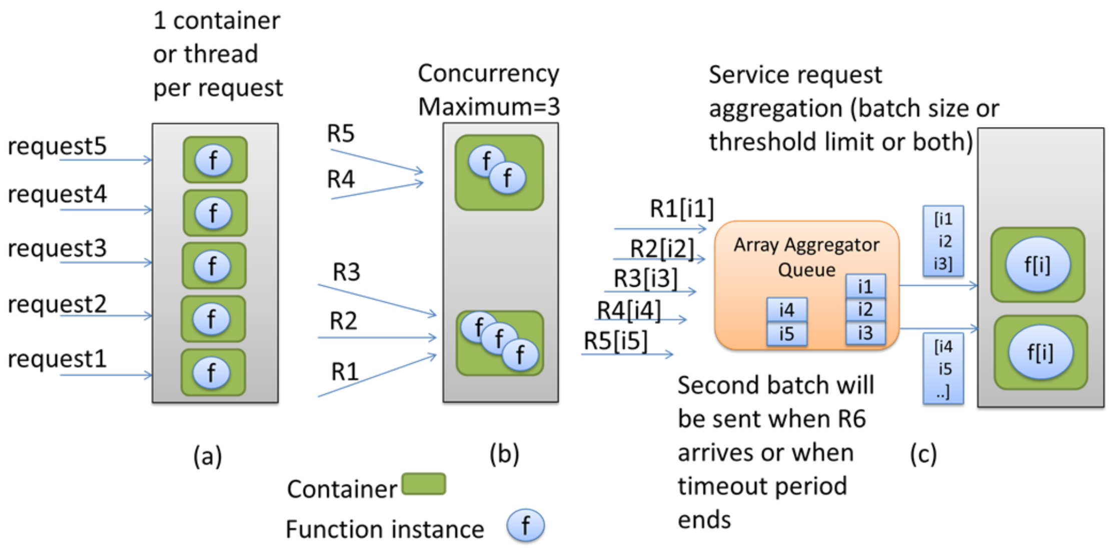


Figure 84: Scaling Patterns (a),(b) from [33] compared to the proposed pattern (c)

One aspect to notice with relation to serverless environments is the term "concurrency". One variation of the term relates to the total number of the function activations in a namespace that execute across the cluster (in different containers) and is more related to throttling aspects, so that a specific namespace and its functions does not overwhelm the overall quota of function executions available to a user [36]. This needs to be distinguished in terminology compared to the case of "function concurrency scaling factor [50]" that is presented in Figure 84b.

The queue-based load levelling pattern[25] resembles the rationale of the batch request aggregator, in the sense that it does not allow the requests to reach the backend in order not to create unpredictable peaks in load. For this reason, it accumulates messages in a queue, from which the consuming function/service retrieves them at its own pace. The main difference here is that the queue-based load levelling pattern does not aggregate function inputs into one function execution in order to optimize costs and improve response times, but it only acts as a queue from which the back-end can retrieve tasks at its own pace. Thus, significant waiting time is added to the requests.
In order to achieve the needed functionality (gathering of incoming requests, creation of batches and launching of processing instances), the pattern needs supporting logic that is presented in Figure 85.

[25] https://docs.microsoft.com/en-us/azure/architecture/patterns/queue-based-load-leveling

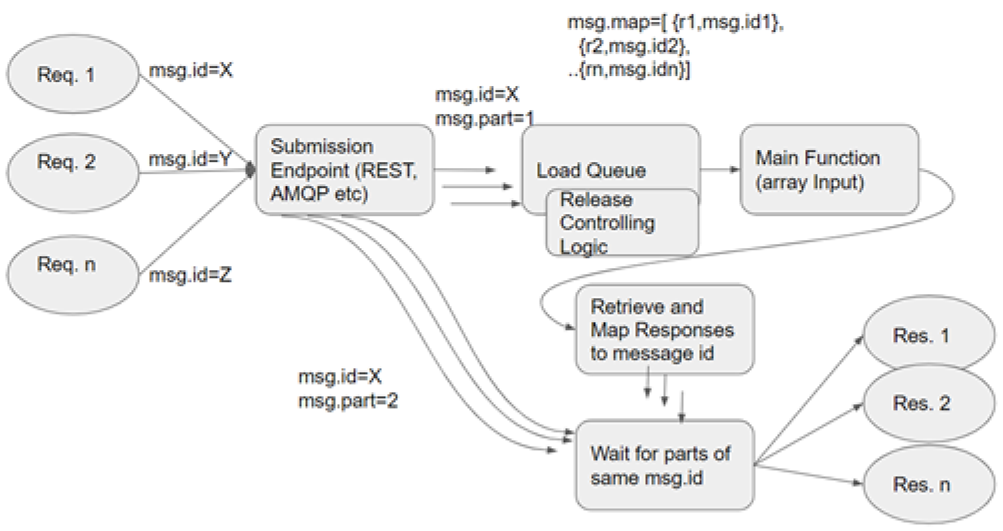

msg.map=[{r1,msg.id1},
{r2,msg.id2},
..{rn,msg.idn}]
Req. 1
msg.id=X
Req. 2
msg.id=Y
Req. n
msg.id=Z
msg.id=X
msg.part=1
Submission Endpoint (REST, AMQP etc)
Load Queue
Release Controlling Logic
Main Function (array Input)
Retrieve and Map Responses to message id
msg.id=X
msg.part=2
Wait for parts of same msg.id
Res. 1
Res. 2
Res. n

Figure 85: Batch Request Aggregator Pattern Structure

The implementation consists of the following main building blocks:

- A submission endpoint accepting external requests and assigning a message id to each one
- A request accumulation layer that stores the requests in a queue, including a map for indicating which external requests have been included in the batch, based on their message id.
- Release logic that launches the request batch once specific criteria are met. This logic may include the calculation of request arrival frequency over a time window.
- A response creation layer in which the original incoming messages wait until the response from the main operation is available. When the overall response is available, it is broken down based on the id map, the individual messages are completed and returned to the clients.

A functional programming framework based on function workflows and message passing between function nodes (such as Node-RED) is a good candidate in order to implement such a logic. To implement the logic of Figure 85, the function flow of Figure 86 was created. Supporting flows were also created in order to aid in management decisions, such as frequency measurement of requests from the system scope as well as requests towards the controlling logic decision function. Monitoring subflows for logging running container numbers are also included.

4.7.3. Pattern examples of usage

In order to use the Request Aggregator, the developer needs to utilize the respective PHYSICS node as the inner logic of an HTTP In and Response nodes (Figure 87). In this manner they can set up whatever input URL they need. What is more, the RA node needs to be properly configured via the node UI. In this setup, they need to insert the target URL for forwarding the batch of the requests, as well as which mode to utilize, simple time out, static batch size or model based batch size setting (more details on these settings are included in the Variations section of this chapter). If the dynamic model-based size setting is chosen, a relevant existing modelID should also be included in order to query for the batch setting. Furthermore, an Executor Mode node needs to be used, in order to indicate that this flow needs to be executed as a service.

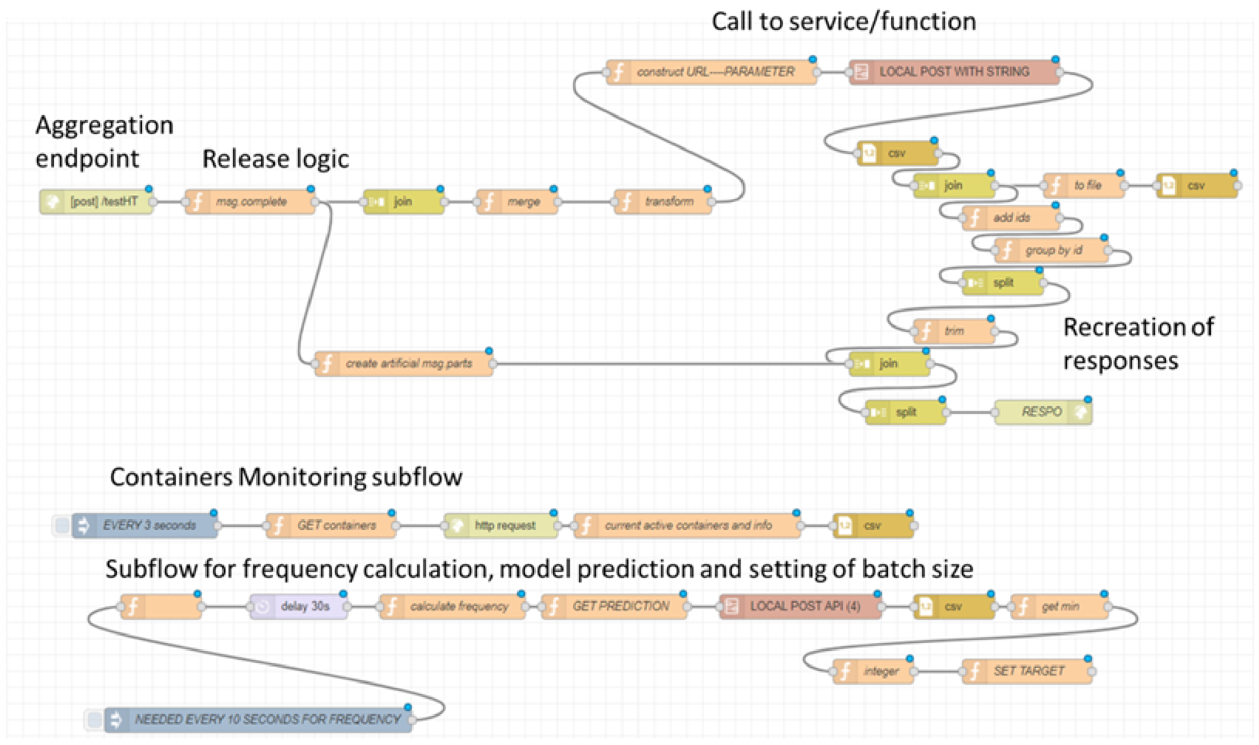
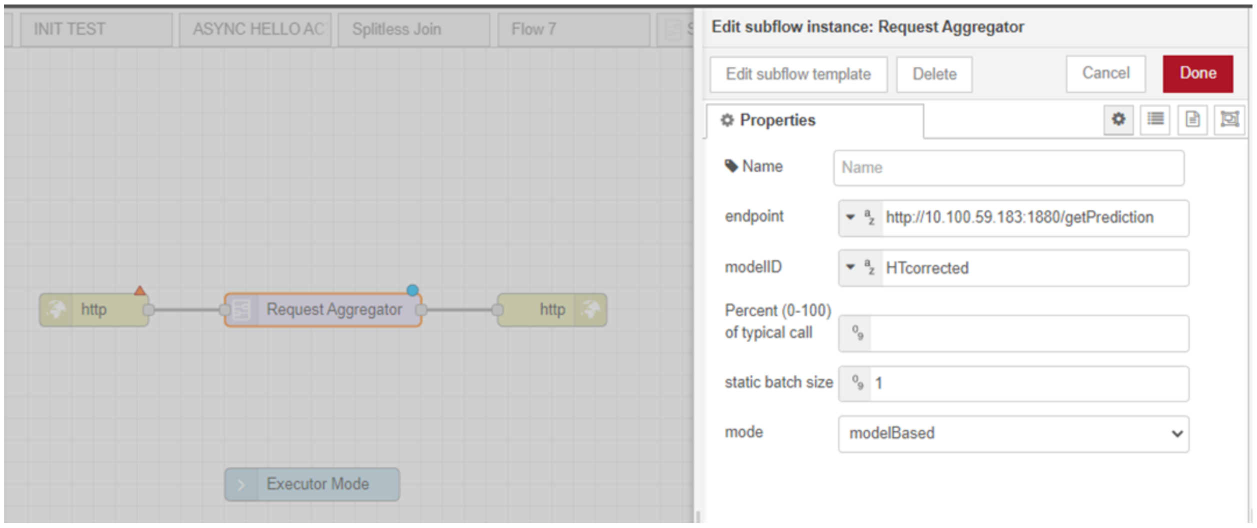


Figure 86: Node-RED flow implementation of the Batch Request Aggregator pattern



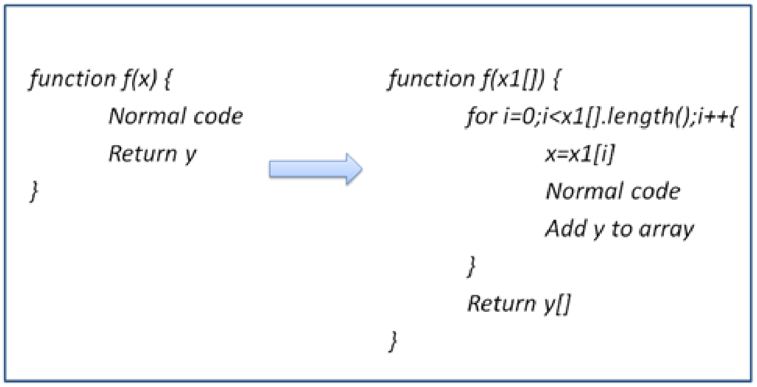
Edit subflow instance: Request Aggregator

Name	Name
endpoint	http://10.100.59.183:1880/getPrediction
modelID	HTcorrected
Percent (0-100) of typical call	
static batch size	1
mode	modelBased

Figure 87: Example flow for the inclusion of the Batch Request Aggregator

4.7.4. Pattern necessary adaptations

In order to apply the respective pattern, the main code modification of the typical function (or microservice) includes its ability to accept an array of the necessary input argument type (regardless of the argument type i.e. integer, double, array, object etc). Then for each element of this array it should process each request, while returning an array of responses in the same order the inputs where received. The pseudo code for the function modification appears in Figure 88.



```
function f(x) {
        Normal code
        Return y
}
```

```
function f(x1[]) {
        for i=0;i<x1[].length();i++{
                x=x1[i]
                Normal code
                Add y to array
        }
        Return y[]
}
```

Figure 88: Example of function adaptation for use with Request Aggregator

4.7.5. Pattern limitations

One limitation of the pattern is in case some information or state needs to be retrieved from an external repository. Such a case would be for example a function flow in FaaS that retrieves a different AI model per request from object storage. Given that the array input now is executed in sequence, rather than in parallel in the case of multiple function containers, the model retrieval is also executed in sequence which may produce large response time delays. In this case alternate configurations or adjustments may be made such as:

- Batching together only requests that target the same model ID
- Keeping model repositories in volumes that are attached dynamically to the containers

Furthermore, global context isolation can not be guaranteed and multitenancy issues may exist (if batching requests from different users). However, the latter may be mitigated if presigned URLs (or the Valet Key pattern) are used in order to retrieve the tenant's model from a private repository.

4.7.6. Pattern variations

Different variations for the pattern are based on diverse release logic to determine the conditions under which the aggregated request batch can be forwarded as one message in the backend layer. It is evident that the frequency of request arrivals plays a key role in this decision. If for example requests are sparse, having a large batch size will imply that the first requests need to wait for a considerable time until the batch is complete, resulting in higher waiting times and overall response time. On the other hand, if requests are very frequent, having a small batch size will lead to higher container numbers.

Thus one needs to regulate the parameters of the pattern based on the current conditions of execution in order to optimize the overall result. Indicative approaches for this case can include:

- a simple timeout period, during which the requests are batched. Each arriving request checks the elapsed period from the previous checkpoint and, if complete, it populates a msg.complete field that alerts the next layer to release the batch. Setting the timeout period as a percentage of the typical response time in single request scenarios can help predict the final response time. For example, a 10%*(averageResponseTime) timeout period will result in only 10% according increase in the final experienced response time, compared to the response time of the service under sparse load.
- a set batch size that needs to be completed before the batch is released. However, in this case the batch size needs to be dynamically regulated during runtime, so that it is set to 1 if the request frequency is low or higher if the frequency is high. The control of the batch size can be based on various methods, such as neural networks, as investigated in D4.1. These need to create a model of the response time, that will be fed with the current conditions of the system (i.e. request arrival frequency) and direct the needed metric (e.g. batch size or timeout interval. An example form of such an ANN model appears in Figure 89. Having the response time as the output is needed primarily for validation reasons.

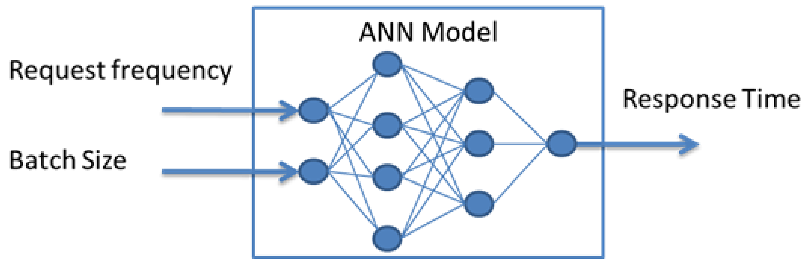


Figure 89: Model Structure for Usage with the Request Aggregator Batch Regulation

4.7.7. Pattern experimentation outcomes

For the experimentation, the initial service implementation of the Model Query operation of the Performance Evaluation Framework of T4.2 was used that appears in Figure 90. This is a performance estimation service that has created ANN-based performance models, stored in a container volume, and enables the enquiry of these models through a REST endpoint. Whenever a request is received, an equivalent container is launched in order to serve the request, run the environment and give back the prediction. The environment is based on the GNU Octave numerical computation tool, an open source equivalent of Matlab. Based on the structure and operation, this service is a hybrid between serverless and microservice environments (REST endpoint and equivalent container launch for serving a request). In principle, any service that includes large preparation overheads compared to the actual useful computation is a target. Such example services are typically AI-based ones (such as model inference) or approaches that create a large number of requests (e.g. Monte Carlo methods) and are based on heavyweight libraries or frameworks. Given the fact that the pattern acts as a preprocessing layer, it can be applied in both typical microservice, threadpool based servers (in order to reduce the number of concurrent threads in the server) as well as in front of the function invocation gateway (in the case of serverless architectures).

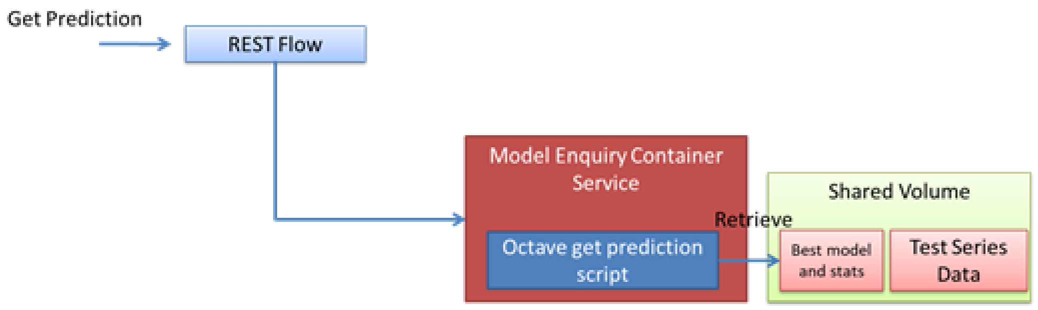


Figure 90: Target Service for Investigation of Benefits of the RA Pattern

In order to test the performance and benefits of the pattern, a series of experiments were performed, in order to stress the system and observe the altered behavior. The resource was a single VM node (4 CPUs, 10GB RAM). No extra resources were made available to the service load in order to evaluate the effect of the pattern without the need to increase the resources. As the main system performance metric, the response time of the service to the various request scenarios and frequencies was considered.

Initial Investigation of Batch Size and Request Period on Response time

Initially a set of separate measurements is performed in order to investigate the effect of the batch size on response times. For this reason a number of different periods (every 0.01, 1 and 5 seconds) of incoming requests and an according diverse batch size (1, 5, 10, 50) is investigated in order to observe their effect on the average response time. The results appear in Figure 91.

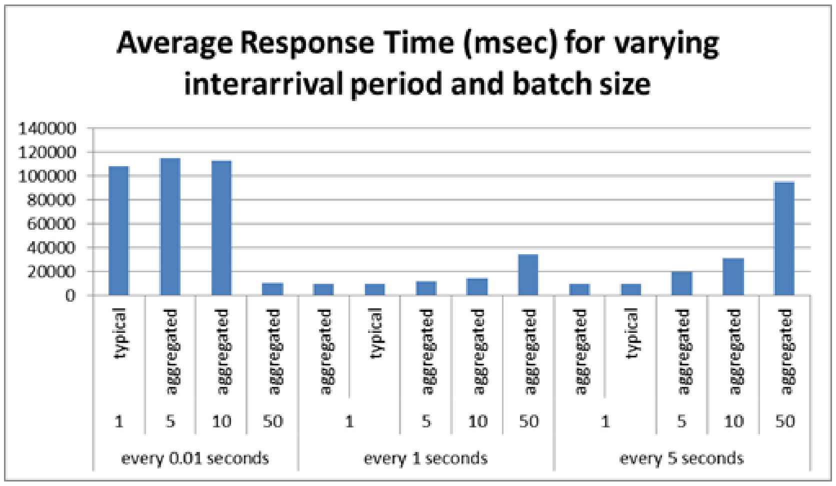


Figure 91: Investigation of Batch Size and Inter-arrival Period on the Response Time Average

The timeout is set to 120000 milliseconds, so the respective values in the graph indicate an unresponsive system due to resource contention and high container numbers. The typical mode is 1 container per request without the usage of the pattern. The aggregated mode with a batch size of 1 applies again 1 container per request but this time through the pattern, in order to investigate delays inserted by the pre and post processing layers.

From the measurements it can be portrayed that the pattern includes some delay due to the request management layer (when comparing typical versus aggregated mode of 1 batch size) in the area of ~ 1 second (or <10% of the total response time). But also, the benefit from the pattern application is portrayed, in the case of high frequency requests (inter-arrival period of 0.01 seconds). In this case the typical system that was previously unresponsive, when applying the aggregation mode with a sufficiently large batch size (~50) manages to stabilize its behavior as well as maintain an average response time that is very close to the average of a single request (10.763 seconds of average response in a period of 0.01 seconds and batch size of 50 versus 10.046 seconds of a typical mode for 1 request every 5 seconds). On the other hand, once the period between requests starts to get higher, the drawback of the pattern is portrayed, given that it needs to wait a large amount of time until the batch size is complete.

Runtime Container Numbers for gradually increasing request frequency

In order to investigate the overall performance of the pattern, a series of measurements was performed, applying the timeout variation. Overall the pattern was tested under a scenario in which the request inter-arrival period was gradually reduced, starting from 10 seconds and being reduced to 5,1, 0.5, 0.1 and 0.01 seconds of inter-arrival. Each setup was set to run for 15 minutes, unless the platform started to get saturated from container numbers in which case the experiment was halted.
The two main variations that were tested were static delays of 500 and 1000 milliseconds before releasing the batch, which is approximately 5% and 10% of the average response time in a single request. During this time the framework accumulated requests coming in. The results of the experiment are presented in the following figures and are compared to the typical, no-batch implementation. It is indicative that as the experiment progresses and frequencies increase, the batch size (accumulated messages during the timeout interval) increases (Figure 92).

This enables the pattern to maintain a constant rate (~20 active containers) of container generation (Figure 93) even at high frequencies (as indicated by Figure 94), in the 10% timeout case (1000 milliseconds of waiting for gathering and batching requests). The no-batch implementation quickly gets saturated (at around sample ~1500 of the experiment), as indicated also from the response times that reach the request timeout limit (Figure 95), reaching very quickly the saturation point of around 80 active containers. This is mapped to a frequency of around 2 messages per second (Figure 94). The 5% timeout is still not sufficient, although it gets saturated a bit later (around sample 5000 of the experiment), achieving a frequency of around 10 messages per second.

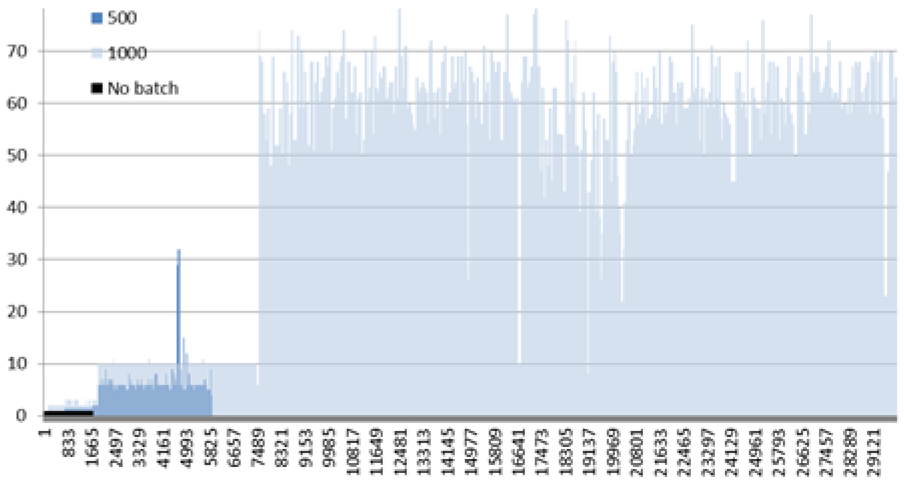


Figure 92: Average batch size of requests (Y axis) in the system as the experiment progresses (X axis:sample number of experiment)

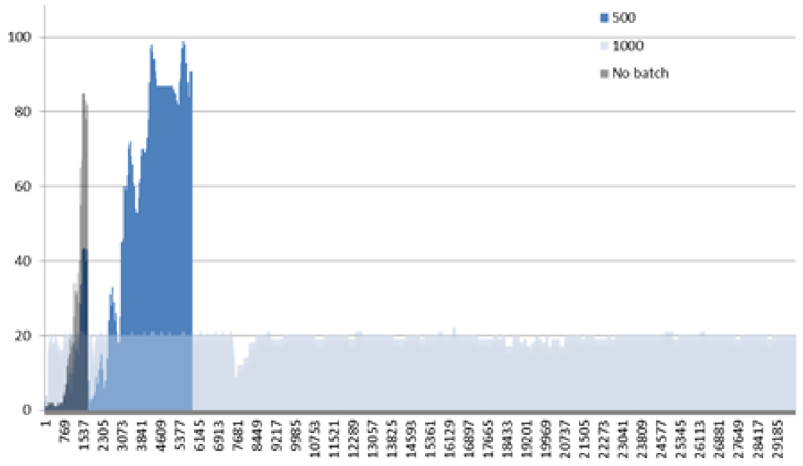


Figure 93: Average number of containers (Y axis) in the system as the experiment progresses (X axis:sample number of experiment)

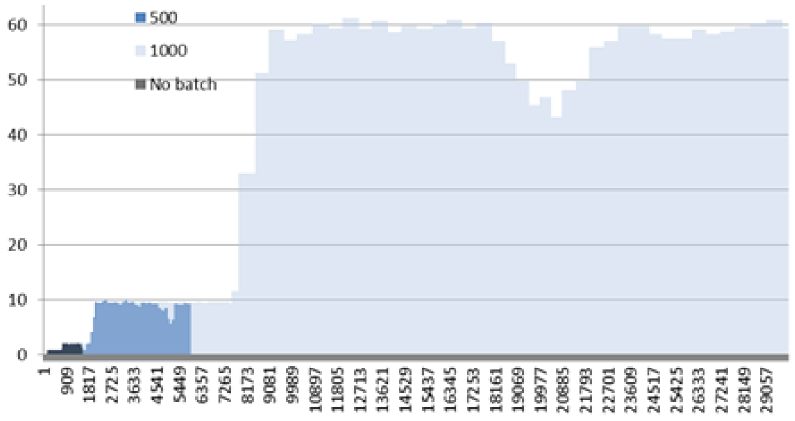


Figure 94: Average frequency of requests (Y axis) in the system as the experiment progresses (X axis:sample number of experiment)

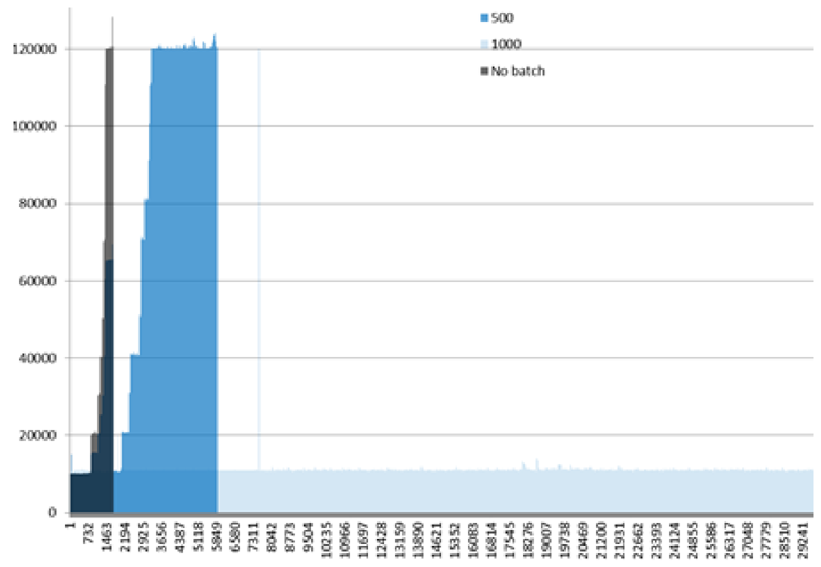


Figure 95: Response time of requests (Y axis) in milliseconds as the experiment progresses (X axis:sample number of experiment)

4.7.8. Pattern publication means

The pattern implementation is available as a subflow in the PHYSICS environment as well as an individual flow[26]. A relevant publication [51] has also been derived from this pattern.

4.8. Cryptography Pattern

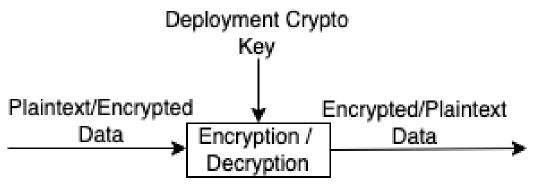
4.8.1. Pattern template description

The template description for the Cryptography Pattern appears in [Table 16](#).

Table 16: Template Description for the Cryptography Pattern

Pattern name	Cryptography,
Relation to Requirements	Req-3.4-Encryption, Req-3.4-SecureComms, Req-6.2-Privacy
Source of pattern	All UCs
Pattern Context	Basic cryptographic functionality to be used for data encryption.

[26] https://flows.nodered.org/flow/a97ec190bbf75cec594092895d39c01d/in/HXSkA2JJLcGA

Pattern Underlying Problem	Utilization of cryptography can be complicated and can create security issues if not correctly executed.
Pattern Usefulness/ Objective	Provide easy to deploy and use encryption/decryption functionalities.
Schema	 Deployment Crypto Key Plaintext/Encrypted Data Encryption / Decryption Encrypted/Plaintext Data
Prerequisites	To deploy the function with a strong cryptographic key in a secure manner.
Condition of application	N/A
Input parameters	The data to be encrypted or decrypted on the payload (msg.payload.data) of the HTTP request and the function (encryption/decryption) in the msg.payload.function.
Output parameters	The data that was encrypted or decrypted on the payload (msg.payload.data) of the HTTP request.
Included functionality	The encryption and decryption of the specified data.
Pattern limitations	The cryptographic key is set during deployment time of the function, this creates the need for a secure method to deploy this key.
Linked to Pattern	OW Skeleton
Indicative Domains of Applicability	Data protection

4.8.2. Pattern implementation details

The pattern is implemented using the encryption and decryption node of the node-red-contrib-crypto-js package using AES encryption (Figure 96).

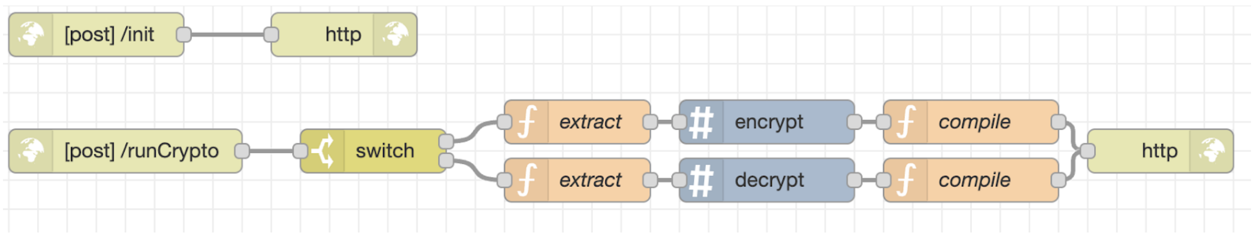


Figure 96: Encryption/Decryption Flow Implementation

The encryption key is deployed with the instantiation of the flow in the ${KEY} environment variable. The triggering message is then first checked for what functionality is requested (encryption or decryption) and it is then processed so that the data is processed correctly. Finally, when the function is executed, the reply message is compiled and sent as an http response.

4.8.3. Pattern examples of usage

The two basic examples are the encryption and decryption use cases. Whenever someone needs to encrypt or decrypt their data, they can utilize this pattern to do so.

4.8.4. Pattern necessary adaptations

N/A

4.8.5. Pattern limitations

The pattern requires that the key is deployed as an environment variable (${KEY}), extra care must be taken when doing so.

4.8.6. Pattern variations: Encrypted Storage

One variation of the cryptography pattern should be created that includes a basic cryptographically-protected storage functionality to be used for data encryption and secure storage. Utilization of cryptography in conjunction with the storage of the encrypted data can be complicated and can create security issues if not correctly executed. The purpose of this variation is to provide easy to deploy and use encrypted storage functionalities. The extended diagram of the cryptography pattern in this case is depicted in Figure 97.

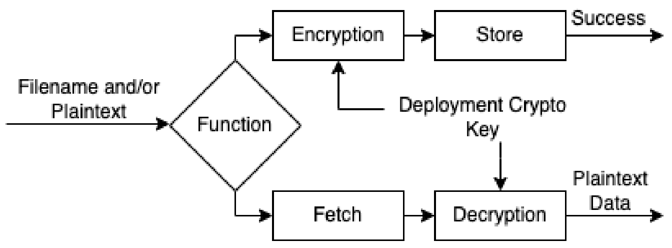


Figure 97: Diagram of the Encrypted Storage variation

The input parameters include the file name (*msg.payload.value.filename*) and/or data to be encrypted on the payload (*msg.payload.value.data*) of the HTTP request and the function (encryption/decryption) in the *msg.payload.value.function*. The output responses include The file name (*msg.payload.filename*) that was stored or the decrypted data on the payload

(*msg.payload.data*) of the HTTP request. The cryptographic key is set during deployment time of the function, this creates the need for a secure method to deploy this key.
Like the cryptography pattern, this pattern is implemented using the encryption and decryption node of the node-red-contrib-crypto-js package using AES encryption (Figure 98).

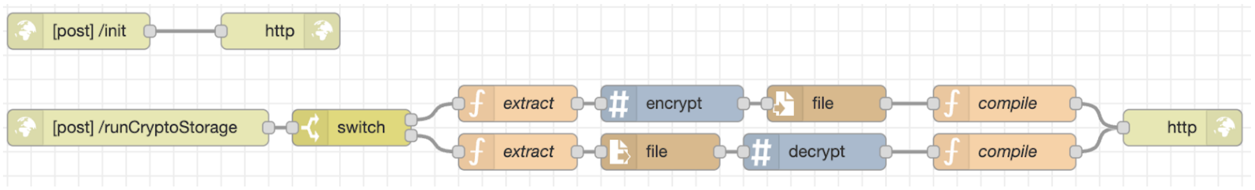


Figure 98: Encrypted/Decrypted Storage Variation of the Cryptography Pattern

The encryption key is deployed with the instantiation of the flow in the ${KEY} environment variable. The triggering message is then first checked for what functionality is requested (encryption/store or fetch/decryption) and it is then processed so that the data is processed correctly. Finally, when the function is executed and the data is either encrypted and stored or fetched and decrypted, the reply message is compiled and sent as an http response.

4.8.7. Pattern publication means

The pattern is available in the PHYSICS palette of the environment as well as an individual flow in the PHYSICS repository.

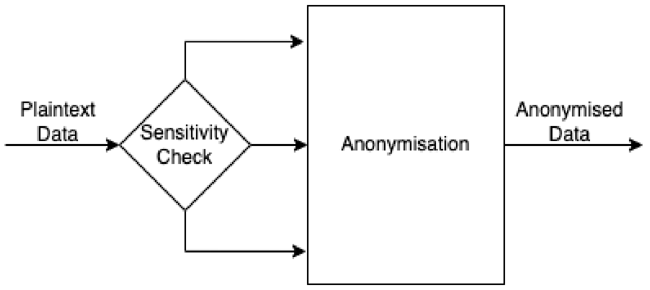
4.9. Custom Anonymization Pattern

4.9.1. Pattern template description

The template description for the Custom Anonymization Pattern appears in Table 17.

Table 17: Template Description for the Custom Anonymization Pattern

Pattern name	Custom Anonymization
Relation to Requirements	Req-3.4-Privacy
Source of pattern	All UCs
Pattern Context	Provide basic data anonymization functionality.
Pattern Underlying Problem	The complexity of the utilization of anonymization and its correct implementation.
Pattern Usefulness/ Objective	Provide easy to use and deploy anonymization functionalities.

Schema	 Plaintext Data Sensitivity Check Anonymisation Anonymised Data
Prerequisites	Potentially modify some of the functionalities for each sensitivity level so that it works according to the needs of the use case.
Condition of application	N/A
Input parameters	The data to be anonymized in msg.payload.data and the sensitivity level in msg.payload.sensitivity.
Output parameters	The anonymised data in msg.payload.data.
Included functionality	A multi-level sensitivity anonymisation functionality.
Pattern limitations	It does not have an automated analysis tool for the dynamic detection of sensitive data. The user sends the data and all of it is anonymised according to the declared sensitivity.
Linked to Pattern	N/A
Indicative Domains of Applicability	Data privacy

4.9.2. Pattern implementation details

The function flow (Figure 99) utilizes the digest node of the node-red-contrib-crypto-js package using SHA512 function. Based on the declared sensitivity the data is either hashed, grouped in a +/-10 groups or replaced with an asterisk. If there is no sensitivity, then the data is left as is. After the processing of the data, the response is processed and returned anonymised in an http response. Each function can be customized so that the desired function is executed.

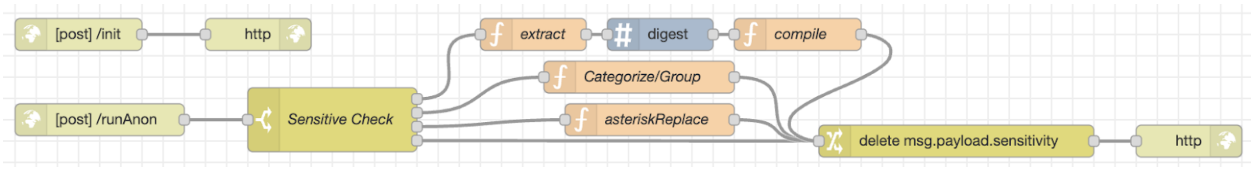


Figure 99: Implementation Flow for the Custom Anonymization Pattern

4.9.3. Pattern examples of usage

This pattern is a custom anonymisation function that acts upon the entire input data according to the specified sensitivity of the data. There are already defined functions that anonymize the data, but according to the needs of the user, these functions can change to create custom functionalities depending on the need of the use case. After deployment, all the user needs to do is to provide the data to be anonymized and its sensitivity.

4.9.4. Pattern limitations

There is not an automated data analysis for the identification of sensitive data. Thus the sensitivity of the input data is declared manually and the function is applied on the entire data input. Furthermore, the categorize/group function only accepts numbers since by their definition, only numbers can be grouped in +-10 groups. Further adaptations will be investigated to incorporate additional data types, this issue is solved by the presidio anonymization pattern.

4.9.5. Pattern variations: Presidio anonymization pattern

In order to support and provide basic data anonymization functionality with automated sensitive data detection, a variation of this pattern was created (Figure 100). The goal is to provide easy to use and deploy anonymization functionalities that automatically detect types of sensitive data and act on them according to a set of rules. To use this pattern a presidio[27] analyzer and presidio anonymizer servers (open source tool) need to be running and their IPs recorded inside the corresponding nodes.

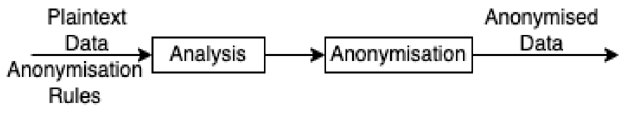


Figure 100: Presidio Anonymization Pattern diagram

The data to be anonymized are included in the *msg.payload.value.data* and the anonymisation rules in *msg.payload.value.anonymizers* (in JSON format according to the presidio documentation). The anonymized data are included in the *msg.payload.data* of the output. This pattern improves upon the custom anonymization pattern by utilizing the presidio analyzer to automatically identify all the types of sensitive data in an input and anonymize only them using the presidio anonymizer. The ideal use case for this pattern is to feed it with any size of data and the rules that specify how types of data are handled (dates, names, credit cards etc.) and the pattern will return the data with only the sensitive types anonymized.

[27] https://microsoft.github.io/presidio/

This pattern requires that the presidio analyzer and anonymizer are deployed and accessible as http servers. The presidio analyzer first checks the entire input data for sensitive data (Figure 101), after this, according to predefined rules, each data type is processed accordingly. There are readily available functions from presidio but this can also be extended with the definition of custom lambda anonymization functions.

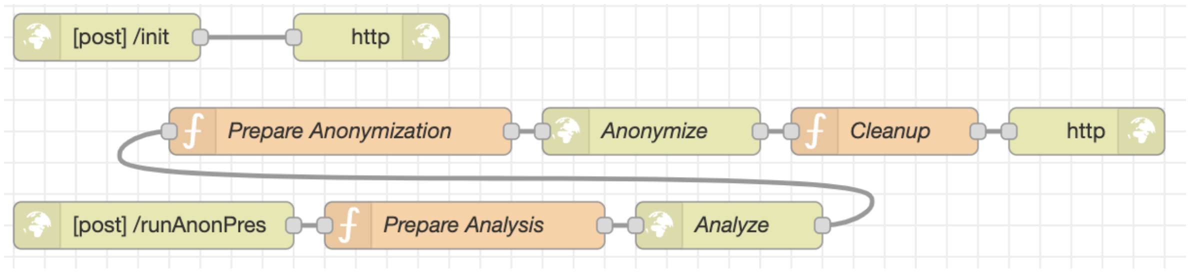


Figure 101: Presidio Anonymization Implementation Flow

From the client side of this pattern, there should be a definition of the rules that handle the different data types identified in the input data. The anonymization rules are defined in JSON format which requires a little more complexity from the caller.

4.9.10. Pattern publication means

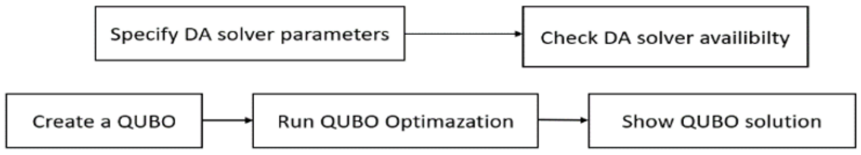
Both patterns are available in the PHYSICS palette of the environment as well as individual flows in the PHYSICS repository.

4.10. Digital Annealer Quantum-Inspired Optimization Pattern

4.10.1. Pattern template description

The template description for the Optimizer Pattern based on the Digital Annealer Unit (DA) appears in Table 18. The DA is designed to solve combinatorial optimization problems at high-speed and to find very good solutions near the global optimum. For this, the combinatorial optimization problem has to be formulated as a Quadratic Unconstrained Binary Optimization (QUBO) problem. The same QUBO formulation can be used on Quantum Annealer or Quantum Gate Computers as soon as they are available to solve problems of such a size. You can find more information about the method in this and the connected papers [52] . In this section we provide an example how such a QUBO problem should be formulated and then how it can be solved using DA. A similar process should be used based on other problems of interest. The QUBO formulation is problem specific and, depending on your constraints or the weighting of the preliminary factors, not necessarily unique. This section can be used to get a first idea of the underlying approach. If you want to know more, you can find a more detailed Tutorial with a lot of different examples from academia to industry-related in the Digital Annealer Tutorial [53].

Table 18: Template Description for the Digital Annealer Optimizer

Pattern name	Digital Annealer Quantum-Inspired Optimization Pattern
Relation to Requirements	Req-5.4-optimization
Source of pattern	N/A
Pattern Context	Solving Quadratic Unconstrained Binary Optimization (QUBO) pattern by using Fujitsu's Digital Annealing Unit.
Pattern Underlying Problem	Using the Digital Annealing Unit can be complex. This flow should ease the process to call the Digital Annealing Unit API in the correct steps. It also provides the basis for further patterns.
Schema	 Specify DA solver parameters Check DA solver availibilty Create a QUBO Run QUBO Optimazation Show QUBO solution
Pattern Usefulness/ Objective	This flow should ease the usage of a Digital Annealing Unit to solve a Quadratic Unconstrained Binary Optimization problem.
Prerequisites	If the problem fits into a 1024^2 matrix, no prerequisites are necessary. If it's bigger or you want to use all features of Digital Annealer V3, credentials must be available for using the DAU.
Condition of application	N/A
Input parameters	Quadratic Unconstrained Binary Optimization problem

Output parameters	Optimised solution
Included functionality	Solving a Quadratic Unconstrained Binary Optimization problem.
Pattern limitations	The optimization problem can be solved with a DAU, which is available in different versions: Version-2 with Annealing or Parallel Tempering, Version-3, CPU emulator). Maximum QUBO request body is at 1024^2 bits for the CPU Emulation solving method.
Linked to Pattern	N/A
Indicative Domains of Applicability	All

4.10.2. Pattern implementation details

The pattern shows some of the first functionalities of Digital Annealer at the example of an typical assignment problem: In collaborative work environments or shared workspaces, there often arises the need to distribute tasks or responsibilities among two individuals or teams efficiently. This scenario can be likened to the classic “Two Persons Assignment” problem, where a set of items, representing tasks or projects, must be allocated to two persons in a manner that minimises the difference in the workload or items assigned to each person. Consider a co-working space with two freelancers, who share various projects and responsibilities. To ensure fairness and productivity, it is crucial to optimize the allocation of tasks between them. This is where the “Two Persons Assignment” pattern proves invaluable. Other problems which need a load balancing can also be formulated as a two-person assignment pattern.

The pattern flow solves the two-person-assignment as an example and provides the basics for other QUBO optimization patterns. The goal of the studied example is to divide a set of items (given as a list of values) between two persons in such a way that the difference in the sums of the assigned items is minimised.

The pattern can be used to solve the QUBO problems. The optimization could be performed using Digital Annealer Unit API (Figure 102) or using a CPU solver, but only for QUBOs with a size less than 1024 bits. A concrete explanation of the solving parameters is explained in the following subsection.

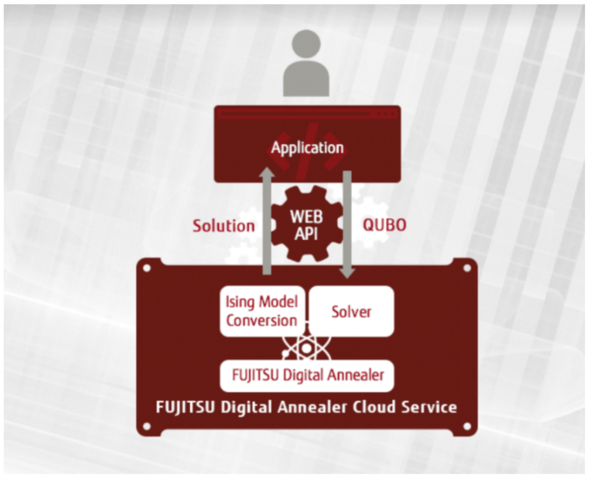


Figure 102: Fujitsu Digital Annealer Unit

4.10.3. Pattern examples of usage

This is an example in how to use the flow to solve a QUBO (here the two-person assignment) using the Quantum-inspired annealing approach.

A. Define Solver parameters:

In this step, the flow pattern user specifies the parameter for solving the optimization problem. The parameters to specify are:

- Solver device (DA or CPU):
    - DA allows the user to solve QUBO problems using the Digital Annealer Units in different versions and with different features. This method requires an API key to access the DA cloud service. (Please contact the Fujitsu DA team to get credentials.)
    - CPU is an emulator of the DA solver. The CPU could only solve QUBO problems a maximum of 1024 bits.
    - CPU is an emulator of the DA solver. The CPU could only solve QUBO problems a maximum of 1024 bits.
- Solver version (V2 or V3):
    - V2: Version 2 of Digital Annealer supports QUBO problems with up to 8192 bits. For V2, the user could choose between two solving methods DA2 (Digital Annealing) or DA2PT (Parallel Tempering).

- V3: Version 3 of Digital Annealer introduced the Digital Annealer System (DAS), which integrates one or multiple Digital Annealer Units (DAU). This system automatically decomposes big QUBO problems into smaller subproblems, which can be solved on the Digital Annealer Units. The results of the subproblems are used to create solutions for the full problem. This concept allows us to overcome limitations in the number of bits that can be supported by a hardware unit like DAU. The Digital Annealer System supports QUBO problems with up to 100,000 bits.

- Solver method (DA2 or DA2PT, only for solver version V2):
  - DA2: The Simulated Annealing Algorithm is a general-purpose metaheuristic algorithm to solve NP-Hard optimization. Simulated Annealing is conceptually based on metallurgical annealing where a crystalline solid is heated and then allowed to cool very slowly until it achieves its most stable lattice energy state. If the cooling schedule is sufficiently slow, the final configuration results in a solid with best structural stability. Simulated Annealing establishes the connection between this type of thermodynamic behaviour and the search for a global minimum for a discrete optimization problem. The algorithm was set up into a ASIC architecture, to generate the high solving speed.
  - DA2PT: Parallel Tempering executes the optimization on multiple temperatures in parallel on several replicas to find the best solution. Each replica can exchange the temperature assigned to it with a temperature of a neighbouring replica based on a Metropolis criterion. This method thus enables a global search for an optimum. In addition, the Digital Annealer can automatically calibrate algorithmic convergence parameters while using the parallel tempering approach. This is useful, for example, when no calibration of convergence parameters of a chosen approach is desired for a particular QUBO problem. A more detailed description can be found in the Digital Annealer Tutorial in section M 2 [53].

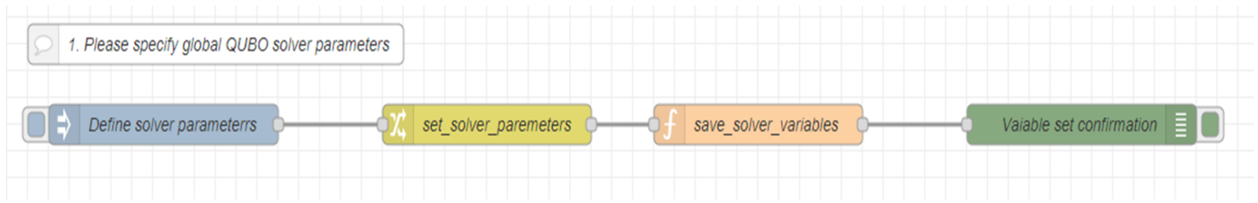

1. Please specify global QUBO solver parameters
Define solver parameterrs
set_solver_paremeters
save_solver_variables
Vaiable set confirmation

Figure 103: Specifying the global solver parameters node-RED flow implementation

After specifying the global parameters, the user specifies the parameters related to the chosen solving method.

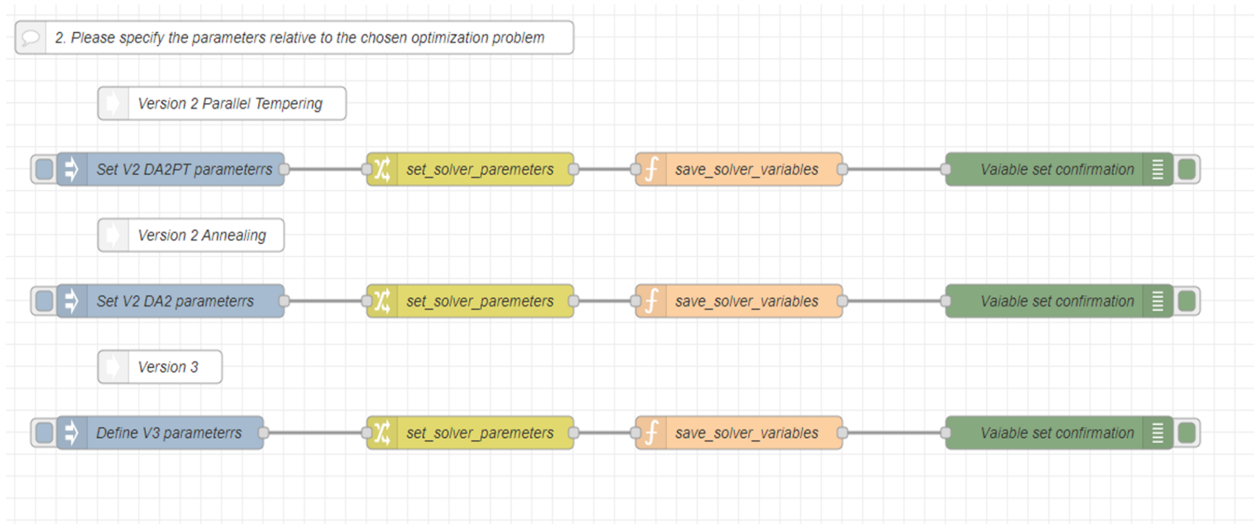


Figure 104: Specifying the parameters relative to the chosen optimization solving method

· Version 2 solving and Parallel Tempering optimization (DA2PTSolver):

When using this API for the first time, specifying DA2PTSolver is recommended since scaling and rounding is enabled and this solver does not require parameter tuning related to a temperature schedule. If it takes a long time to find an optimal solution or an optimal solution cannot be found even if DA2PTSolver is used, consider trying DA2Solver.

- *number_iterations* (integer): The number of searches per anneal. Specify an integer from 1 to 2000000000. (Default: 2000000)

- *number_replicas* (integer): The number of replicas. The number of runs of annealing in parallel at initialized different temperatures. Specify an integer from 26 to 128.

Please be aware, automatic tuning rarely leads to optimal speed or the best results. Which parameters can be changed, are described in the next section.

· Version 2 solving and Annealing optimization:

- *expert_mode* (boolean): Specify either true or false (default: false), depending on whether enabling or disabling Scaling and Rounding and the manual setting function for each parameter (Parameter Setting).

- When "true" is specified:
    - Scaling and Rounding: Disabled
    - Parameter Setting: Enabled
- When "false" is specified:

§ Scaling and Rounding: Enabled

§ Parameter Setting: Disabled

- *number_iterations* (integer): The number of searches per anneal. Specify an integer from 1 to 2000000000. (Default: 2000000)

- *number_runs* (integer): The number of repetitions of annealing. Specify an integer from 16 to 128.

- *offset_increase_rate* (float): The energy offset value is increased sequentially by multiples of offset_increase_rate when there is no candidate for the new state. Specify a value from 0 to 2000000000. Can be specified only when "true" is specified for the expert_mode parameter.

- *temperature_decay* (float): Multiplier for changing the temperature during annealing. The range of possible values depends on the temperature_mode specification. Can be specified only when "true" is specified for the expert_mode parameter.

- *temperature_interval* (integer): Temperature change interval during annealing. Specify an integer equal to or greater than 1. Can be specified only when "true" is specified for the expert_mode parameter.

- *temperature_mode* (string): Annealing temperature change model. Specify one of these values: "EXPONENTIAL," "INVERSE," or "INVERSE_ROOT." (Default: EXPONENTIAL). Specifying "EXPONENTIAL" is recommended. The new temperatures for each temperature change model are calculated as follows (T0=temperature_start):

  o EXPONENTIAL: Tn+1 = Tn × (1 - temperature_decay)

  o INVERSE: Tn+1 = Tn × (1 - temperature_decay × Tn)

  o INVERSE_ROOT: Tn+1 = Tn × (1 - temperature_decay × Tn × Tn)

Specify a value for temperature_decay so that the right side of the equation for each mode is equal to or greater than 0. The ranges of possible values for each mode are:

o EXPONENTIAL: 0 ≦ temperature_decay ＜ 1

o INVERSE: 0 ≦ temperature_decay ＜ 1 / temperature_start

o INVERSE_ROOT: 0 ≦ temperature_decay ＜ 1 / (temperature_start × temperature_start)

If a smaller value is specified for temperature_decay, the annealing temperature change becomes slower. Can be specified only when "true" is specified for the expert_mode parameter.

- *temperture_start* (float): Annealing start temperature. Specify a value greater than 0. Can be specified only when "true" is specified for the expert_mode parameter.

- *solution_mode* (string): Optimal solution return mode. Specify "COMPLETE" or "QUICK." (Default: COMPLETE)

  o When "COMPLETE" is specified: All solutions for the number of annealing executions specified with number_replicas are returned. The same solutions are consolidated into one and returned with the total rate of appearance frequency set for "frequency."

  o When "QUICK" is specified: Among all the solutions for the number of annealing executions specified with number_replicas, only one solution with the lowest energy (optimal solution) is returned.

- Please be aware that the specification of the parameters has a big impact on the solution quality. So please try different values or learn more about them in our Digital Annealer Tutorial Section M 6 and the corresponding examples [53].

· Version 3 solving:

- *time_limit_sec* (integer): Maximum running time of DA in seconds. Specifies the upper limit of running time. The unit is seconds. The calculation is terminated when the running time reaches the upper limit time specified by time_limit_sec. Specifies an integer from 1 to 1800. (Default: 10)

- *target_energy* (double): Threshold energy for fast exit. Specifies the target energy value. If not specified, the calculation will be performed without setting the target energy value. When the minimum energy value reaches the target energy value, the calculation is terminated even if the running time does not reach the upper limit time. Specifies a value from -2126 to 2126. (Default: disabled)

- *num_run* (integer): The number of parallel attempts of each group. Specifies an integer from 1 to 16. (Default: 16)

- *num_group* (integer): The number of groups of parallel attempts. The num_run x num_group specifies the number of parallel attempts. Specifies an integer from 1 to 16. (Default: 1)

● *num_output_solution (integer):* The number of output solutions of each group. num_output_solution x num_group specifies the number of output solutions. Specifies an integer from 1 to 1024. (Default: 5)
● *gs_level* (integer): Level of the global search. In the global search, the search starting point with local solution group escape is determined, and the constrained search combining various search methods is repeatedly executed as a processing unit. The higher the value,

the longer the constraint exploitation search. Specifies the level of the global search. Lower level is weak on Global Search. Specifies an integer from 0 to 100. (Default: 5)

- *gs_cutoff* (integer): Global search cutoff level. Specifies the convergence judgment level for global search constraint usage search. The higher the value, the longer the period during which the constraint-based search energy on which convergence is based is not updated. Convergence assessment is turned off at 0. Specifies an integer from 0 to 1000000. (Default: 8000)
- *penalty_auto_mode* (integer): Coefficient adjustment mode. Specifies the coefficient adjustment mode for constraint terms.
    - o 0: behavior with fixed value specified by penalty_coef
    - o 1: internally autofit with penalty_coef as initial value

Specifies an integer 0 or 1. (Default: 1)

- *penalty_coef* (integer): Coefficient of the constraint term. Specifies the coefficient of the constraint term. Specifies an integer from 1 to 9223372036854775807. (Default: 1)

- *penalty_inc_rate* (integer): Parameters for automatic adjustment of constraint terms. Specifies the parameter for automatic adjustment of the constraint term in the global search. Specifies an integer from 100 to 200. (Default: 150)

- *max_penalty_coef (integer)*: Maximum constraint term coefficient. Specifies the maximum constraint term coefficient. Set to 0 for no maximum value. Specifies an integer from 0 to 9223372036854775807. (Default: 0)

- Please be aware that the specification of the parameters has a big impact on the solution quality. So please try different values or learn more about them in our Digital Annealer Tutorial Section M 6 and the corresponding examples [53].

B. Define the QUBO parameters:

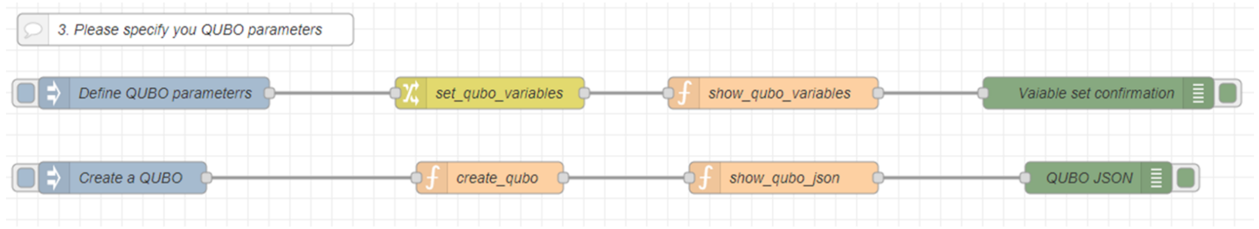
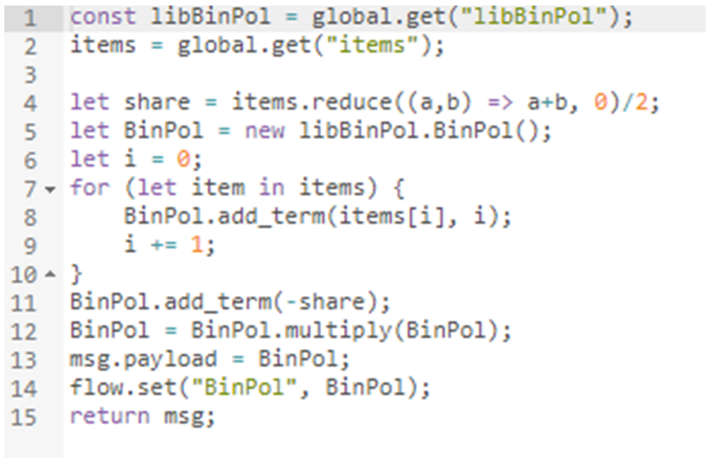


Figure 105: Specifying the QUBO parameters Node-RED implementation

In this step, the user defines the QUBO formulation of the optimization problem in the *create_qubo* function. Figure 106 shows for example the QUBO formulation of the two-persons assignment optimization problem. Using the function *set_qubo_variables*, the model user defines the list of values to split as a list.

Please note that the functions in this step are specific to the two-persons assignment optimization problem. When using the model to solve other QUBOs, the QUBO formulation must be defined in *create_qubo* and the QUBO parameters must be given in a new block *set_variable*.



```
const libBinPol = global.get("libBinPol");
items = global.get("items");

let share = items.reduce((a,b) => a+b, 0)/2;
let BinPol = new libBinPol.BinPol();
let i = 0;
for (let item in items) {
    BinPol.add_term(items[i], i);
    i += 1;
}
BinPol.add_term(-share);
BinPol = BinPol.multiply(BinPol);
msg.payload = BinPol;
flow.set("BinPol", BinPol);
return msg;
```

Figure 106: QUBO formulation of the two-persons assignment in Node-RED

C. Checking solver availability:

Before starting the optimization process, in this step of the flow user checks the availability of the solving device (CPU server, Digital Annealer Service) by sending a health check request. Once the response has been returned and the message "CPU/DA solver available" is displayed, the QUBO solver can be launched.

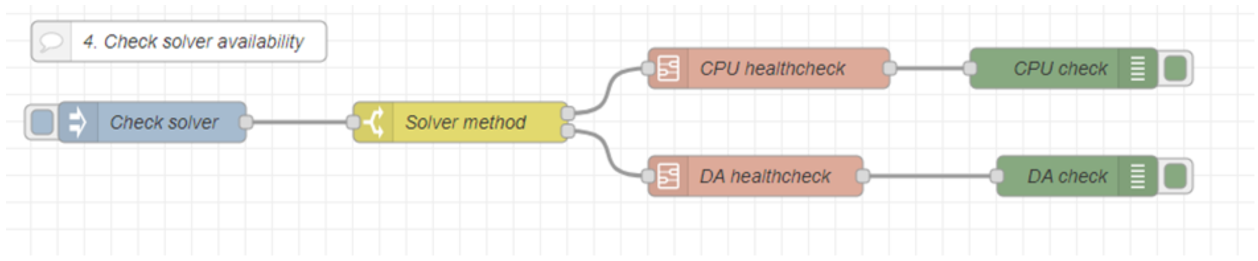


Figure 107: Checking solver availability Node-RED implementation

D. Solve the QUBO and show the results:

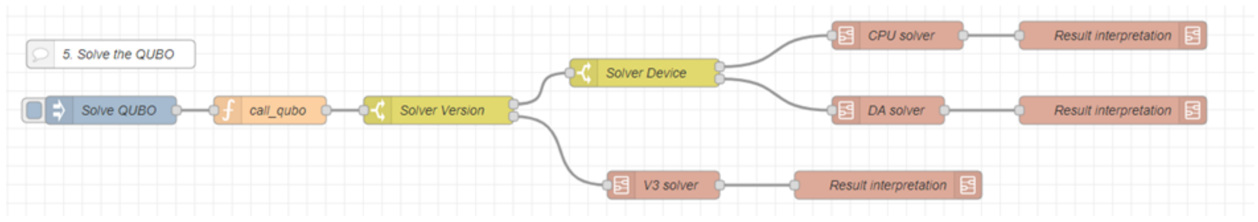
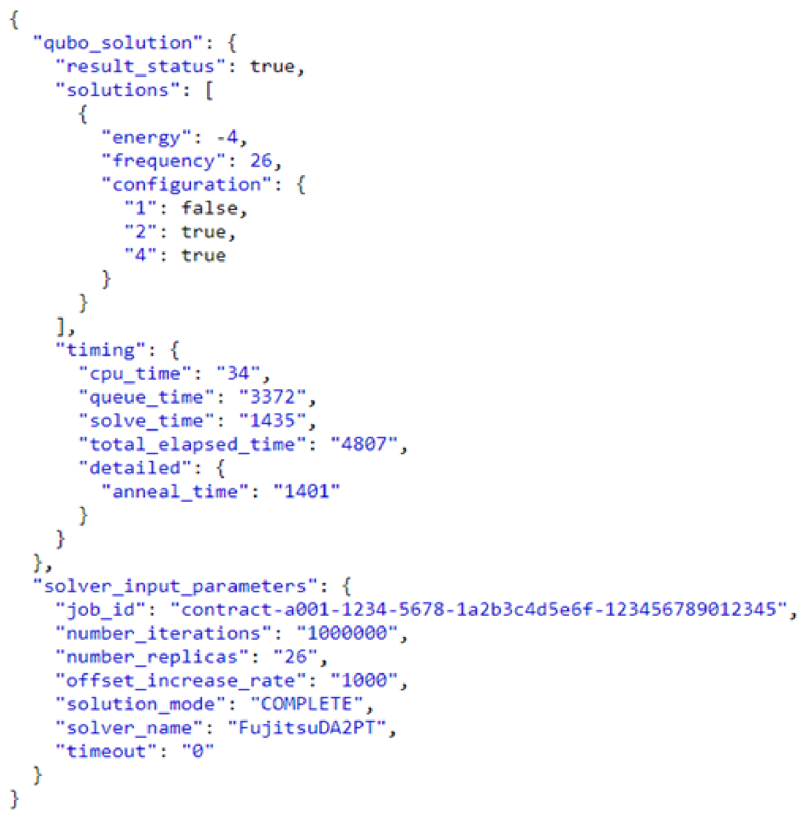


Figure 108: Solving the QUBO Node-RED implementation

During this step, the QUBO is solved using the annealing approach. The solution is returned as a JSON file (see Figure 109). The QUBO response is described as follow:

- result_status (boolean): Processing result status (true or false).
- solutions: Array of QuboSolutions
  - energy: Energy of the optimal solution
  - frequency: Appearance frequency of the optimal solutions with the same configuration
  - configuration: Value for each variable x (true or false)
- timing: List of timings:
  - cpu_time: Time for using CPU (Unit: millisecond)
  - queue_time: Waiting time for processing (Unit: millisecond)
  - solve_time: Time for processing by solver (Unit: millisecond)

    (The wait time for other processes is also included.)
  - total_elapsed_time: Total time required to find an optimal solution (Unit: millisecond)
  - anneal_time: Time for processing with the Digital Annealer hardware (Unit: millisecond)



```
{
  "qubo_solution": {
    "result_status": true,
    "solutions": [
      {
        "energy": -4,
        "frequency": 26,
        "configuration": {
          "1": false,
          "2": true,
          "4": true
        }
      }
    ],
    "timing": {
      "cpu_time": "34",
      "queue_time": "3372",
      "solve_time": "1435",
      "total_elapsed_time": "4807",
      "detailed": {
        "anneal_time": "1401"
      }
    }
  },
  "solver_input_parameters": {
    "job_id": "contract-a001-1234-5678-1a2b3c4d5e6f-123456789012345",
    "number_iterations": "1000000",
    "number_replicas": "26",
    "offset_increase_rate": "1000",
    "solution_mode": "COMPLETE",
    "solver_name": "FujitsuDA2PT",
    "timeout": "0"
  }
}
```

Figure 109: Response Body of a QUBO solution

The interpretation of the results is done via the sub-flow "Result interpretation". For example (two-person assignment), the solution would be two lists showing the items assigned to each person (person A and person B).

4.10.4. Pattern limitations

- The optimization is done by simulated annealing and therefore a near-by approximation of the optimized solution.
- To use the Digital Annealer solution method, an API key is required to access the DA cloud service (please contact Fujitsu DA team to get credentials). The CPU is an alternative free emulator, but only for small QUBOs.
- The CPU emulator could only solve QUBO with less than 1024 bits. Version 2 of DA supports QUBO's with up to 8192 bits. The Digital Annealer System (Version 3) supports QUBO's with up to 100,000 bits.

4.10.5. Pattern experimentation

To test the performance of DA on solving the two-person assignment optimization problem, we generated a list of 100 random values from 10 to 50 and solved the QUBO using the different solve parameters. The results which are presented in Table 19 shows the average running time of repeating the previous procedure 10 times.

Table 19: Average total solving time for the two-persons assignment

Solver Version	Solver Method	Solver Device	Average total solving time (ms)	Average annel_time (ms)
V2	DA2PT	CPU	48,37	-
		DA	298,65	120,71
	DA2	CPU	43,23	-
		DA	186,1	13,5
V3	-	-	10561,4	

4.10.6. Advanced customization of Digital Annealer Optimizer Pattern

In this section, we investigate how to more customise the Digital Annealer (DA). The customization is done through extending the JavaScript classes who are responsible to glue Node-Red with DA. The classes are designed to provide support for definition of a QUBO problem through binary polynomials, configuration of DA, solving a QUBO problem on DA, and processing the returned solutions form DA. In this section we provide the implementation details of these classes.

A- BinPol class

This class is used to form a binary polynomial which is used to define a QUBO problem. The instances of this class will be sent to DA for finding its optimum solution. Figure 110 provides the overview of the BinPol class.

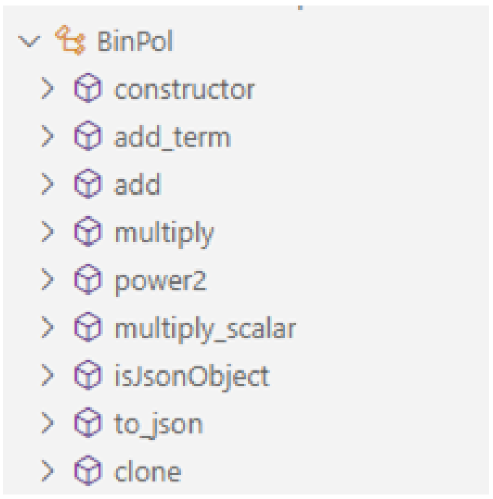


Figure 110: Overview of the BinPol class

The BinPol offers few methods to construct a binary polynomial (BP).

- Method add_term() is used to add terms to an existing BinPol object.
- Methods add() and multiply() are respectively used to add and multiply two BPs.
- Method power2() raises the current BP to the power of two.
- Method multiply_scalar() is used to multiply the current BP with a given scalar.
- Method clone() clones a new BP from the current BP.
- Method to_json() exports the current BP to a JSON form that can be processed by the solver class.

B- Configuration classes

The classes which are provided as the category for configuration are classes which are used to configure the interactions with DA, both for version 2 and version 3.
Figure 111 shows the existing classes for configurations.

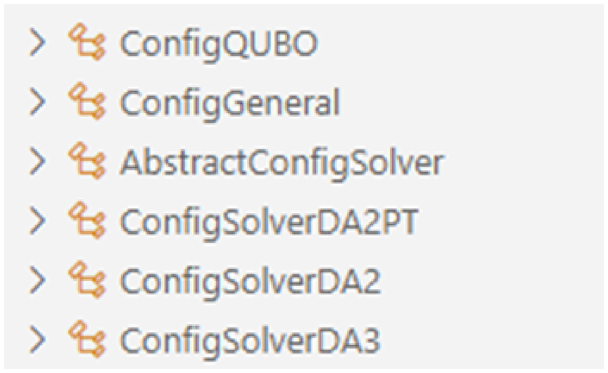


Figure 111: Overview of classes for Configuration of DA

Here we provide a brief introduction of the implemented classes and their usage. In the coming subsections we will go deeper into each class.

- Class *ConfigQUBO* is the abstract base class for all other classes in the configuration module.
- Class *ConfigGeneral* is the base class for general configuration of DA (e.g. configuration of URL of solvers, credentials keys, etc.).
- Class *AbstractConfigSolver* is the abstract class which is the base class for specific configuration for ver.2 and ver.3 of DA.
- Class *ConfigSolverDA2PT* provides the functionalities for Parallel Tempering solver-mode for version 2 of DA.
- Class *ConfigSolverDA2* provides the functionalities for version 2 of DA (no Parallel Tempering).
- Class *ConfigSolverDA3* provides the functionalities for version 3 of DA.

Figure 112 shows the UML diagram which depicts the relationship between the above classes.

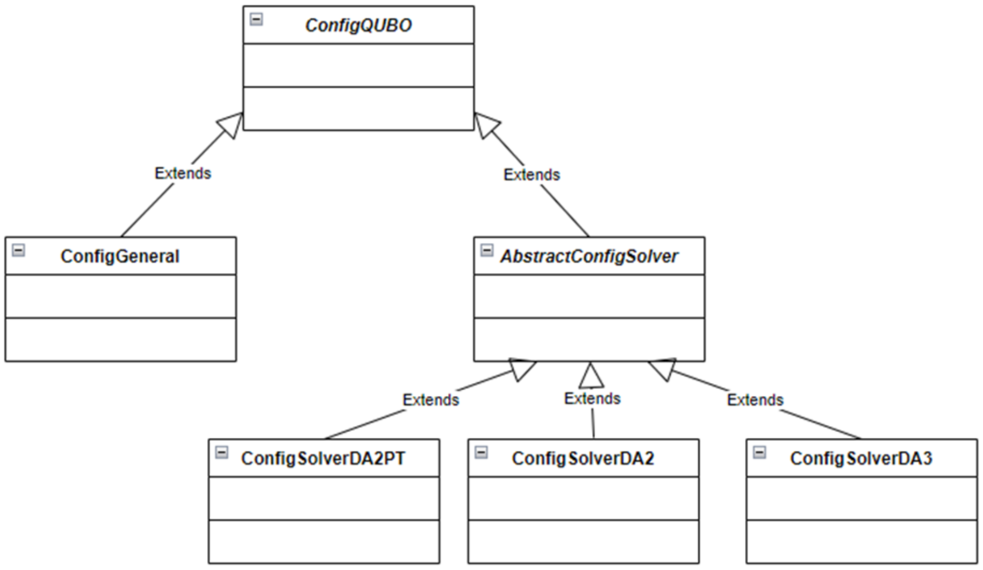


Figure 112: UML diagram for classes in the configuration module

B.1. ConfigQUBO class

This is an abstract class which is the base of all other classes for configuration. Figure 113 shows the overview of the class.

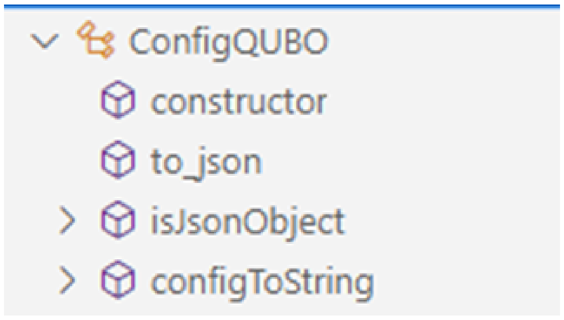


Figure 113: Abstract base class of all configuration

- Abstract method *to_json()* converts a configuration class to a JSON representation which is used for communication with the DA server.
- Abstract methods of *configToString()* represent the string representation of the JSON object of the class.

The above methods provide the same functionalities in all sub-classes.

B.2. ConfigGeneral class

The ConfigGeneral class provides the base functionalities corresponding with general aspects of communication with DA service. Figure 114 shows the overall view of the class.

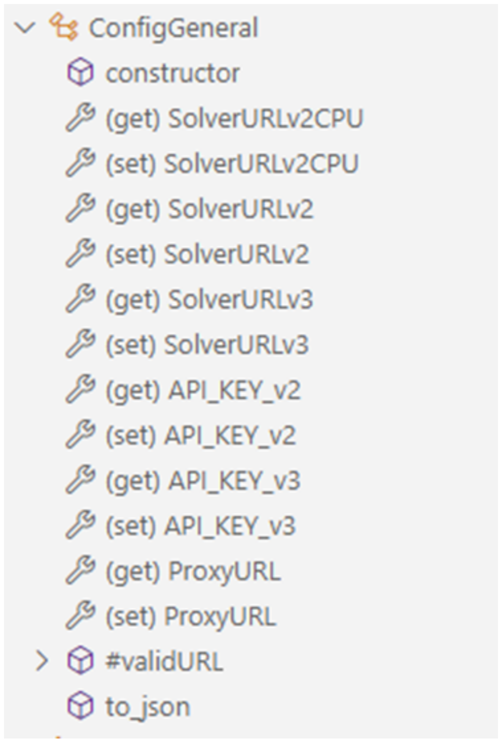


Figure 114: Overall view of the ConfigGeneral class

The class provides configuration for URLs of the DA service as well as the API KEYS which are required to authenticate and use the service. More concretely the following properties are provided (can be set/get as well):

- SolverURLv2CPU: specifies the URL of the CPU solver for DA.
- SolverURLv2: specifies the URL for solver service of version 2 of DA.
- SolverURLv3: specifies the URL for solver service of version 2 of DA.
- API_KEY_v2: is the key for authentication and usage of solver service for the version 2 of DA.
- API_KEY_v3: is the key for authentication and usage of solver service for the version 3 of DA.
- ProxyURL: specifies the URL for a proxy service. This is used when the docker of Node-Red should communicate with solver service through a proxy.

B.3. AbstractConfigSolver class

This is an abstract class for version 2 and version 3 of DA solvers and base of other classes.

B.4. ConfigSolverDA2PT class

ConfigSolverDA2PT is used for configuration of the Parallel Tempering mode of the DA version 2. It extends the AbstractConfigSolver class. Figure 115 shows the overview of the class.

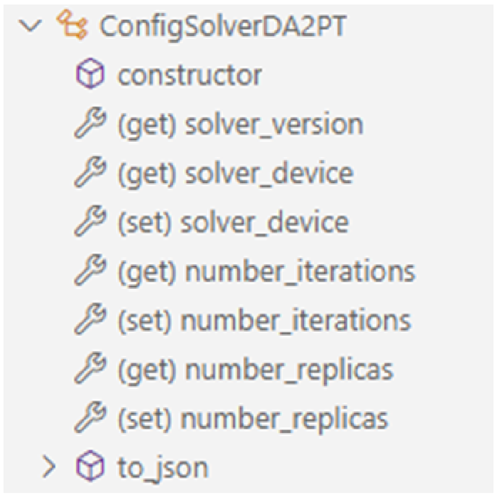

ConfigSolverDA2PT
constructor
(get) solver_version
(get) solver_device
(set) solver_device
(get) number_iterations
(set) number_iterations
(get) number_replicas
(set) number_replicas
to_json

Figure 115: Overview of the ConfigSolverDA2PT class

The class provides the following properties (which can be set/get):

- solver_version: returns the version of DA solver i.e. “V2”
- solver_device: specifies if the solver should be “CPU” or “DA”.
  - Valid values: “CPU” | “DA”
- number_iterations: specifies the number of iterations used for the solver.
  - Default value: 1000
  - Valid range: [1, 2*10^9]
- number_replicas: specifes the number of replicas used for the solver:
  - Default value: 26
  - Valid range: [26, 128]

B.5. ConfigSolverDA2 class

ConfigSolverDA2 is used for configuration of the DA version 2. It extends the AbstractConfigSolver class. Figure 116 shows the overview of the class.

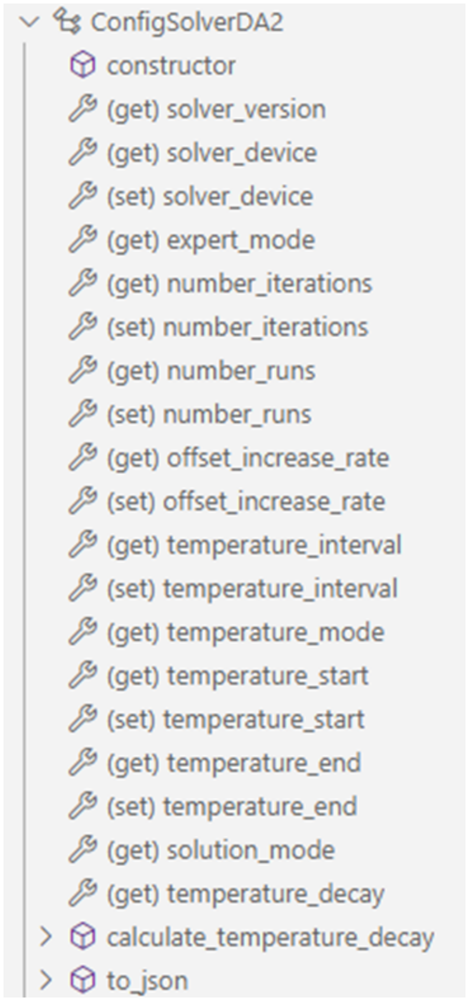

ConfigSolverDA2
constructor
(get) solver_version
(get) solver_device
(set) solver_device
(get) expert_mode
(get) number_iterations
(set) number_iterations
(get) number_runs
(set) number_runs
(get) offset_increase_rate
(set) offset_increase_rate
(get) temperature_interval
(set) temperature_interval
(get) temperature_mode
(get) temperature_start
(set) temperature_start
(get) temperature_end
(set) temperature_end
(get) solution_mode
(get) temperature_decay
calculate_temperature_decay
to_json

Figure 116: Overview of the ConfigSolverDA2 class

The class provide the following properties (which can be set/get):

- solver_version: returns the version of DA solver i.e. "V2"
- solver_device: specifies if the solver should be "CPU" or "DA".
    - Valid values: "CPU" | "DA"
- number_iterations: specifies the number of iterations used for the solver.
    - Default value: 1000
    - Valid range: [1, 2*10^9]
- number_runs: specifies the number of runs used for the solver:
    - Default value: 16

    - Valid range: [16, 128]

- offset_increase_rate: specifies the offset_increase_rate of DA
    - Default value: 25
    - Valid range: [0, 2*10^9]
- temperature_interval: specifies the temperature_interval of DA.
    - Default value: 100
    - Valid range: [16,128]
- temperature_mode: returns the temperature_mode of DA, in this implementation "EXPONENTIAL".
- temperature_start: specifies the temperature_start of DA
    - Default value: 1000.0
    - Valid range: [0.0, inf)
- temperature_end: specifies the temperature_end of DA
    - Default value: 1.0
    - Valid range: [0.0. inf)
- solution_mode: returns the solution_mode of DA, in this implementation "COMPLETE".
- temperature_decay: returns the calculated temperature_decay of DA solver. The calculation is done through the calculate_temperature_decay() method.
- calculate_temperature_decay(): these methods calculate the temperature_decay of the DA.

B.6. ConfigSolverDA3 class

ConfigSolverDA3 is used for configuration of the DA version 3. It extends the AbstractConfigSolver class. Figure 117 shows the overview of the class.

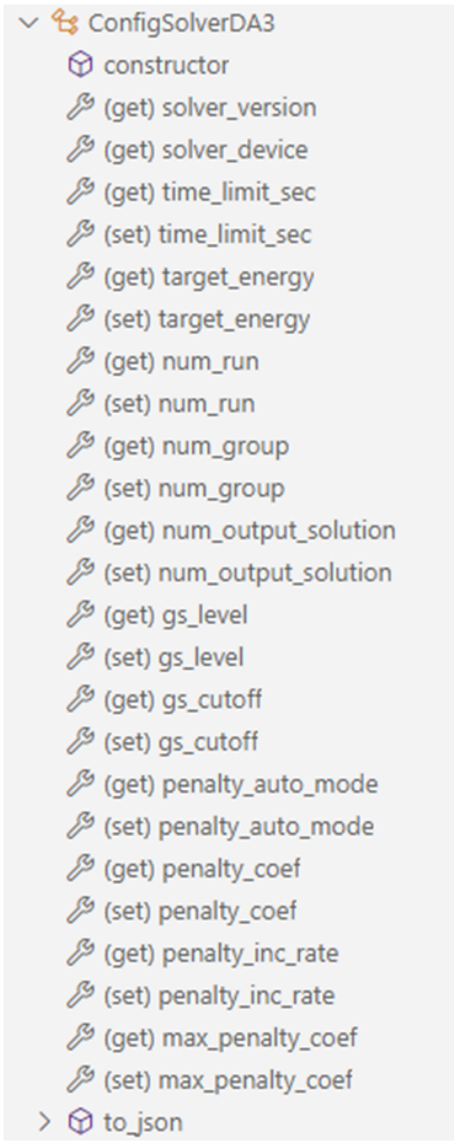

ConfigSolverDA3
constructor
(get) solver_version
(get) solver_device
(get) time_limit_sec
(set) time_limit_sec
(get) target_energy
(set) target_energy
(get) num_run
(set) num_run
(get) num_group
(set) num_group
(get) num_output_solution
(set) num_output_solution
(get) gs_level
(set) gs_level
(get) gs_cutoff
(set) gs_cutoff
(get) penalty_auto_mode
(set) penalty_auto_mode
(get) penalty_coef
(set) penalty_coef
(get) penalty_inc_rate
(set) penalty_inc_rate
(get) max_penalty_coef
(set) max_penalty_coef
to_json

Figure 117: Overview of the ConfigSolverDA3 class

The class provide the following properties (which can be set/get):

- solver_version: returns the version of DA solver i.e. “V3”
- solver_device: returns the solver device for version 3, i.e. “DA”
- time_limit_sec: specifies the time limit of DA solver in seconds.
    - Default value: 10
    - Valid range: [1,1800]
- number_runs: specifes the number of runs used for the solver:

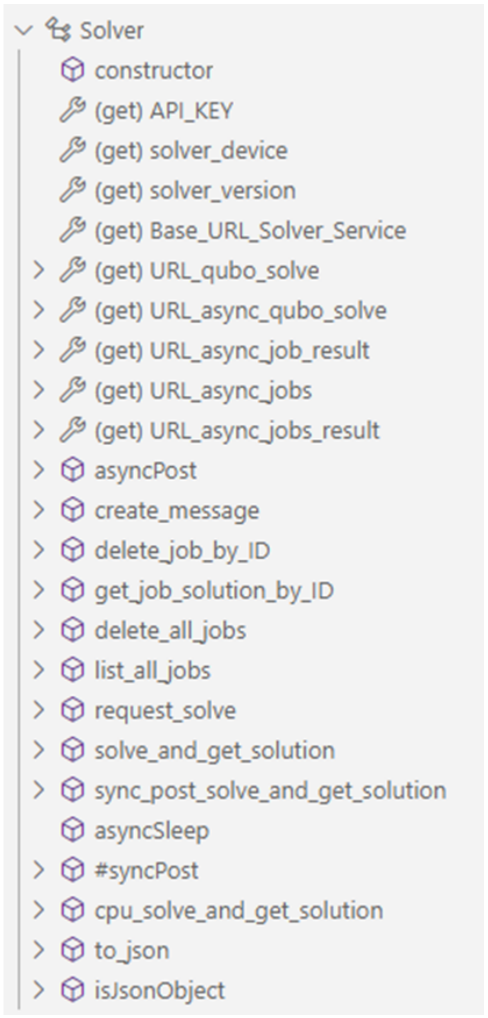
  - o Default value: 16
  - o Valid range: [16, 128]

- target_energy: specifies the target_energy of DA solver.
- num_run: specifies the num_run of DA solver.
  - o Default value: 16
  - o Valid range: [1,16]
- num_group: specifies the num_group of DA solver
  - o Default value: 1
  - o Valid range: [1,16]
- num_output_solution: specifies the num_output_solution of DA solver.
  - o Default value: 5
  - o Valid range: [1, 1204]
- gs_level: specifies the gs_level of DA solver
  - o Default value: 5
  - o Valid range: [0,100]
- gs_cutoff: specifies the gs_cutoff of DA solver
  - o Default value: 8000
  - o Valid range: [0, 1000000]
- penalty_auto_mode: specifies the penalty_auto_mode of DA solver
  - o Default value: 1
  - o Valid range: [0,1]
- penalty_coeff: specifies the penalty_coeff of DA solver
  - o Default value: 1
  - o Valid range: [1, 9223372036854775807]

- penalty_inc_rate: specifies the penalty_inc_rate of DA solver
    - Default value: 150
    - Valid range: [100,200]
- max_penalty_coef: specifies the max_penalty_coef of DA solver
    - Default value: 0
    - Valid range: [1, 9223372036854775807]

C- Solver Class

The solver class is responsible for managing solve requests to DA and getting solution results. Figure 118 shows the overview of the class.



```
Solver
    constructor
    (get) API_KEY
    (get) solver_device
    (get) solver_version
    (get) Base_URL_Solver_Service
  > (get) URL_qubo_solve
  > (get) URL_async_qubo_solve
  > (get) URL_async_job_result
  > (get) URL_async_jobs
  > (get) URL_async_jobs_result
  > asyncPost
  > create_message
  > delete_job_by_ID
  > get_job_solution_by_ID
  > delete_all_jobs
  > list_all_jobs
  > request_solve
  > solve_and_get_solution
  > sync_post_solve_and_get_solution
    asyncSleep
  > #syncPost
  > cpu_solve_and_get_solution
  > to_json
  > isJsonObject
```

Figure 118: Overview of the Solver class

The class provides the following functionalities and properties.

- The constructor of the class takes two configuration objects of solverConfig and generalConfig as follows :
    - o constructor (solverConfig, generalConfig)
        - § The solverConfig is an instance of the one of sub-classes of the abstract class AbostractConfigSolver i.e., an instance of ConfigSolverDA2PT, ConfigSolverDA2 or ConfigSolverDA3.
        - § The generalConfig is an instance of GeneralConfig class.
    - o The Solver class uses these to configuration to properly communicate with the correct version of DA solver service.
- API_KEY: returns the key associated with the generalConfig.
- solver_device: returns the associated device for solver i.e. “CPU” or “DA”.
- Base_URL_Solver_Service: is the base part of the URL which provides the solver service.
- URL_*:
    - o These properties provide the common URLs for solving, querying and getting solutions on the DA service.
- async asyncPost(messageBody, urlStr)
    - o Gets a messageBody (string) and asynchronously post it the give URL and returns the result as a JSON object.
    - o This method is asynchronous.
- create_message(messageMethod, messageBody)
    - o This method creates a message with a given messageBody
    - o The messageMethod can be one of "POST", "GET", "DELETE".
    - o The created message will be of given type accordingly.
- async delete_job_by_ID(jobID)
    - o This method requests the DA service for deletion of an already posted job by given jobID.

  - It returns a JSON object of the deletion results or raise an exception if an error occurs.
  - This method is asynchronous.

- async get_job_solution_by_ID(jobID)
  - gets the solution of an already posted job by given jobID
  - it returns a JSON object of the solution or raise an exception if an error occurs.
  - This method is asynchronous.
- async delete_all_jobs()
  - This method deletes all posted jobs on DA service.
  - This method is asynchronous.
- async list_all_jobs()
  - This method lists all jobs existing on DA service.
  - This method is asynchronous.
- async request_solve(binPol)
  - This method requests the DA for solving the given binPol.
  - This method returns a JSON object which contains the jobID of the solve request.
  - This method is asynchronous.
- async solve_and_get_solution(binPol)
  - This method requests the DA to solve the given binPol and waits till the solution is available.
  - This method returns a JSON object containing the solution.
  - This method is asynchronous.
- sync_post_solve_and_get_solution(binPol)
  - This a synchronous method for requesting the DA to solve the given bibPol and getting the solution.

- cpu_solve_and_get_solution(binPol)
    - This is a method which requests the solution of the given binPol on CPU.
- to_json(binPol)
    - This method provides the JSON object of the solver with a given binPol.
    - This method is used in the other methods which request DA for solving a binPol.

D- Solution Class

This class is used to work with the JSON object of the solution returned by the DA service more efficiently. Figure 119 provides an overview of the class.

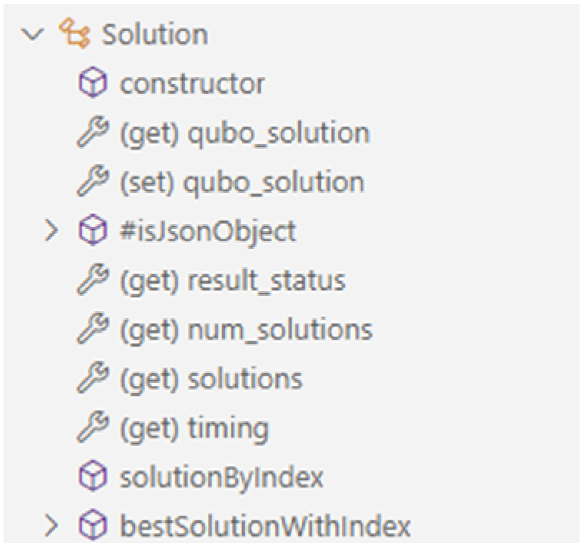


Figure 119: Overview of the Solution class

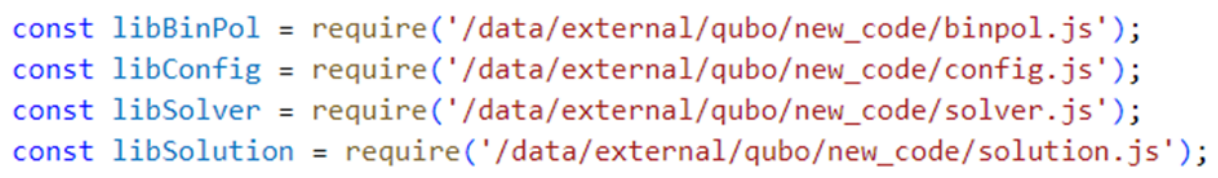
The class provides the following functionalities/properties.

- constructor(solutionJson)
    - The constructor of the class gets the solution JSON object returned from DA service.
- qubo_solution:
    - Returns the value of the “qubo_solution” key of the JSON object.
    - This property can also be set if the value of “qubo_solution” key is already provided by other mean than within a JSON object.
- result_status: returns the value of the “result_status” key in a qubo_solution
- num_solutions:
    - Returns the total number of solutions returned by DA (the solution of a QUBO problem might not be unique)
- solutions: returns the value of the “solutions” key within the qubo_solution
    - This effectively gets a list of all solutions returned by DA.

- timing: returns the value of the “timing” key within the qubo_solution
    - This effectively returns the time sent by DA to compute the solution.
- solutionByIndex(index)
    - This returns the solution at the index.
    - The value of index should be within [0, num_solutions)
- bestSolutionWithIndex()
    - This returns an array of the [bestSolution, bestSolutionIndex]
    - The bestSolution is the solution with the minimum energy from all the solutions returned by DA.
    - If more than one bestSolution exists, the first one is returned.

E- How to use the JavaScript classes

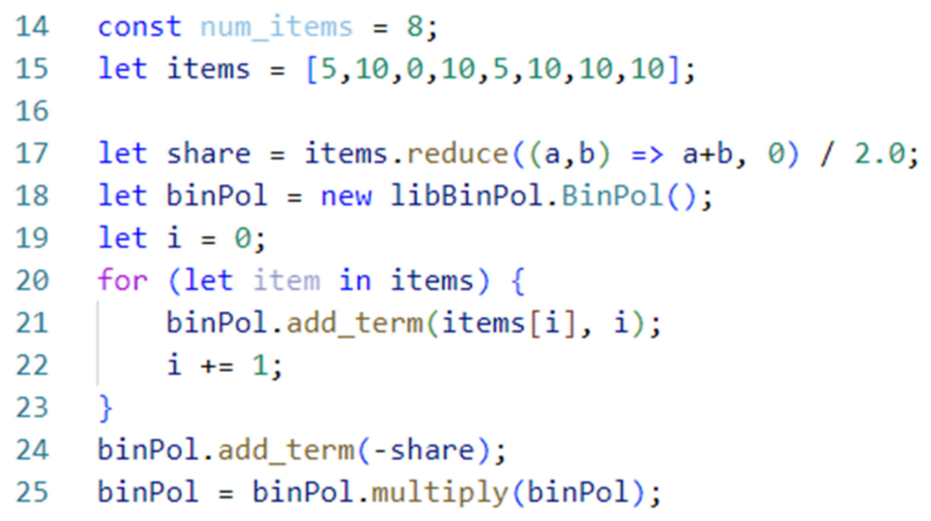
In this section we give an overview of how to use the JavaScript classes described in the previous sections. The first step is to load the previous modules and classes. This is done through the following code snippet (Figure 120):



```
const libBinPol = require('/data/external/qubo/new_code/binpol.js');
const libConfig = require('/data/external/qubo/new_code/config.js');
const libSolver = require('/data/external/qubo/new_code/solver.js');
const libSolution = require('/data/external/qubo/new_code/solution.js');
```

Figure 120: Loading of Requirements for Javascript Classes

Typically, the next step is to define your binary polynomial which you would like to solve. This is done through the using the BinPol class from its associated module. The following code snippet (Figure 121) shows how to define the two-person example discussed earlier:

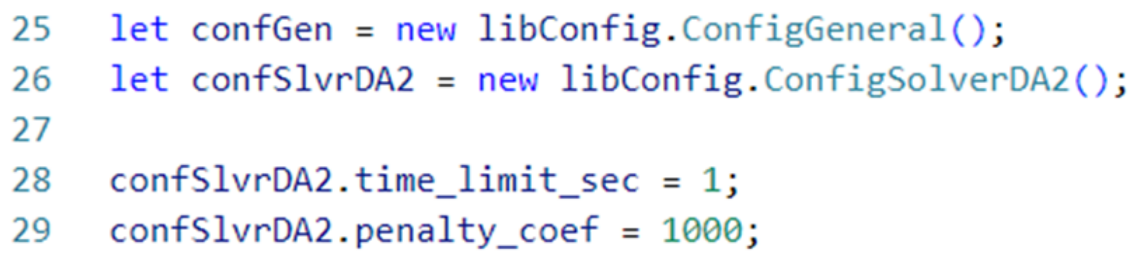


```
const num_items = 8;
let items = [5,10,0,10,5,10,10,10];

let share = items.reduce((a,b) => a+b, 0) / 2.0;
let binPol = new libBinPol.BinPol();
let i = 0;
for (let item in items) {
    binPol.add_term(items[i], i);
    i += 1;
}
binPol.add_term(-share);
binPol = binPol.multiply(binPol);
```

Figure 121: Problem Definition for the two-person Assignment

In the above snippet, what happens is that you need to add proper terms of your QUBO formulation to the binPol object. This is done typically through methods of *add_term(), add(), multipliy()* etc. After defining your binPol object you need to properly instantiate the configuration classes bases on your targeted DA service, i.e., version 2 (CPU, DA, Parallel Tempering) or version 3. In the following code snippet, we instantiate the ver.2 of DA using ConfigSolverDA2 class:

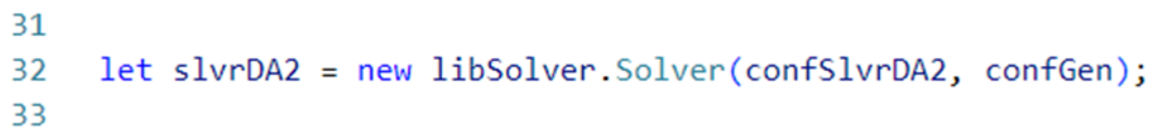


```
25   let confGen = new libConfig.ConfigGeneral();
26   let confSlvrDA2 = new libConfig.ConfigSolverDA2();
27
28   confSlvrDA2.time_limit_sec = 1;
29   confSlvrDA2.penalty_coef = 1000;
```

Figure 122: Addition of Configuration Information for QUBO Problem

In line 25, we define a general configuration object “confGen” and in line 26, we define the “confSlvrDA2” for version 2 of DA solver. We may also manipulate and adjust the default properties of solver (lines 28,29).

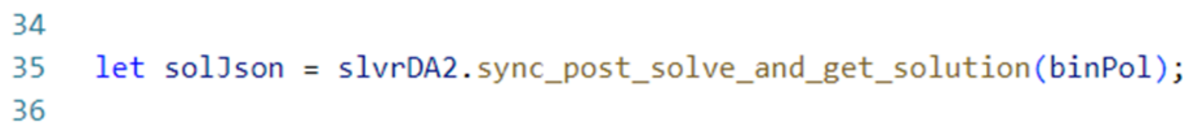
Having defined the configuration objects (Figure 123), we can now define a solver object which properly handles the solve requests based on the chosen configuration:



```
let slvrDA2 = new libSolver.Solver(confSlvrDA2, confGen);
```

Figure 123: Definition of Solver Object

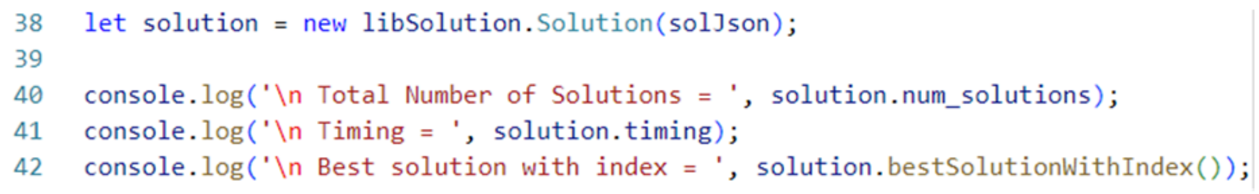
Now we can post a (synchronous) solve request for our binPol object as follows:



```
let solJson = slvrDA2.sync_post_solve_and_get_solution(binPol);
```

Figure 124: Retrieval of Result from Solver

DA returns a JSON object containing the result. We can now use the Solution class to properly handle the result:

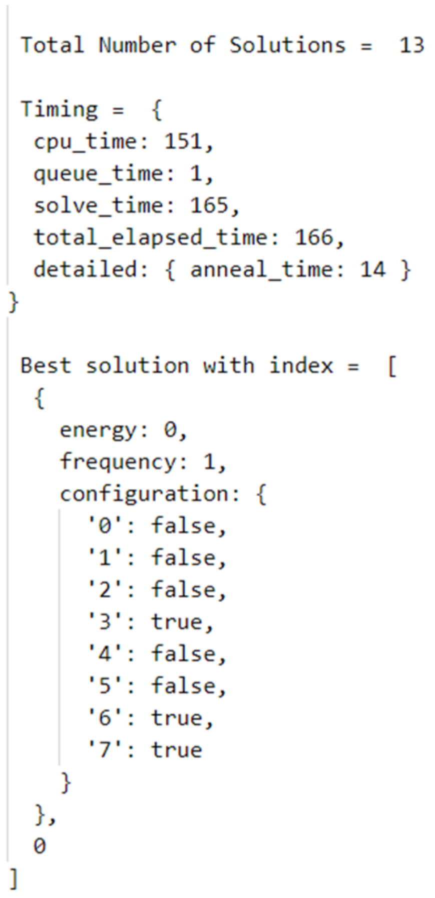


```
let solution = new libSolution.Solution(solJson);

console.log('\n Total Number of Solutions = ', solution.num_solutions);
console.log('\n Timing = ', solution.timing);
console.log('\n Best solution with index = ', solution.bestSolutionWithIndex());
```

Figure 125: Result Post-processing for Solver Output

The output of the above snippet is given below:



```
 Total Number of Solutions =  13

 Timing =  {
  cpu_time: 151,
  queue_time: 1,
  solve_time: 165,
  total_elapsed_time: 166,
  detailed: { anneal_time: 14 }
}

 Best solution with index =  [
  {
    energy: 0,
    frequency: 1,
    configuration: {
      '0': false,
      '1': false,
      '2': false,
      '3': true,
      '4': false,
      '5': false,
      '6': true,
      '7': true
    }
  },
  0
]
```

Figure 126: Final Output of Solver Results

Note that we can principally use the same approach for other versions of the solver configuration classes.

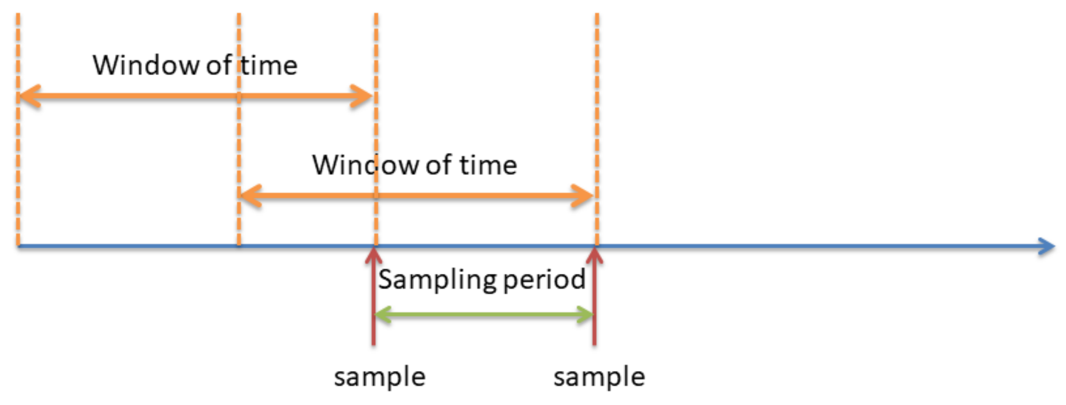
4.11. Openwhisk Sliding Window Action Monitor

4.11.1. Pattern template description

The template description for the Sliding Window Action Monitor Pattern appears in Table 20.

Table 20: Template Description for the Sliding Window Action Monitor Pattern

Pattern name	Sliding Window Action Monitor
Relation to Requirements	Load Monitoring, Performance

Source of pattern	Logging needs of the UCs, Management needs of the application and platform
Pattern Context	Execution in a FaaS cluster is subject to a number of different QoS features.
Pattern Underlying Problem	To monitor and document the performance of a FaaS cluster, covering a variety of different metrics (wait time, init time, execution time etc). Furthermore, logging at the level of the application is not always straightforward when it comes to failures or other errors during function execution.
Pattern Usefulness/ Objective	The pattern allows the developer or platform to monitor the condition of a cluster for a dynamically set time window in the past, for one specific or all executed functions in that interval. Furthermore it covers error reporting from function logs, in order to have concentrated and easily extracted information. Based on this, various criteria can be applied in order to further automate management processes, either at the application level or at the platform level.
Schema	 Window of time Window of time Sampling period sample sample
Prerequisites	Location of the Openwhisk cluster as well as credentials for the user namespace that needs monitoring
Condition of application	General monitoring needs
Input parameters	Openwhisk details as well as window of time to be included and sampling period, action name (optional) for monitoring
Output parameters	Average of various reporting statistics Errors from function logs
Included functionality	Retrieval of latest activations, calculation of relevant metrics, extraction of main error log information
Pattern limitations	The subflow uses a flow variable in order to control starting and stopping, hence only one OW monitor should be used in each flow.

Linked to Pattern	OW Skeleton with Error Catch node, so that inner function errors are retrieved in detail
Indicative Domains of Applicability	Application Level Performance, Platform level management

4.11.2. Pattern implementation details

This is a subflow node in order to monitor the latest performance (sliding window) of Openwhisk actions. The flow pings periodically (based on a set polling period) the target Openwhisk installation in order to retrieve the last executed actions (based on the time window parameter) and extract statistics from their execution (Figure 127).

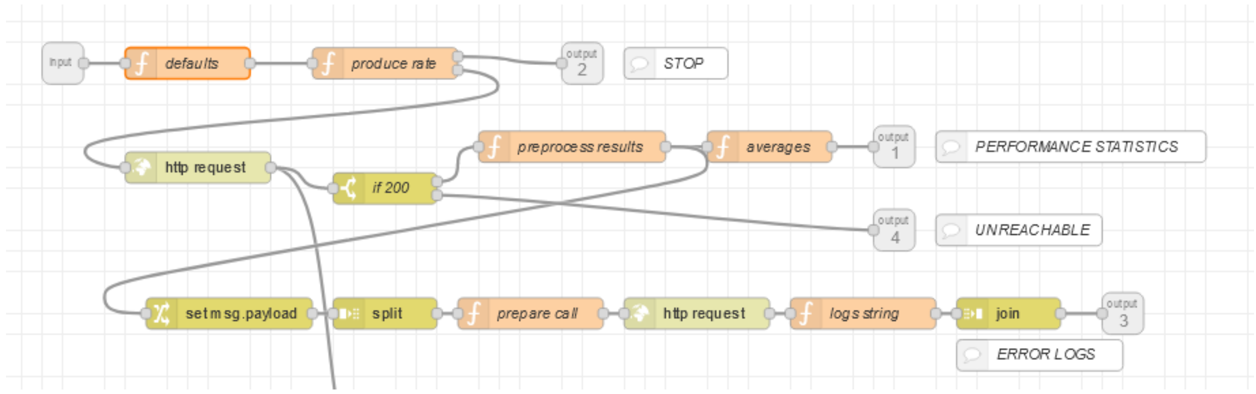
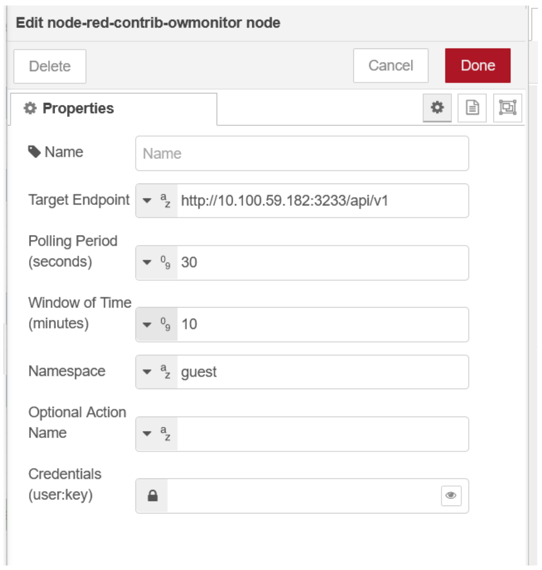


Figure 127: Implementation Flow for the OW Monitor

The configuration (Figure 128) can be applied either via the incoming message fields or via the node UI.



Edit node-red-contrib-owmonitor node

Delete | Cancel | Done

Properties

Name	Name
Target Endpoint	http://10.100.59.182:3233/api/v1
Polling Period (seconds)	30
Window of Time (minutes)	10
Namespace	guest
Optional Action Name	
Credentials (user:key)	

Figure 128: Configuration UI of the OW Monitor

Configuration includes:

- the polling time of the monitor (default 30 seconds, msg.pollingPeriod)
- the target Openwhisk endpoint (msg.targetEndpoint)
- credentials for that endpoint (msg.creds) in the form of user:key
- the window of time (in minutes) in the past for which you want to retrieve activation results (msg.window)
- an optional action name (msg.action), if one wants to filter specific action activations and monitor only them

The monitor can be stopped via a msg.stop=true message. The subflow uses a flow variable in order to control starting and stopping, hence only one OW monitor should be used in each flow.

4.11.3. Pattern examples of usage

An indicative usage flow appears in Figure 129. It can be started and stopped via the relevant triggers.

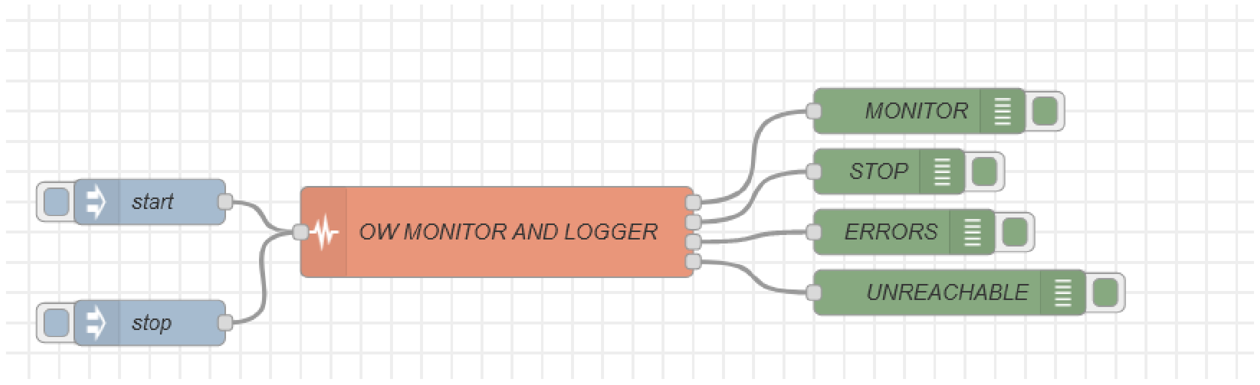
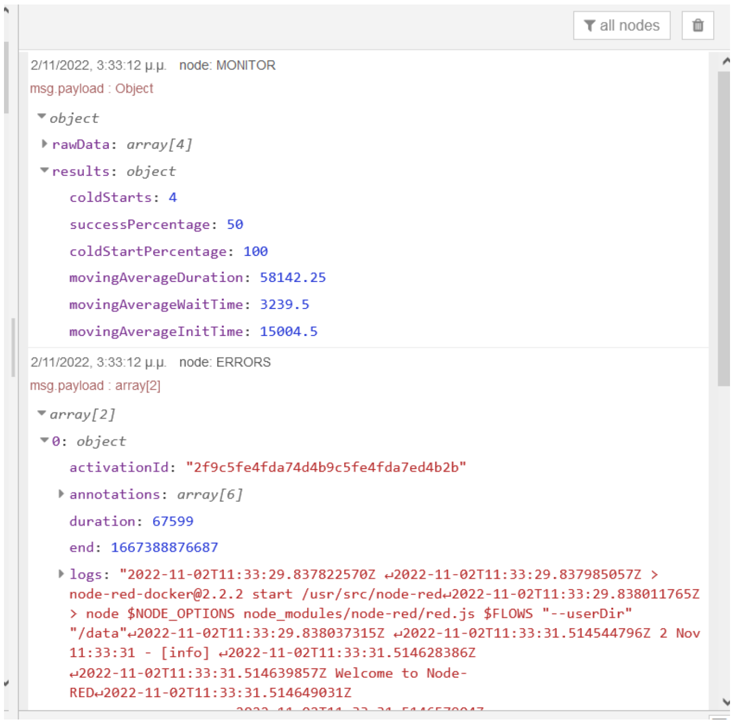


Figure 129: Example of OW Monitor Usage

The outputs are 4:

- output 1-MONITOR is the normal performance report
- output 2-STOP is the notification of the STOP operation
- output 3-ERRORS is used for filtering ERRORS and according logs in the executed functions
- output 4-UNREACHABLE is used when the Openwhisk endpoint is unreachable, either due to msg.statusCode>200 value in the Openwhisk API call for getting the activation or error cases such as ECONNREFUSED or ENOTFOUND

The results in Output 1 include the raw data acquired (msg.payload.rawData) in the last window of time, as well as the moving averages of duration (msg.payload.results.movingAverageDuration), init time (msg.payload.results.movingAverageInitTime) and wait time (msg.payload.results.movingAverageWaitTime) of the window. The number of cold starts are also included (msg.payload.results.coldStarts) as well as the percentage with relation to the total calls (msg.payload.results.coldStartPercentage). If no activations are found, then the averages return NaN values. Example outputs are portrayed in Figure 130, in which the overall performance report is given, as well as the detailed logs of the executions that errored in the last observed window of time.



```
2/11/2022, 3:33:12 μ.μ.   node: MONITOR
msg.payload : Object
▾object
 ▸rawData: array[4]
 ▾results: object
    coldStarts: 4
    successPercentage: 50
    coldStartPercentage: 100
    movingAverageDuration: 58142.25
    movingAverageWaitTime: 3239.5
    movingAverageInitTime: 15004.5
2/11/2022, 3:33:12 μ.μ.   node: ERRORS
msg.payload : array[2]
▾array[2]
 ▾0: object
    activationId: "2f9c5fe4fda74d4b9c5fe4fda7ed4b2b"
   ▸annotations: array[6]
    duration: 67599
    end: 1667388876687
   ▸logs: "2022-11-02T11:33:29.837822570Z ↵2022-11-02T11:33:29.837985057Z >
    node-red-docker@2.2.2 start /usr/src/node-red↵2022-11-02T11:33:29.838011765Z
    > node $NODE_OPTIONS node_modules/node-red/red.js $FLOWS "--userDir"
    "/data"↵2022-11-02T11:33:29.838037315Z ↵2022-11-02T11:33:31.514544796Z 2 Nov
    11:33:31 - [info] ↵2022-11-02T11:33:31.514628386Z
    ↵2022-11-02T11:33:31.514639857Z Welcome to Node-
    RED↵2022-11-02T11:33:31.514649031Z
```

Figure 130: Output of the OW Monitor Process

Avoiding Spam of Error Notifications/Reports

The OW monitor can be used to detect errors in functions,which for example can be forwarded to the developer. However one needs to consider how the monitor works based on the rationale of a time window. So if this time window is large, the output might also contain older activations that have already been forwarded and dealt with. Furthermore the notification is sent whenever the monitor is triggered, as dictated by the msg.pollingPeriod field. Thus one needs to configure these properties accordingly so that there is no spamming (multiple notifications for the same error message).

For example, if one is interested in a fine grained performance monitor, they can use a low polling period (e.g. 60 seconds) and a time window of 10 minutes. On the other hand, if one needs to manipulate the error outputs for debugging information, they may configure the OW monitor logger with the same value in the window and the polling period. This will ensure that there are no overlaps of information and only new errors will be propagated in each report. One consideration in this case is that the msg.window uses minutes while the polling period uses seconds so the necessary conversion needs to be applied.

4.11.4. Pattern publication means

The OW Monitor subflow is available

- at the Node-RED flows repository: https://flows.nodered.org/flow/a86475720659b3ed9eb5024052d94b1d/in/HXSkA2JJLcGA
- As an npm node: https://www.npmjs.com/package/node-red-contrib-owmonitor

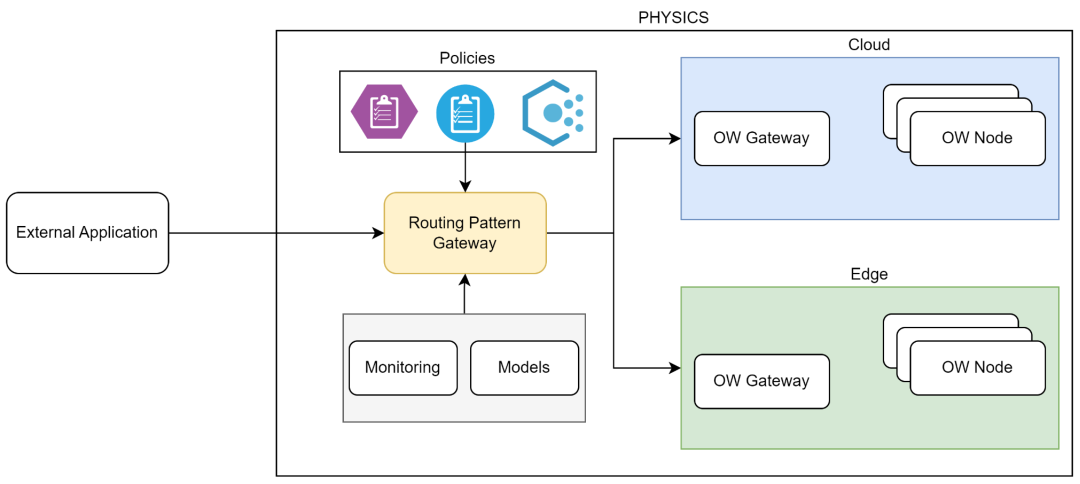
4.12. High Availability Routing Pattern

4.12.1. Pattern template description

The template description of the High Availability pattern appears in Table 21.

Table 21: Template Description for the High Availability Routing Pattern

Pattern name	High Availability Routing
Relation to Requirements	Ability to switch between different available Openwhisk locations based on monitored conditions
Source of pattern	Smart Manufacturing UC, Smart Agriculture UC
Pattern Context	In order to enhance aspects such as availability or performance of an application, more than one locations may be available in the cloud/edge continuum. Hence a way needs to exist in order to decide in real-time towards which location the requests will be forwarded. This decision should also adapt to the current conditions of execution in order to satisfy user requirements. For example, the Smart Manufacturing needs to define a default internal location for execution of the functions. However if that location is unavailable to switch to the main Cloud installation, so that no downtime appears in the production line. On the other hand, the Smart Agriculture UC needs by default to run the simulations on the central cloud for scalability reasons, but if for some reason the connection between the edge and the cloud is lost, to be able to run a limited simulation version on the edge just to maintain some level of results production.

Pattern Underlying Problem	The process should be based on a continuous monitoring approach. Furthermore, different users may have different requirements, hence the pattern should adapt to different needed metrics. Also the pattern should be able to be dynamically adjusted if at some point the redirection needs to change.
Pattern Usefulness/ Objective	The pattern integrates the usage of the Openwhisk monitoring subflow in order to enable the selection based on all the available metrics (success percentage, cold starts, wait time limits, duration etc). The user can set limits for the desired metric and the pattern redirects requests towards the secondary location if the limits are violated. Periodically it switches back to the main endpoint in order to check the status and redirects the traffic to check the status. The user can define the ratio of traffic directed towards the primary and the secondary endpoint.
Schema	 PHYSICS Policies Cloud OW Gateway OW Node External Application Routing Pattern Gateway Edge Monitoring Models OW Gateway OW Node
Prerequisites	The pattern can be included in the invocation flow towards the function.
Condition of application	In case of more than one available locations .
Input parameters	Selection of the metric and the target value upon which the redirection will be performed, as well as the ratio of redirection.
Output parameters	Invocation is redirected to the active output (primary or secondary) based on the defined ratio.
Included functionality	POST /endpoints (to update or set ratio of redirection) GET /endpoints
Pattern limitations	N/A
Linked to Pattern	Generic use, it needs as input the output of the Openwhisk monitoring subflow created from the PHYSICS project

Indicative Domains of Applicability	Availability and performance management

4.12.2. Pattern implementation details

The implementation is based on a Node-RED subflow that is fed by both the incoming messages as well as monitoring information from the PHYSICS Openwhisk Sliding Window monitor subflow (Figure 131).

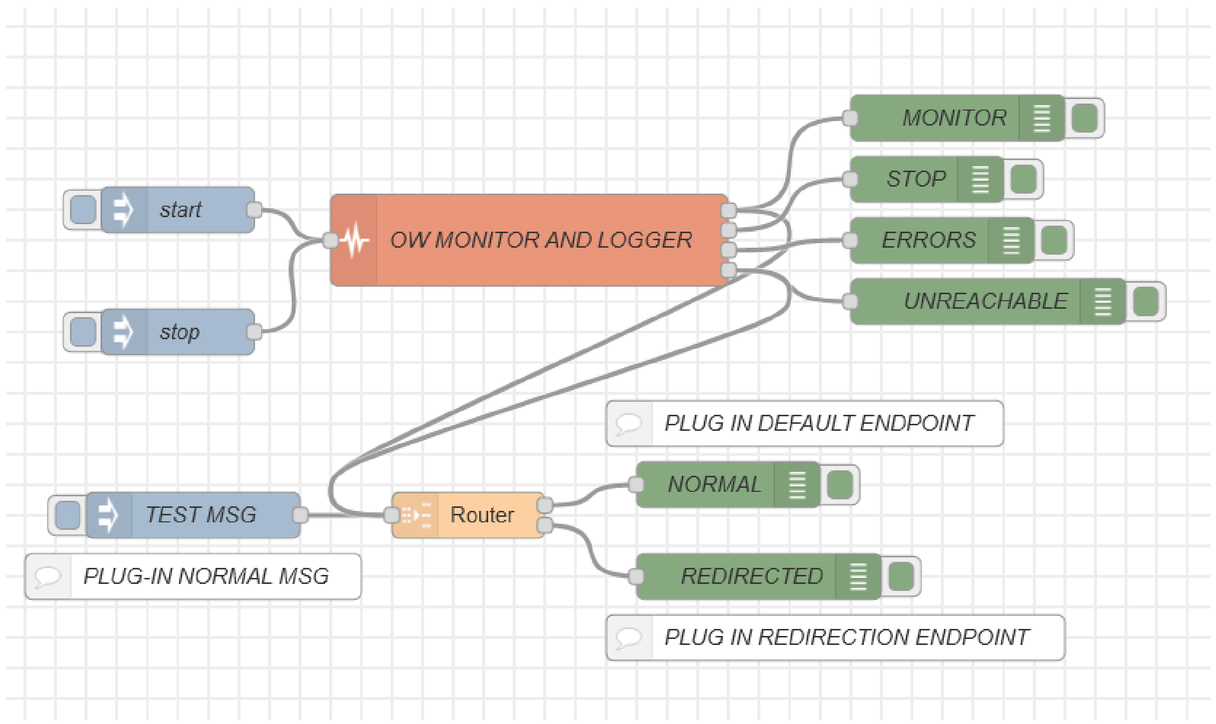
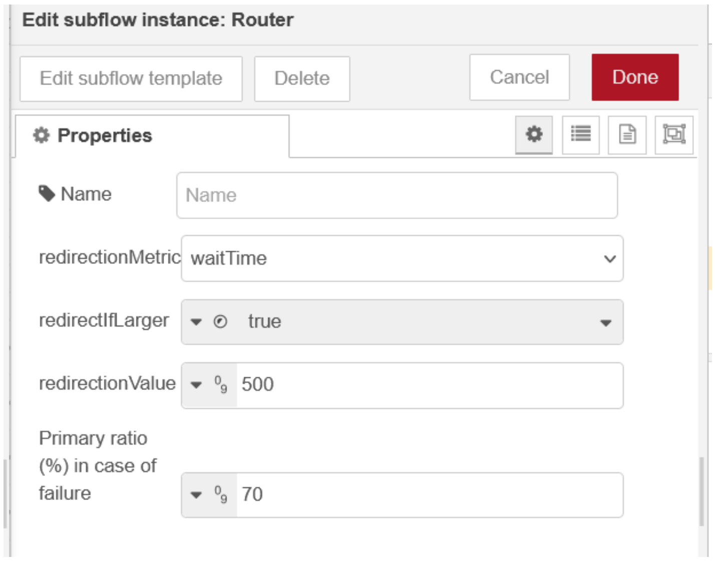


Figure 131: Integrated Use of Router with the OW Monitor

The user needs to select through the subflow UI the following parameters:

- redirection metric: one of the monitored parameters of the Openwhisk monitoring subflow, i.e. wait time, init time, duration or success percentage
- redirection value is the target value above or under which the redirection should occur
- comparison logic through the boolean redirectIfLarger parameter. This is used to define the comparison between the target value and the current value. So if redirectIfLargeris set to true, the redirection will occur if the redirection metric is larger than the redirection value
- primary ratio may be used to define whether the redirection will be of a fallback nature or a shared traffic option. Thus a default value of 0 will redirect all traffic to the secondary output.

- A non zero value will probabilistically redirect the according percentage of requests to the primary endpoint and the remaining ones to the second output.



Edit subflow instance: Router

Edit subflow template | Delete | Cancel | Done

Properties

Name: Name

redirectionMetric: waitTime

redirectIfLarger: true

redirectionValue: 500

Primary ratio (%) in case of failure: 70

Figure 132: Configuration UI for the Router

It uses flow variables in order to store the latest monitored metrics (Figure 133). These variables are updated based on the polling period set in the OW monitor subflow. The redirection applies only for these periods that the rule set by the developer is violated. In case it does not, the traffic is directed towards the main endpoint.

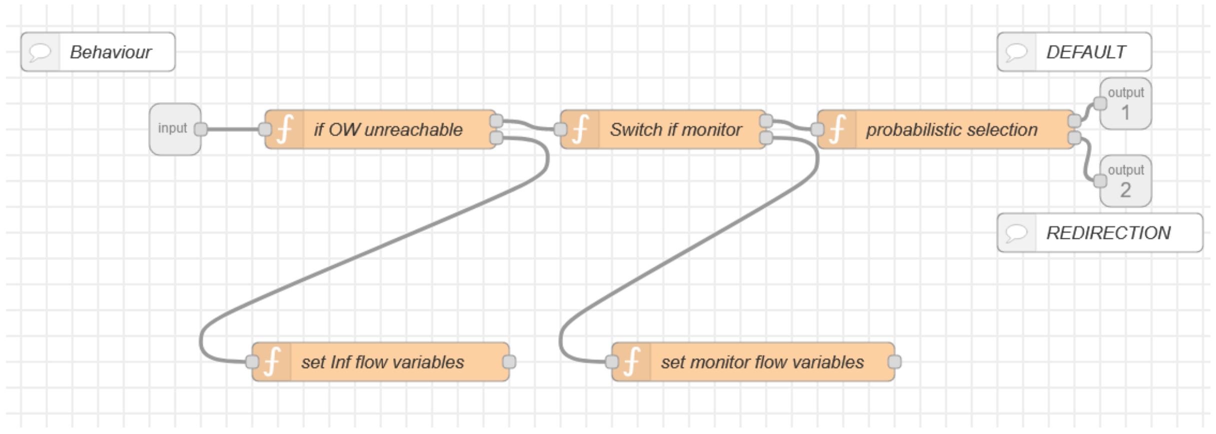


Figure 133: Implementation Flow for the Router

If the response from the Openwhisk environment is in the area of 50X (system not available) all the traffic will be automatically redirected to the secondary endpoint. The subflow defines 3 endpoints for dynamic management of the ratio:

- POST /endpoints for setting a new ratio for the amount of traffic that should be redirected. The ratio can also be defined at runtime through a POST /endpoints method with a body like the one shown in Code 6

```
{
 "endpoints":[
    {
    "endpoint":"primary",
    "percent":90
    }
  ]
}
```

Code 6: POST /endpoints payload for setting a new ratio for the amount of traffic

This will override the set value in the UI and can be used for runtime adaptation in order to support e.g. the implementation of Circuit Breaker approaches:

- GET /endpoints to retrieve the current set ratio
- DELETE /endpoints in order to reset the ratio. If applied, then the UI values will be reasserted.

4.12.3. Pattern examples of usage

In the aforementioned UI screenshot configuration, 30% of the requests coming in at the Test Msg inject node will be forwarded to the redirection endpoint if the wait time of the default Openwhisk location exceeds 500 milliseconds. The remaining 70% will be forwarded to the primary endpoint output. If the wait time is under 500 milliseconds, all of the messages will be directed towards the primary output. The decision will be updated each time new metrics from the OW monitor come in. This is configurable in the OW monitor subflow (polling period option). In case of inability to reach the default Openwhisk endpoint (indicated by a relevant failed value in the incoming `msg.statusCode` property of the OW monitor output) the Router switches by default to the redirection output. Thus if the normal message that needs to be routed has a `msg.statusCode` property, it should be removed before feeding into the Router node.

Furthermore, the combination of the OW monitor and the Router were also used in the context of D4.2 and the experiment on the runtime adaptation described in that document. In that case the Router was stripped down of the comparing logic and integrated with the Forecaster Engine in order to receive the percentages of redirection predicted by the Forecaster. The according adapted flow appears in Figure 134 while the details of the experiment are included in D4.2.

4.12.4. Pattern limitations

The user of the pattern should take under consideration that oscillations may be observed during operation. Hence a step wise logic for reducing the ratio could be applied by exploiting the dynamic POST endpoints for ratio setting.

4.12.5. Pattern publication means

All relevant artefacts have been created and uploaded in the Node-RED flow repository and included in the project assets repository[28].

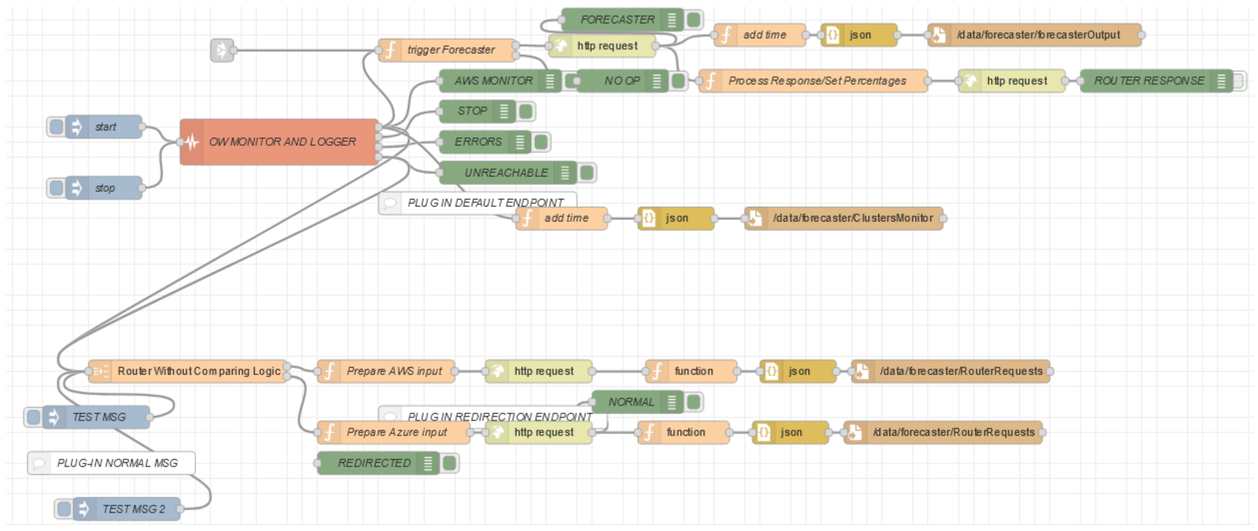


Figure 134: Integrated Usage Between Router, OW Monitor and Forecaster from WP4

4.13. Semaphore Node Pattern

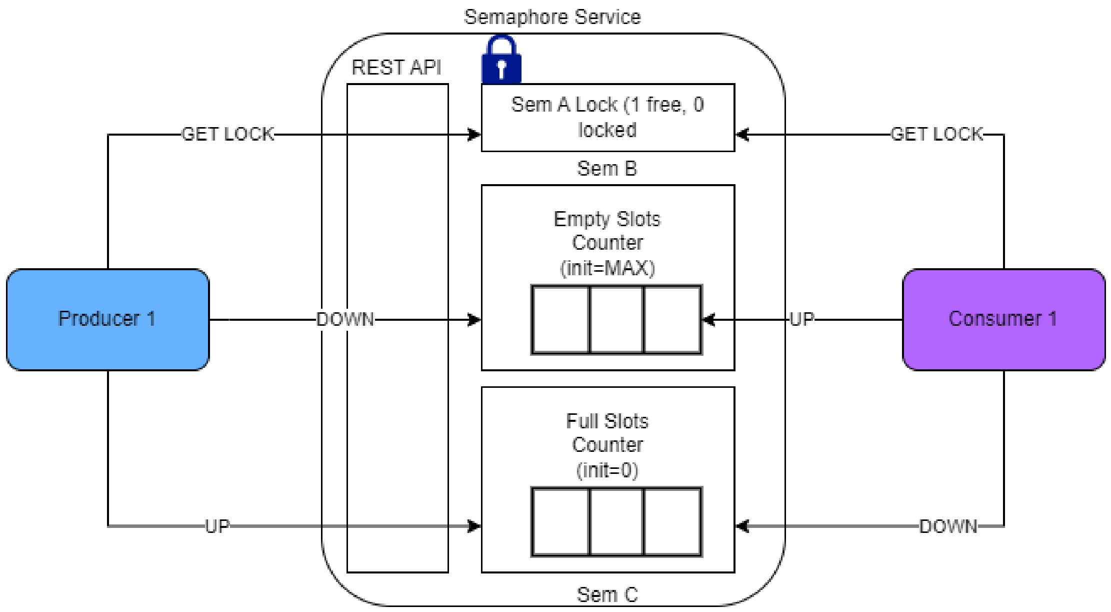
4.13.1. Pattern template description

The template description for the Distributed Lock pattern appears in Table 22.

Table 22: Template Description for the Split Join pattern Pattern

Pattern name	Semaphore Service
Relation to Requirements	Synchronization needs in arbitrary workflows with distributed clients
Source of pattern	Workflow Orchestration
Pattern Context	Two or more functions that need to synchronize to proceed with an operation in an isolated or coordinated manner

[28] https://flows.nodered.org/flow/359b15796c59e574de354e7f243ed3c4/in/HXSkA2JJLcGA

Pattern Underlying Problem	In many cases, distributed parts of an application execution may need to be synchronized or mutually excluded with relation to a given action. This in typical system programming is performed through semaphore/thread libraries or even higher level abstractions offered from common languages. However, in a distributed function context, these mechanisms are not available between code that may potentially be running in different functions or, even if it is in the same function, different function instances may execute on different containers.
Pattern Usefulness/ Objective	The pattern aims to exploit the ability of node.js not to interrupt function execution, as long as no node.js worker threads or asynchronous operations are used. Thus the access to a common resource such as a flow or global variable in a Node-RED flow is guaranteed to be performed at any given time only by the executing function. To this end it exposes a CRUD approach on semaphore creation, achievable through REST APIs.
Schema	 Semaphore Service REST API Sem A Lock (1 free, 0 locked Sem B Empty Slots Counter (init=MAX) Full Slots Counter (init=0) Sem C Producer 1 Consumer 1 GET LOCK GET LOCK DOWN UP UP DOWN
Prerequisites	The pattern needs to be applied as a service.
Condition of application	In case of distributed clients that need to enter a critical section or coordinate their execution.
Input parameters	Semaphore name to perform the CRUD operation, type of operation (down to obtain the lock, up to release the lock).
Output parameters	Depends on the method used, can be success/fail on doing the operation as well as details on the semaphore status (e.g. value after the operation).
Included functionality	Create or delete a semaphore resource Raise (up) or lower (down) the value of a semaphore
Pattern limitations	N/A

Linked to Pattern	Generic use, however the specific pattern can collaborate with the SJ one. The split join pattern for example can reduce the rate of creating containers based on a typical producer/consumer model, in which the available container slots in the testbed are taken under consideration
Indicative Domains of Applicability	Parallel computations, Workflow creation, Distributed Application Synchronization

4.13.2. Pattern implementation details

The implementation is based on a Node-RED subflow that creates a semaphore service to act as a distributed lock/synchronization mechanism (Figure 135). It uses flow variables in order to store the state of a semaphore. Changes in the semaphore value are uninterruptable at the level of the node.js process. This is due to the fact that the latter executes them in the single threaded eventloop and does not interrupt their execution unless an async method or worker thread is used in the function's implementation.

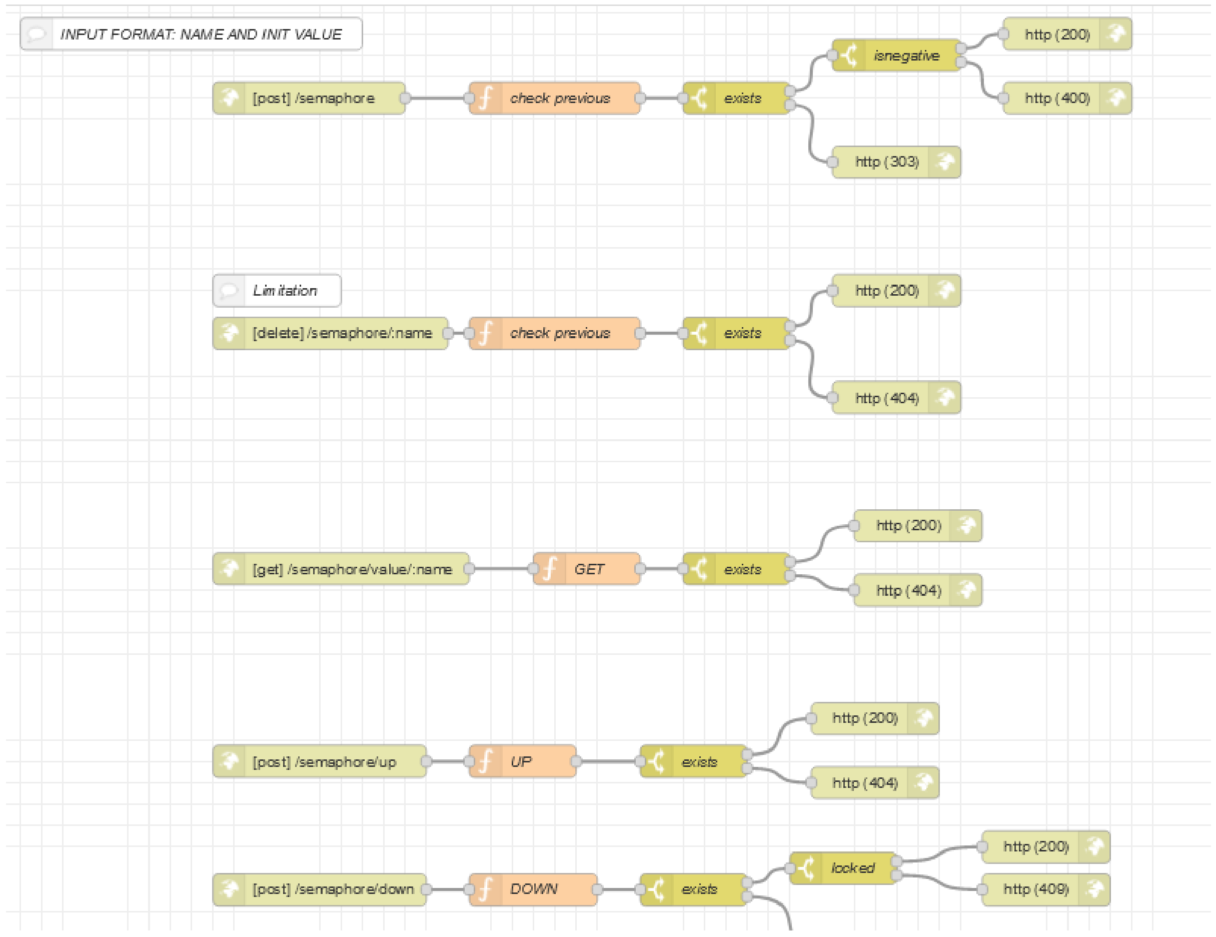


Figure 135: Implementation of the Semaphore Service

The subflow defines 5 endpoints:

- POST /semaphore for new semaphore creation. The body should contain name and initialization value of the created semaphore: {"name":"<semname>","value":1}. The value needs to be a positive integer. If not positive, a 400 HTTP error code is returned. A positive float will be converted to integer. If the semaphore already exists, a 303 HTTP error code is returned.
- DELETE /semaphore needs to have only the name attribute in the body ({"name":"<semname>"}). If the semaphore does not exist a 404 HTTP error code is returned
- GET /semaphore/value/:name for retrieving the current value. If the semaphore does not exist a 404 HTTP error code is returned
- POST /semaphore/up for increasing the value by 1 with a body of the semaphore name {"name":"<semname>"} . If the semaphore does not exist a 404 HTTP error code is returned.
- POST /semaphore/down for decreasing the value by 1 with a body of the semaphore name {"name":"<semname>"}. If the semaphore is already at 0,a relevant message Semaphore locked with a 409 HTTP error code is returned. If the semaphore does not exist a 404 HTTP error code is returned.

4.13.3. Pattern examples of usage

A created semaphore can be used as a lock (if initialized at 1). As in any semaphore related library, it is the responsibility of the clients of the distributed application to correctly use a call sequence that will indicate if the client can proceed or not to what is considered the critical section or to correctly use the up/down methods.

For example, a semaphore locked by one client (with a down at 1, resulting to the value being 0) can be unlocked by another client that performs afterwards an up from 0 at the same semaphore. There is no notion of lock ownership by a specific client that performed the initial down. Compared to the typical semaphore libraries, this implementation does not have the ability to make the calling process sleep, if the semaphore is locked.

A created semaphore can also be used as a synchronization counter (any initialization value>0 can be used) in a type of producer/consumer problem. However the lock gets activated at 0, like commonly in semaphores, thus a reverse semantics semaphore needs to be used. For example defining the max available slots and then reducing by one for each producer client. An example producer/consumer implementation using this subflow is included in Figure 136.

4.13.4. Pattern necessary adaptations

The clients need to have the according logic implemented in order to exploit the outcome of the REST call. Thus in case of not being able to acquire a lock, they should retry until succeeding.

4.13.5. Pattern limitations

The GET method is included only for informative reasons. There is no guarantee that the value might not change by the time the response is received by the client. In this version a kind of busy wait and polling of the client is anticipated until getting the lock.

4.13.6. Pattern variations

For the future, a relevant variation may include the ability to have callback URIs of the client and a queue inside the semaphore, in order to alert them once a lock is freed.

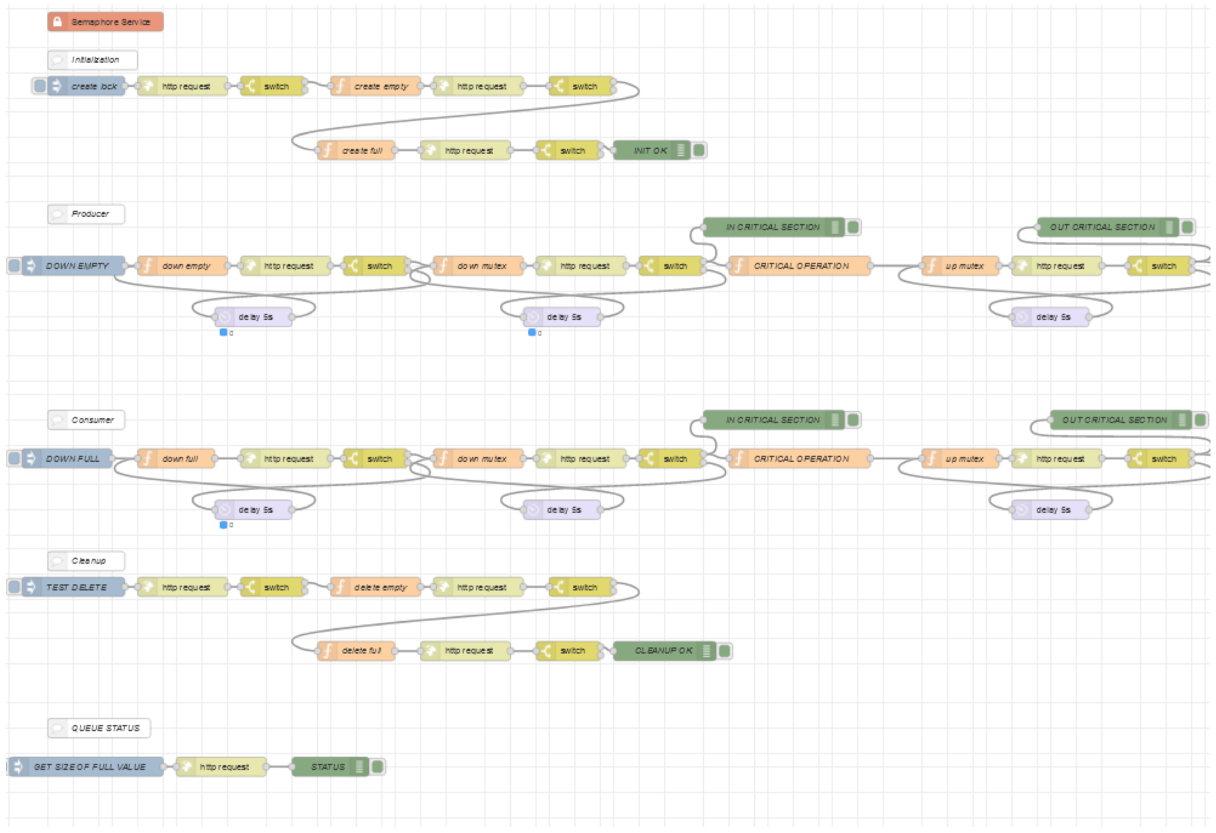


Figure 136: Example Use of the Semaphore Pattern for the Instantiation of a Producer/Consumer structure

4.13.7. Pattern publication means

All relevant artefacts have been created and uploaded in the Node-RED flow repository and included in the project assets repository[29][30].

[29] Main subflow node: https://flows.nodered.org/flow/5145332c834610e917776e2835bd8037/in/HXSkA2JJLcGA

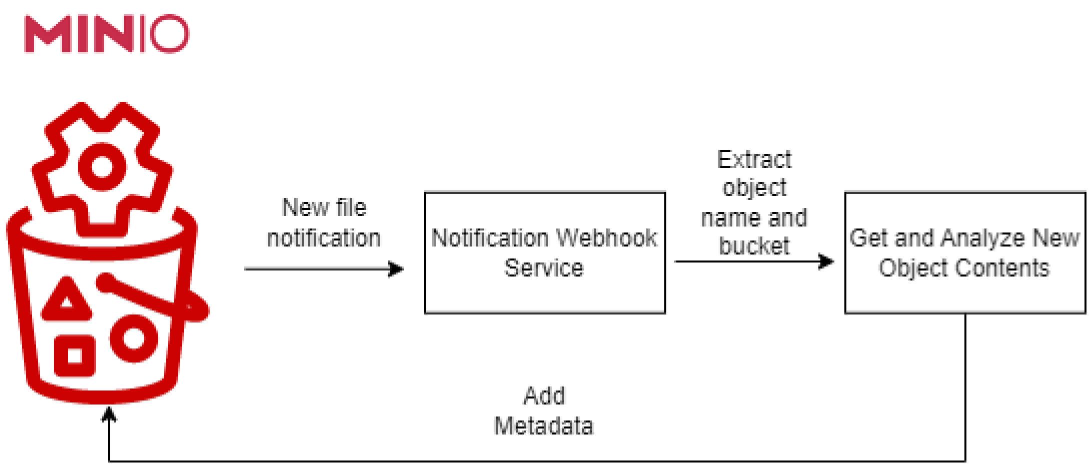
[30] Testing flows for the producer consumer example https://flows.nodered.org/flow/4ef7edabc7c651334d9104bebc6d65f4/in/HXSkA2JJLcGA

4.12. Automated Object File Annotator Pattern

4.12.1. Pattern template description

The template description for the Automated Object File Annotator pattern appears in Table 23.

Table 23: Template Description for the Split Join Pattern

Pattern name	Automated Object File Annotator
Relation to Requirements	Preprocessing of data to reduce large queries
Source of pattern	ETL processes
Pattern Context	An object file with data can include arbitrary metadata annotations that describe some aspect of these data (e.g. min/max of the values inside the object).
Pattern Underlying Problem	When large datasets are created, typically through a gradual upload of individual files, querying of these data or subsequent processing and analysis at some stage may become too resource intensive, having to filter a very large number of files and data points within these files.
Pattern Usefulness/ Objective	The pattern aims to exploit the notification features of common object storage systems like Minio and AWS S3, as well as the ability to define any custom metadata on top of an object file, in order to automatically extract annotations for new individual files contents when the initial files are created and uploaded. Having rich metadata means that in a subsequent analysis stage, one can query these metadata structures and directly filter out files that are out of the query range without having to access the contents of the files. This means that the needs for actual file content access are significantly reduced. Extracting metadata from the initial small files is much easier than having to do that on the entire file collection at a subsequent stage.
Schema	 MINIO New file notification Notification Webhook Service Extract object name and bucket Get and Analyze New Object Contents Add Metadata

Prerequisites	The pattern needs to be applied as a service. An existing Minio installation needs to exist and a webhook notification endpoint needs to be configured to target the pattern service.
Condition of application	In case of large datasets that are stored in many small files collections.
Input parameters	Name of the data field for which the metadata need to be calculated. The input to the pattern is the automatically generated notification from the Minio system whenever a new object file is uploaded on the monitored bucket.
Output parameters	Success or fail of adding metadata to the file
Included functionality	POST/events web hook endpoint
Pattern limitations	At the moment, the pattern implements a generic example of a csv file format, from which the developer defines which column name is the one to be used for extracting metadata. Examples with extracting the minimum and maximum values of that column are included in the implementation. In case some other form of input file or other form of metadata calculation needs to be performed, the pattern flow needs to be adapted. This is reasonable since it falls under the arbitrary application scope and data format.
Linked to Pattern	N/A
Indicative Domains of Applicability	Big Data, Data Analytics, ETL processes

4.12.2. Pattern implementation details

The implementation is based on a Node-RED subflow that creates a POST /events webhook endpoint for Minio to push the relevant notifications (Figure 137). Following, the flow filters notification messages based on new objects put but also if they already have the specific metadatum needed. This is because the flow puts the object again with the new metadata, so it creates a new notification and thus an infinite loop if this comparison is not made. The user can configure which metadata field this skip comparison can be performed in the relevant switch node 'if already has metadata skip'. The user also needs to configure the details of the MINIO instance in the MINIO nodes.

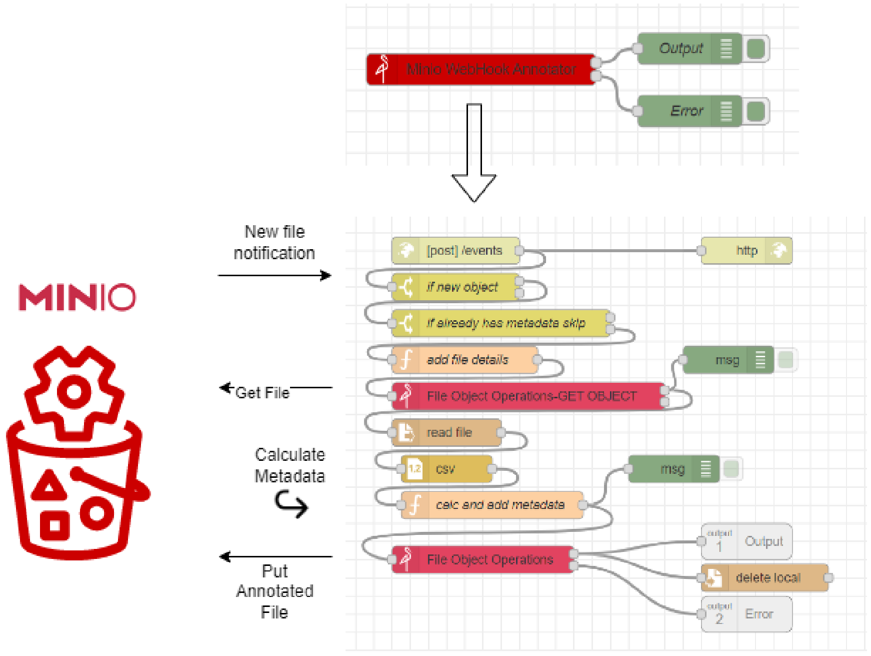


Figure 137: Implementation of the Automated Object File Annotator Pattern

4.12.3. Pattern examples of usage

The flow in Figure 137 shows an example of using a csv file with columns id and value and extracting the min and max of the column name set in the flow UI (default: value). The user can change this in the function node 'calc and add metadata', where the relevant logic can be adapted. When the file gets uploaded on the bucket that has been configured to use the specific webhook for notifications, a new notification will be generated and forwarded to the endpoint. This will trigger the execution of the flow. Input and output results are included in Figure 138.

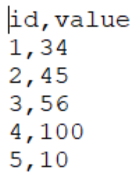
4.12.4. Pattern necessary adaptations

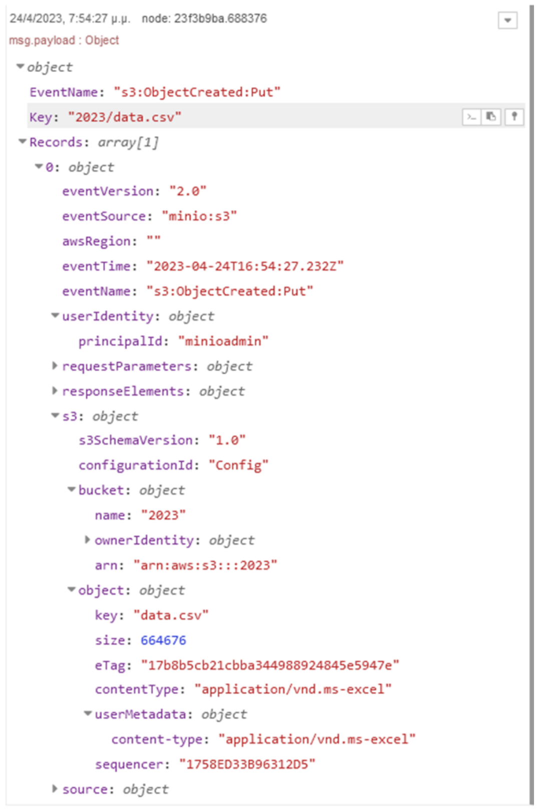
Adaptations need to be made if the input file is of different format and also if new metadata calculations are needed.

4.12.5. Pattern limitations

At the moment the flow needs to retrieve the file object, read the contents and then put again the object to replace the previous version. In future versions we will exploit the copyObject API method in order to avoid putting back the overall object but just replacing the metadata.

Phase	Example

Input file	`id,value`
	`1,34`
	`2,45`
	`3,56`
	`4,100`
	`5,10`

Notification in webhook	24/4/2023, 7:54:27 μ.μ. node: 23f3b9ba.688376
	msg.payload : Object
	`▾object`
	`  EventName: "s3:ObjectCreated:Put"`
	`  Key: "2023/data.csv"`
	`▾Records: array[1]`
	`  ▾0: object`
	`      eventVersion: "2.0"`
	`      eventSource: "minio:s3"`
	`      awsRegion: ""`
	`      eventTime: "2023-04-24T16:54:27.232Z"`
	`      eventName: "s3:ObjectCreated:Put"`
	`    ▾userIdentity: object`
	`        principalId: "minioadmin"`
	`    ▸requestParameters: object`
	`    ▸responseElements: object`
	`    ▾s3: object`
	`        s3SchemaVersion: "1.0"`
	`        configurationId: "Config"`
	`      ▾bucket: object`
	`          name: "2023"`
	`        ▸ownerIdentity: object`
	`          arn: "arn:aws:s3:::2023"`
	`      ▾object: object`
	`          key: "data.csv"`
	`          size: 664676`
	`          eTag: "17b8b5cb21cbba344988924845e5947e"`
	`          contentType: "application/vnd.ms-excel"`
	`        ▾userMetadata: object`
	`            content-type: "application/vnd.ms-excel"`
	`          sequencer: "1758ED33B96312D5"`
	`    ▸source: object`

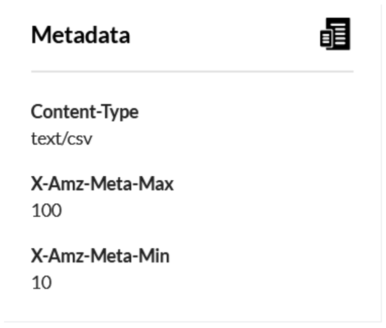
Output result (metadata included in Minio)	**Metadata** **Content-Type** text/csv **X-Amz-Meta-Max** 100 **X-Amz-Meta-Min** 10

Figure 138: Example Inputs and Outputs of the Automated Object File Annotator

4.12.6. Pattern publication means

All relevant artefacts have been created and uploaded in the Node-RED flow repository and included in the project assets repository[31].

4.13. Dynamic Orchestrator Pattern Section

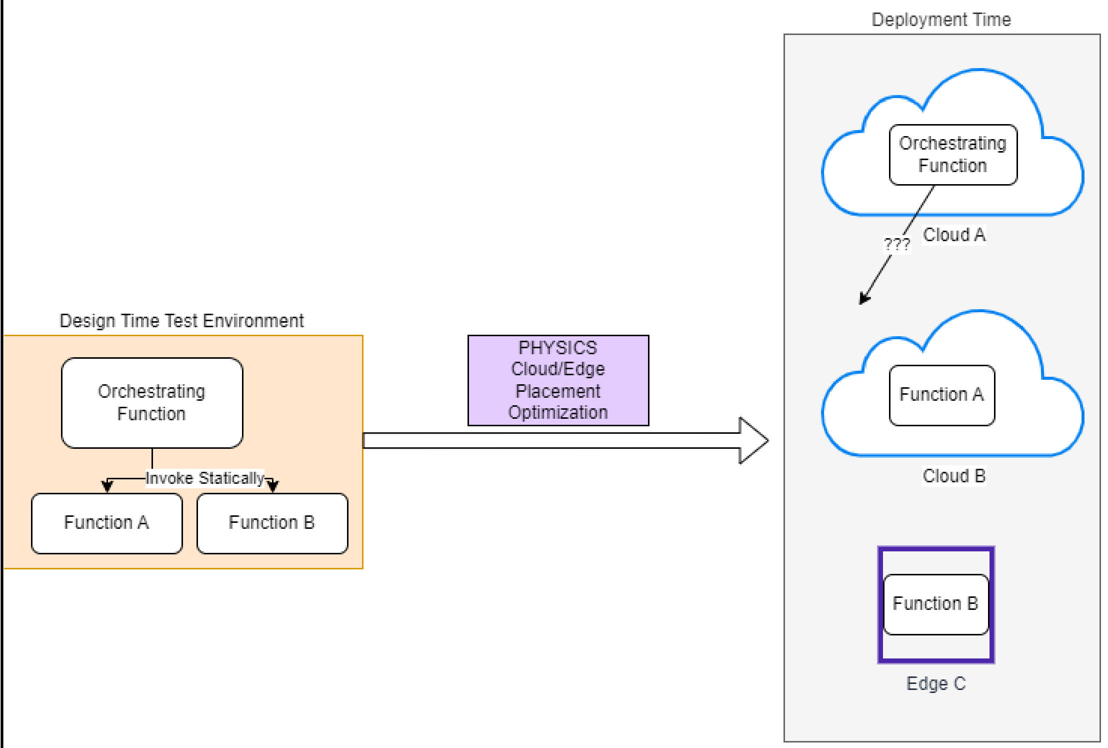
4.13.1. Pattern template description

The template description for the Dynamic Orchestrator pattern appears in Table 24.

Table 24: Template Description for the Dynamic Orchestrator Pattern

Pattern name	Dynamic Orchestrator
Relation to Requirements	WorkflowDef, Placement Optimization
Source of pattern	PHYSICS dynamic scenarios for optimized deployments across hybrid cloud/edge or multi-cloud environments
Pattern Context	Usage of function workflows in hybrid cloud/edge or multi-cloud environments in which dynamic placement of the functions may be applied or changed at any time. The Dynamic Orchestrator aims to support the creation of an application that includes functions (a.k.a. Orchestrator functions) that somehow use other functions in their process.
Pattern Underlying Problem	While the orchestrator function is created at design time, the developer uses endpoints towards the externally used functions on the test or even production OW environment. However, during the normal application deployment, PHYSICS can include in its placement optimization step the distribution of one or more used functions in different locations if multiple OW clusters are available. Therefore the workflow needs to be updated with the correct endpoints to use for every invocation to an external function.
Pattern Usefulness/ Objective	A relevant pattern needs to be applied in order for the orchestrating function to be made aware of the eventually used locations so that it can function properly and invoke the used functions in the finally selected endpoints. This operation could be performed through other means, e.g. proxy functions or services, however this would imply the insertion of another step in the request, leading to higher response times, as well as bottlenecks and single points of failure in the system since all relevant requests would pass from the proxy service.

[31] https://flows.nodered.org/flow/f91edbe34663d0e4074d5ab37b65a3e0/in/HXSkA2JJLcGA

Schema	 Design Time Test Environment Orchestrating Function Invoke Statically Function A Function B PHYSICS Cloud/Edge Placement Optimization Deployment Time Orchestrating Function ??? Cloud A Function A Cloud B Function B Edge C
Prerequisites	Semantic Extractor, WP4 Orchestrator (for automated usage inside PHYSICS)
Condition of application	Usage in a flow that invokes an external action at a multi-cluster environment
Input parameters	Name of the external action to invoke. If used independently from the remaining PHYSICS platform, the incoming message to the Dynamic OW Action needs to include the action name in the node UI as well as the host location, namespace and credentials to the external location where the function to be invoked is deployed
Output parameters	N/A
Included functionality	The provided pattern adapts the url of the external action invocation to the provided input
Pattern limitations	N/A
Linked to Pattern	Node-RED flow as function

Indicative Domains of Applicability	Serverless computing environments, cloud application development, workflow orchestration

4.13.2. Pattern implementation details

The pattern is implemented as a PHYSICS semantic node subflow and appears in Figure 139:

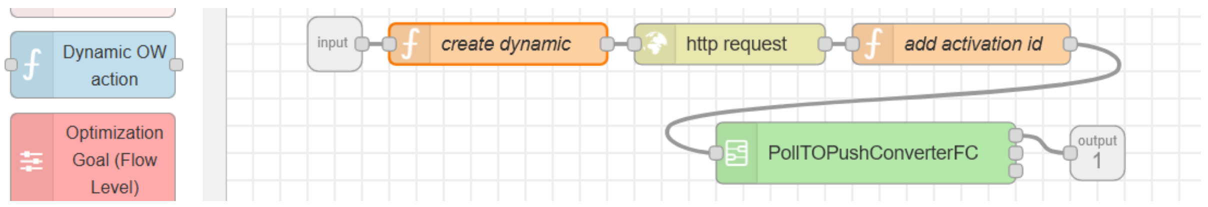


Figure 139: Implementation of the Dynamic Orchestrator Pattern

Its main operation is rather simple, it exploits incoming information from the message in order to adapt the url to which the function invocation will be performed. The incoming message to the Dynamic OW Action needs to include the action name in the node UI as well as the following fields in the msg.payload.value:

- __<action_name>_HOST
- __<action_name>_NAMESPACE
- __<action_name>_CREDS

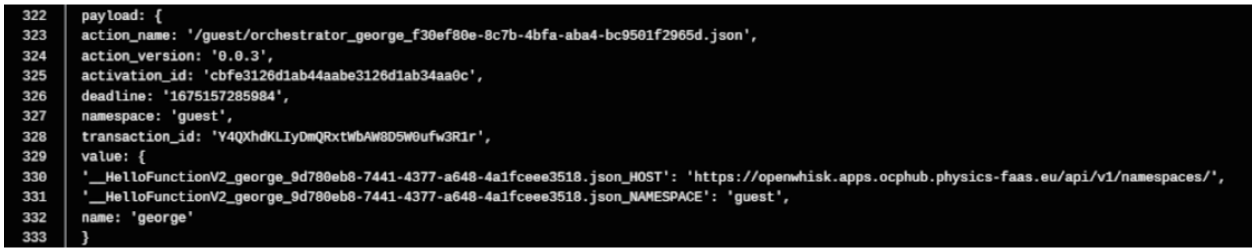
The subflow includes also the PollToPushConverterFC, so that it directly returns the final result of the invoked function for the continuation of the workflow. By using that subflow, the developer may create an arbitrary workflow of various used actions, linked in whatever manner they see fit.

The difficult part in this case is to define how the overall process will be performed automatically. To enable its automation, one of Openwhisk's features may be exploited, i.e. the ability to define parameters to a registered function upon creation or update. An example of such a registration appears in the following snippet:

```
wsk action update -i <ORCHESTRATOR FUNCTION NAME> -p __<DYNAMIC ACTION NAME>_HOST
https://openwhisk.apps.ocphub.physics-faas.eu/api/v1/ -p __<DYNAMIC ACTION NAME>_NAMESPACE guest
-p __<DYNAMIC ACTION NAME>_CREDS user:token
```

Code 7: An example define parameters to a registered function upon creation or update

Then the parameter info is included by default by Openwhisk in each invocation of that action, along with any other input the user may have defined.



```
payload: {
action_name: '/guest/orchestrator_george_f30ef80e-8c7b-4bfa-aba4-bc9501f2965d.json',
action_version: '0.0.3',
activation_id: 'cbfe3126d1ab44aabe3126d1ab34aa0c',
deadline: '1675157285984',
namespace: 'guest',
transaction_id: 'Y4QXhdKLIyDmQRxtWbAW8D5W0ufw3R1r',
value: {
'__HelloFunctionV2_george_9d780eb8-7441-4377-a648-4a1fceee3518.json_HOST': 'https://openwhisk.apps.ocphub.physics-faas.eu/api/v1/namespaces/',
'__HelloFunctionV2_george_9d780eb8-7441-4377-a648-4a1fceee3518.json_NAMESPACE': 'guest',
name: 'george'
}
```

Figure 140: Example Function Input including Dynamic Parameters of Invocation

Thus in the context of PHYSICS, the Orchestrator component in WP4 needs to pass these parameters in the orchestrating function container automatically through the respective K8S configuration for all dynamic actions used in a flow, during their creation. This step happens after the placement optimization, hence the parameterization of the orchestrator function will include the finally selected location for the target function.

One final step is needed in the process, the ability of the platform in WP4 to detect which dynamic actions are used in an app graph. This was the reason the Dynamic OW Action subflow is defined as a semantic node. Following the typical PHYSICS process, any relevant properties of that node (in this case the name of the external function to use) get included in the App Graph sent to WP4 as an annotation. Thus the Orchestrator component in WP4 can find the DynamicActionName tag and configure the registration of the orchestrator function with the location parameters for the action name that is included in this tag.

4.13.3. Pattern examples of usage

In the example below (Figure 141), an orchestrating function is created that invokes another function. The name of the other function needs to be included in the Dynamic OW action UI.

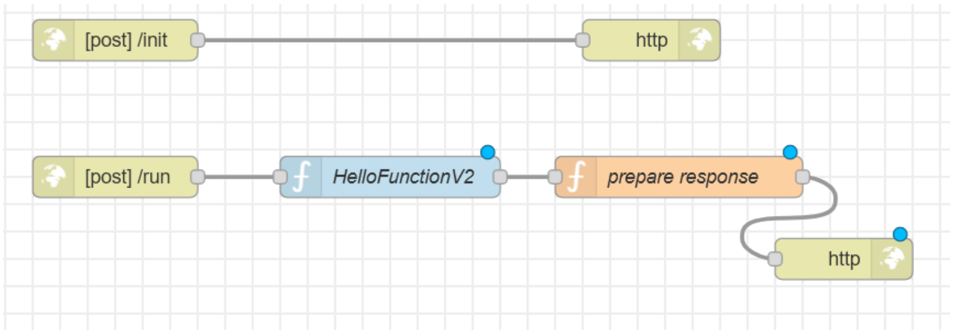
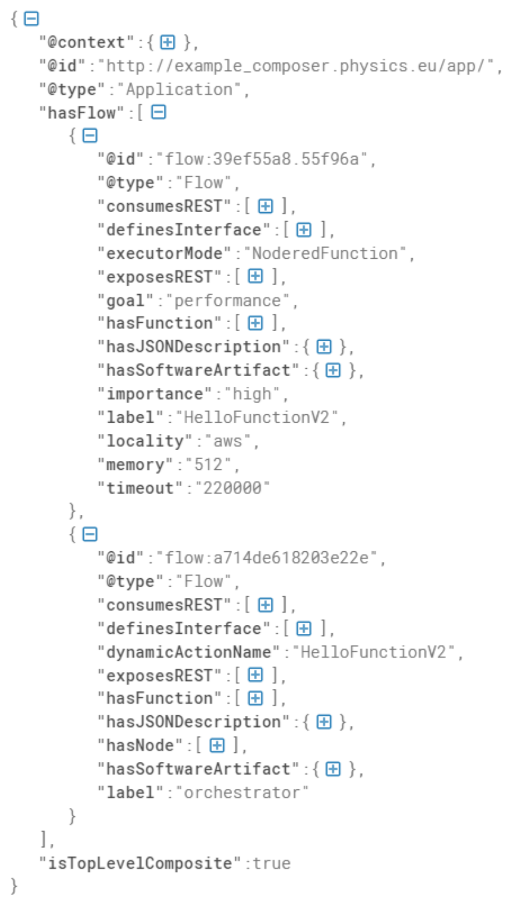


Figure 141: Example Function using the Dynamic Orchestrator Pattern

Assuming that the finally created application will also contain the HelloFunctionV2 (if not then the HelloFunctionV2 is already deployed and there is no need for the dynamic orchestration), an example of an app graph appears in Figure 142 that includes the aforementioned orchestrating function, in which the usage of the second function (helloFunctionV2) is included in the dynamicActionName tag.



```
{ ⊟
   "@context" : { ⊞ },
   "@id" : "http://example_composer.physics.eu/app/",
   "@type" : "Application",
   "hasFlow" : [ ⊟
      { ⊟
         "@id" : "flow:39ef55a8.55f96a",
         "@type" : "Flow",
         "consumesREST" : [ ⊞ ],
         "definesInterface" : [ ⊞ ],
         "executorMode" : "NoderedFunction",
         "exposesREST" : [ ⊞ ],
         "goal" : "performance",
         "hasFunction" : [ ⊞ ],
         "hasJSONDescription" : { ⊞ },
         "hasSoftwareArtifact" : { ⊞ },
         "importance" : "high",
         "label" : "HelloFunctionV2",
         "locality" : "aws",
         "memory" : "512",
         "timeout" : "220000"
      },
      { ⊟
         "@id" : "flow:a714de618203e22e",
         "@type" : "Flow",
         "consumesREST" : [ ⊞ ],
         "definesInterface" : [ ⊞ ],
         "dynamicActionName" : "HelloFunctionV2",
         "exposesREST" : [ ⊞ ],
         "hasFunction" : [ ⊞ ],
         "hasJSONDescription" : { ⊞ },
         "hasNode" : [ ⊞ ],
         "hasSoftwareArtifact" : { ⊞ },
         "label" : "orchestrator"
      }
   ],
   "isTopLevelComposite" : true
}
```

Figure 142: Example of App Graph with Dynamic Action Inclusion

4.13.4. Pattern necessary adaptations

The user needs to include the external function name to invoke.

4.13.5. Pattern publication means

The subflow is available in the PHYSICS collection on the Node-RED flows repository as well as in the RAMP[32].

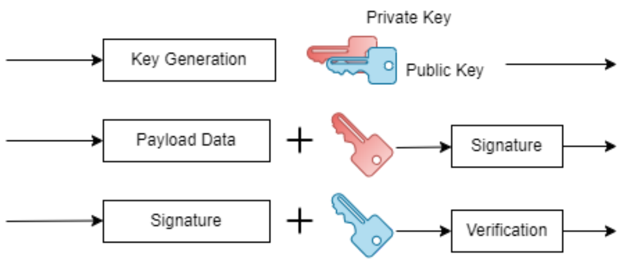
[32] https://flows.nodered.org/flow/7970809f43d73b0a2e7af27f7165420c#

4.14. Digital Signatures Pattern

4.14.1. Pattern template description

The template description for the Digital Signatures Pattern appears in Table 25.

Table 25: Template Description for the Digital Signatures Pattern

Pattern name	Digital Signatures
Relation to Requirements	Req-3.4-Encryption, Req-3.4-SecureComms, Req-6.2-Privacy
Source of pattern	All UCs
Pattern Context	Public-key cryptography functionalities for digital data signatures and verification.
Pattern Underlying Problem	Authentication for data payloads, ensuring that data cannot be renounced or transactions denied.
Pattern Usefulness/ Objective	Provide easy to interact public key infrastructure functionalities.
Schema	 Private Key Key Generation Public Key Payload Data + Signature Signature + Verification
Prerequisites	To generate an asymmetric keypair and save it in a secure manner.
Condition of application	N/A
Input parameters	The privateKey when invoking signing operation and signature with publicKey for verification operation
Output parameters	Keypair for key generation, signature for signing operation and verification response for verifying operation
Included functionality	Generation of asymmetric keypair, signing payloads and verifying payload signatures.
Pattern limitations	The keys (privateKey or publicKey) can be added to the payload during deployment time of the function, creating the need to deploy them in a secure manner.

Linked to Pattern	Generic use, data signing, verifying signature of payload
Indicative Domains of Applicability	Data authentication, Data protection, Non repudiation

4.14.2. Pattern implementation details

The pattern is implemented using the PKI node, of the node-red-contrib-crypto-blue package, which utilizes the Ed25519 curve for asymmetric key-pair generation, signing payloads and verifying payload signatures (Figure 143).

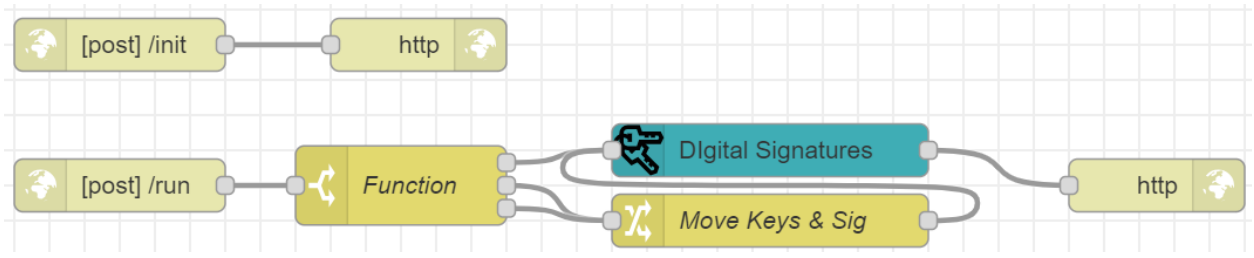
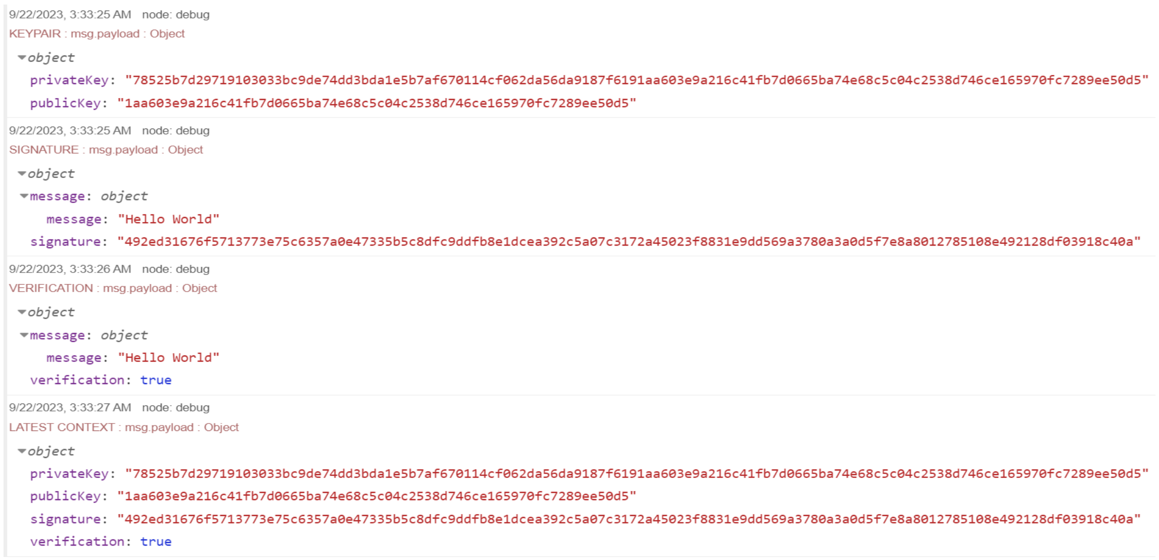


Figure 143: Node-RED flow of the Digital Signatures Pattern

The digital signatures keys can be generated by setting the payload function to "*generate*", creating a private key and its corresponding public key (these are stored in the flow context when running the tests). Signing a payload can be done by setting the payload function to "*sign*" with the payload accompanied by the private key that will be used to sign the payload data. Signature verification can be done by setting the payload function to "*verify*" accompanied by the signature and the public key that will be used to verify the data signature. An example output for each function mentioned, appears in Figure 144.



```
9/22/2023, 3:33:25 AM   node: debug
KEYPAIR : msg.payload : Object
 ▾object
   privateKey: "78525b7d29719103033bc9de74dd3bda1e5b7af670114cf062da56da9187f6191aa603e9a216c41fb7d0665ba74e68c5c04c2538d746ce165970fc7289ee50d5"
   publicKey: "1aa603e9a216c41fb7d0665ba74e68c5c04c2538d746ce165970fc7289ee50d5"
9/22/2023, 3:33:25 AM   node: debug
SIGNATURE : msg.payload : Object
 ▾object
  ▾message: object
     message: "Hello World"
   signature: "492ed31676f5713773e75c6357a0e47335b5c8dfc9ddfb8e1dcea392c5a07c3172a45023f8831e9dd569a3780a3a0d5f7e8a8012785108e492128df03918c40a"
9/22/2023, 3:33:26 AM   node: debug
VERIFICATION : msg.payload : Object
 ▾object
  ▾message: object
     message: "Hello World"
   verification: true
9/22/2023, 3:33:27 AM   node: debug
LATEST CONTEXT : msg.payload : Object
 ▾object
   privateKey: "78525b7d29719103033bc9de74dd3bda1e5b7af670114cf062da56da9187f6191aa603e9a216c41fb7d0665ba74e68c5c04c2538d746ce165970fc7289ee50d5"
   publicKey: "1aa603e9a216c41fb7d0665ba74e68c5c04c2538d746ce165970fc7289ee50d5"
   signature: "492ed31676f5713773e75c6357a0e47335b5c8dfc9ddfb8e1dcea392c5a07c3172a45023f8831e9dd569a3780a3a0d5f7e8a8012785108e492128df03918c40a"
   verification: true
```

Figure 144: Example Output of the Digital Signature Node

4.14.3. Pattern examples of usage

The basic operations of the pattern include the generation of the keypair, the signature creation of a data payload and its verification (Figure 145). The pattern can be used whenever someone wants to provide authentication for data in the flow or verify data signatures created from other functions.

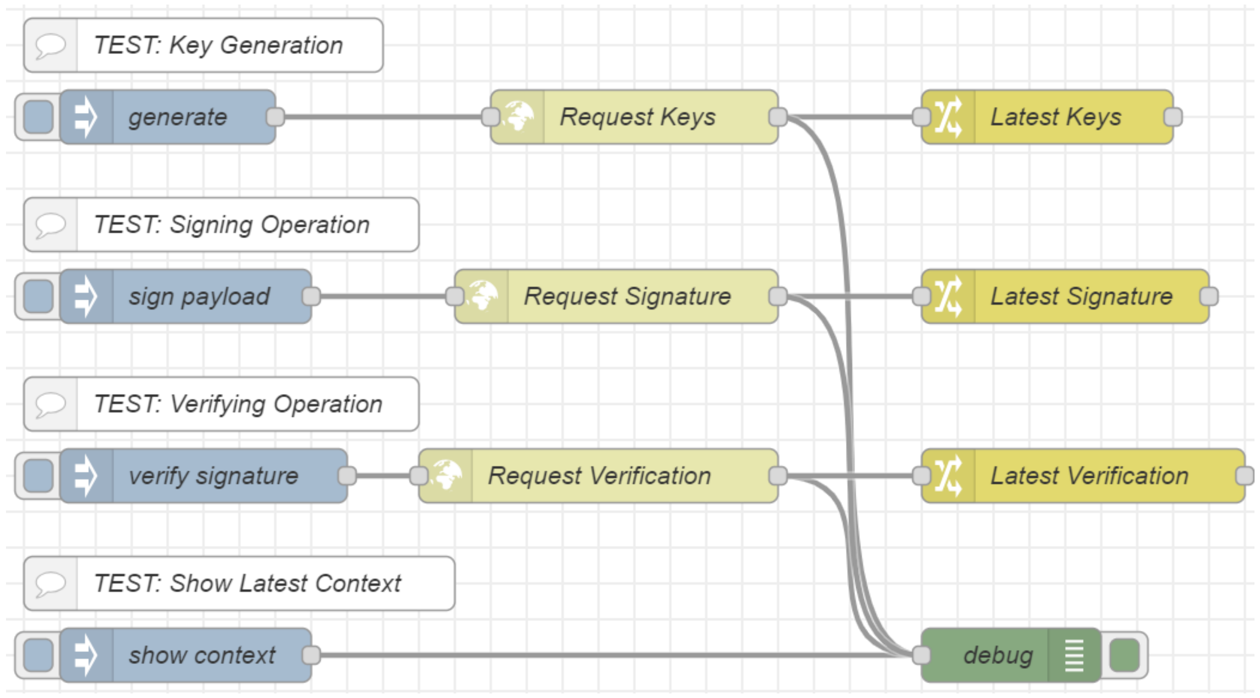


Figure 145: Example Usage Flows of the Digital Signature

4.14.4. Pattern limitations

The keys must be provided in the payload in hex format when using the pattern. The keys can also be configured on the Digital Signatures node by the user directly or a specific msg parameter. The pattern is meant to be used by a single account as there is the possibility of leaking the private key, if shared with others.

4.14.5. Pattern publication means

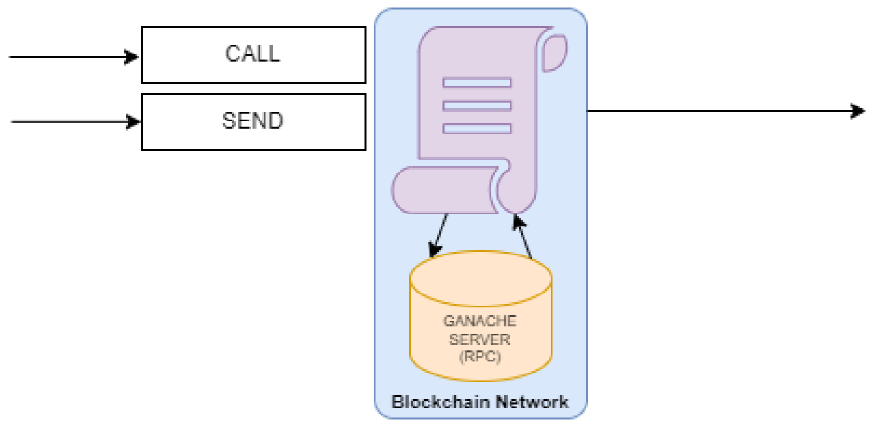
The pattern requires the node-red-crypto-blue [54] module installed and can be found in the node-red flow repository [55].

4.15. Smart Contracts Pattern

4.15.1. Pattern template description

The template description for the Smart Contracts Pattern appears in Table 26.

Table 26: Template Description for the Smart Contracts Pattern

Pattern name	Smart Contracts
Relation to Requirements	Req-3.4-Smart Contracts, Req-3.4 Smart Contract Templates
Source of pattern	All UCs
Pattern Context	Smart Contract functionalities blockchain development and integration.
Pattern Underlying Problem	Interacting with smart contracts, enhancing the flow with blockchain transactions.
Pattern Usefulness/ Objective	Provide an easy service to develop and interact with Smart Contracts.
Schema	
Prerequisites	Requires an RPC server to be running, a Smart Contract ABI and Bytecode configured in the pattern and a web3 account accessible by the RPC.
Condition of application	N/A
Input parameters	Contract function name, corresponding function arguments and .

Output parameters	Contract data or transaction, based on the operation requested.
Included functionality	Call, send and transact with contract functionalities.
Pattern limitations	Both Smart Contract and web3 accounts must be accessible by the RPC server.
Linked to Pattern	Generic use, blockchain smart contract development
Indicative Domains of Applicability	Distributed Ledger Technology, Smart Contract development & integration

4.15.2. Pattern implementation details

The pattern is implemented with two service endpoints using the available nodes of the node-red-contrib-web3-blue package that utilizes the web3 module for smart contract and RPC interactions. The pattern should be used through an http call either for viewing data in the smart contract (/call) or sending data to the smart contract (/send), invoking the corresponding function of the smart contract. Both services require an RPC server, a web3 account and a deployed contract in place in order to request the services from other flows.

4.15.3. Pattern examples of usage

Users can request or send to deployed smart contracts in a simple manner.

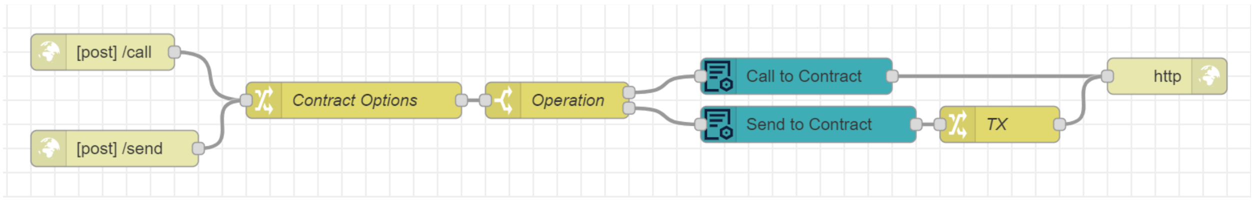


Figure 146: Smart Contract Service Implementation

The "Contract Options" change node takes care of properly aligning the contract node parameters from the provided payload and then, depending on the URL method invoked forwards to the corresponding contract action to perform.

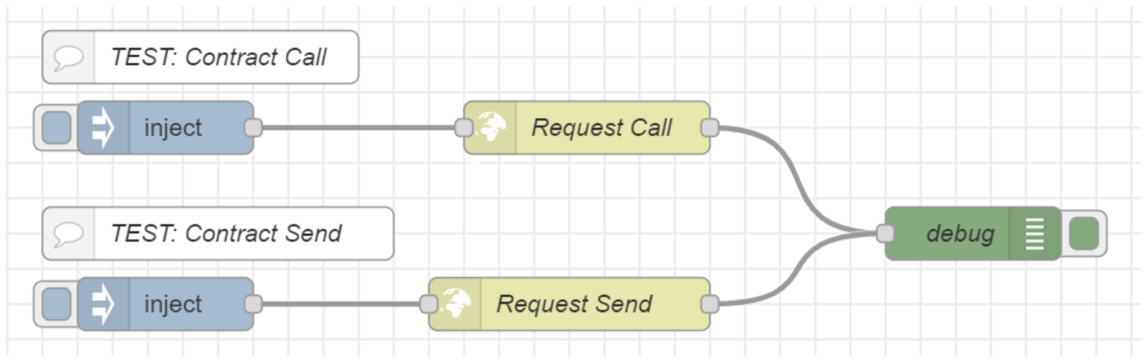


Figure 147: Test Flows for Invoking the Contract Server

The pattern requires at least one Smart Contract to be deployed on the RPC server.

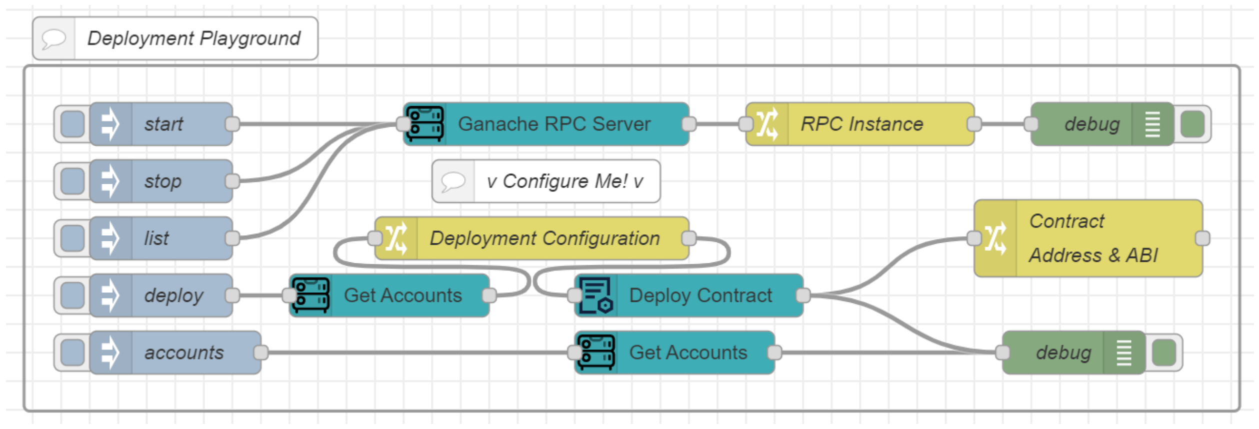


Figure 148: Deployment Playground of Smart Contracts Pattern

Alongside the injector testing nodes, a deployment playground is also provided so that users can easily create an RPC server instance ,deploy smart contracts and configure their own contracts.

4.15.4. Pattern necessary adaptations

Users should modify and configure their ABI, bytecode and constructor arguments of their Smart Contracts and then deploy it so that they can access it from the service endpoints.

4.15.5. Pattern limitations

Pattern requires an active RPC instance to be running and reachable and an account that can be used by that RPC. The pattern comes with a Deployment Playground to help the user in the contract deployment process but can only store context for the latest deployed contract.

4.15.6. Pattern publication means

The pattern requires the node-red-web3-blue [56] module installed and can be found in the node-red flow repository [57].

5. Gamification Approach for PHYSICS Training

Incorporating gaming elements into software training, also known as "gamification," is a process that involves the application of game mechanics and artefacts into non-game environments, with the purpose of enhancing the learning experience and practical knowledge earned. Due to the interactive nature of this method, retaining of information is generally improved. Users gain from immediate feedback, which allows them to quickly correct errors and comprehend concepts. In addition, gamification can foster a healthy sense of competition and provide flexibility, enabling users to learn at their own pace. These elements can convert learning from a potentially tedious process into an enjoyable and motivating experience.

However, gamification is not without challenges as there is a possibility that users will prioritise acquiring rewards over truly comprehending the content, leading to superficial learning. The inclusion of gaming elements may not appeal to everyone, and some may find them distracting and creating a balanced gamified experience can be more difficult and expensive than conventional methods. Additionally, if game elements are not aligned with learning objectives, they may offset the importance of the training content.

Gamification can be an effective way to engage and motivate learners, especially when traditional software training methods are perceived as difficult to follow or boring. In the context of PHYSICS, the development of game mechanics can facilitate quicker onboarding to the project's artefacts and patterns with increased engagement and better familiarity with the Node-RED ecosystem. To this end, a game server was developed, employing the necessary mechanics to create a gamified learning environment for training. The game is served as a web page application with options for local mode for training and development, online mode enabling online competitions and a future planned flow/subflow sharing capabilities through a marketplace.

5.1. Game Mechanics

The developed game server employs several game mechanics to make the training experience more motivating and engaging. These include the following mechanics:

- Points: Users can earn points by completing objectives in the game. These points can be used to unlock hints for an objective, in case the user is stuck on an objective, as well as other objectives that might have an unlock cost in order to play them.
- Achievements: Users can earn achievements by completing objectives in the game. These achievements include a description of what the user has accomplished to get them and optionally points and a hint on what to do in order to gain the achievement.
- Levels: Users can earn levels (with total earned points counting as experience) by completing objectives in the game. The levels allow the user to track their progress and knowledge of the training subjects.

- Avatars: When registering for online mode the user has a choice between several avatars. The avatars are bound to the user's account while they gain points and level up, allowing for a more personalised experience when playing the game.
- Badges: Users can earn badges by completing objectives in the game. These badges include a title , which the user can equip for their avatar, and optionally points and a hint on what to do in order to gain the badge.
- Objectives: Each objective requires the user to create a certain flow to complete it. Objectives aim at specific training tasks and can be rewarded with points, achievements, level or badges.
- Storylines: Users are presented with storylines that are composed of several objectives each. A storyline is a compact training course, where the user is presented with several tasks to complete on a specific training subject of a pattern, module or aspect of flow-programming.

5.2. Game Design & Implementation

The gamification approach encompasses various elements, such as the integration of the Node-RED visual environment and the monitoring of the user's advancement inside the game. The Figure 149 illustrates the architectural components of the gamification approach:

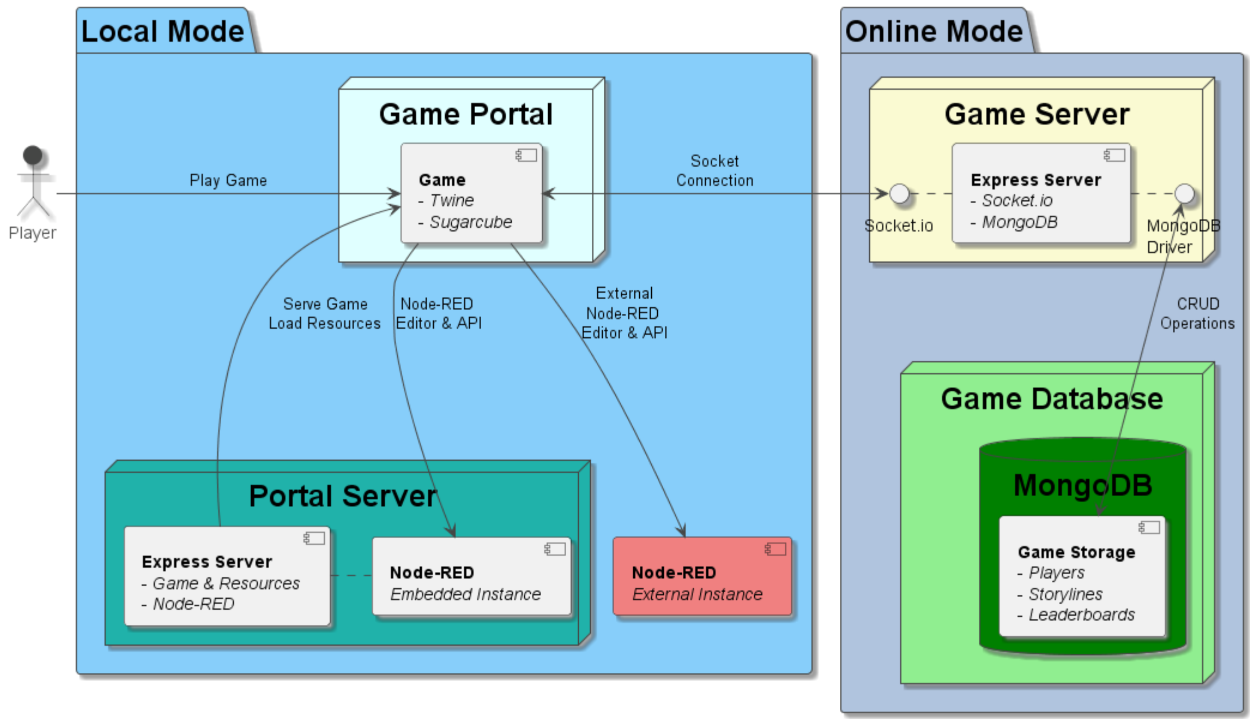


Figure 149 :Gaming Server Architecture

The gamification architecture is comprised of Local and Online Mode with the following components:

- Local Mode: The necessary components for playing the game locally.

  - Portal Server: The Portal Server consists of two components, a NodeJS Express server that hosts the Game Portal and an embedded instance of the Node-RED environment that is accessible from the Game. The Portal Server also provides the Game with the necessary game resources for loading the user interface backgrounds, icons, and tutorials.
  - Game Portal: The Game is served as an HTML page developed with Twine, an open-source tool for creating interactive non-linear stories, and SugarCube as the JavasScript library that handles elements of the UI and the story logic.
  - External Node-RED: The Game can be configured to use an external Node-RED instance along with any required credentials or authentication token.
- Online Mode: The necessary components for playing the game online with other users.
  - Game Server: The Game Server consists of a NodeJS Express server that receives and manages socket connections from the Game and connects to the Game Database to perform CRUD operations on the data stored in the Game Storage..
  - Game Database: The Game Database stores the document records of registered Players, available Storylines, and online Leaderboards, which are transmitted to the Game via the Game Server socket connection.5.3. Game Storyline Definition.

Storylines incorporate a gamified training experience for the user to develop their flow-programming skills, with the purpose of increasing engagement and motivation for the task at hand. In this gamification context, storylines are training modules pertaining to a particular pattern, flow, or subflow that require users to complete a number of objectives in order to finish the storyline. Each storyline is defined in a *.yaml* file containing the objectives, each with its own hints and steps to complete it, the rewards for completing objectives, the rules for checking that a user has completed an objective or is eligible for a reward and the resources (*flows, subflows, modules, payloads, files*) required to be loaded for the storyline.

5.3. Game Storyline Structure

A storyline can be structured from a *.yaml* file adjacent to a "resources" directory that contains any external resources for the storyline. The file structure of a test storyline is shown (Figure 150):

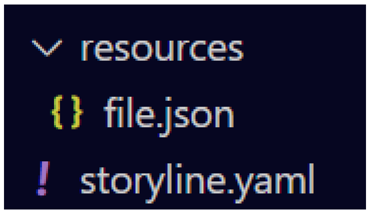


Figure 150: File structure of a test storyline

The *.yaml* storyline file structure is comprised of the *name*, *description*, *rules*, *rewards*, *resources* and *objectives* fields shown in Figure 151:

```
name: Test Storyline
```

```
description: A description for a test storyline with 2 objectives

objectives:
 - name: Objective 1
   description: Objective 1 Description
   rule: Rule A
   cost: 0
   rewards:
     - 100 Points
 - name: Objective 2
   description: Objective 2 Description
   rule: Rule B
   cost: 50
   steps:
     - name: Step 1
       step: Do this!
     - name: Step 2
       step: Do that!
   rewards:
     - Test Badge

rules:
 - name: Rule A
   success: Hello Back
   failure: Try again
   condition: {“in”:[{“var”:”message”},“Hello World”]}
 - name: Rule B
   success: Well Done!
   failure: Not Okay
   condition: {“==”:[{“var”:”number”},100]}

rewards:
 - name: 100 Points
   type: Points
   points: 100
 - name: Test Badge
   type: Badge
   title: Beginner
   points: 50

resources:
 - name: Flow#1
   type: Flow
   label: Inject Flow
   nodes:
        - {id: 037cd75a1699311e ,type: debug ,z: 29275a5bc3bc6851 ,name: debug ,active: true ,tosidebar: true ,console: false ,tostatus: false
,complete: payload ,targetType: msg ,statusVal: "" ,statusType: auto ,x: 390 ,y: 140 ,wires: []}
     - {id: 66641ea2455a1c26 ,type: inject ,z: 29275a5bc3bc6851 ,name: "", props:[{p: payload}, {p: topic, vt: str}], repeat: "", crontab: "", once: false,
onceDelay: 0.1, topic: "", payload: "", payloadType: date, x: 200, y: 140, wires:[["037cd75a1699311e"]]}
   objective: Objective 1
 - name: Payload#1
   type: Payload
   payload:
     number: "15"
```

```
- name: File#1
  type: File
  file: file.json
```

Figure 151: Template Structure of a Gaming Scenario

5.4. Game Storyline Development

The Local Mode of the Game enables the user to create their own storylines by designating the following properties and attributes:

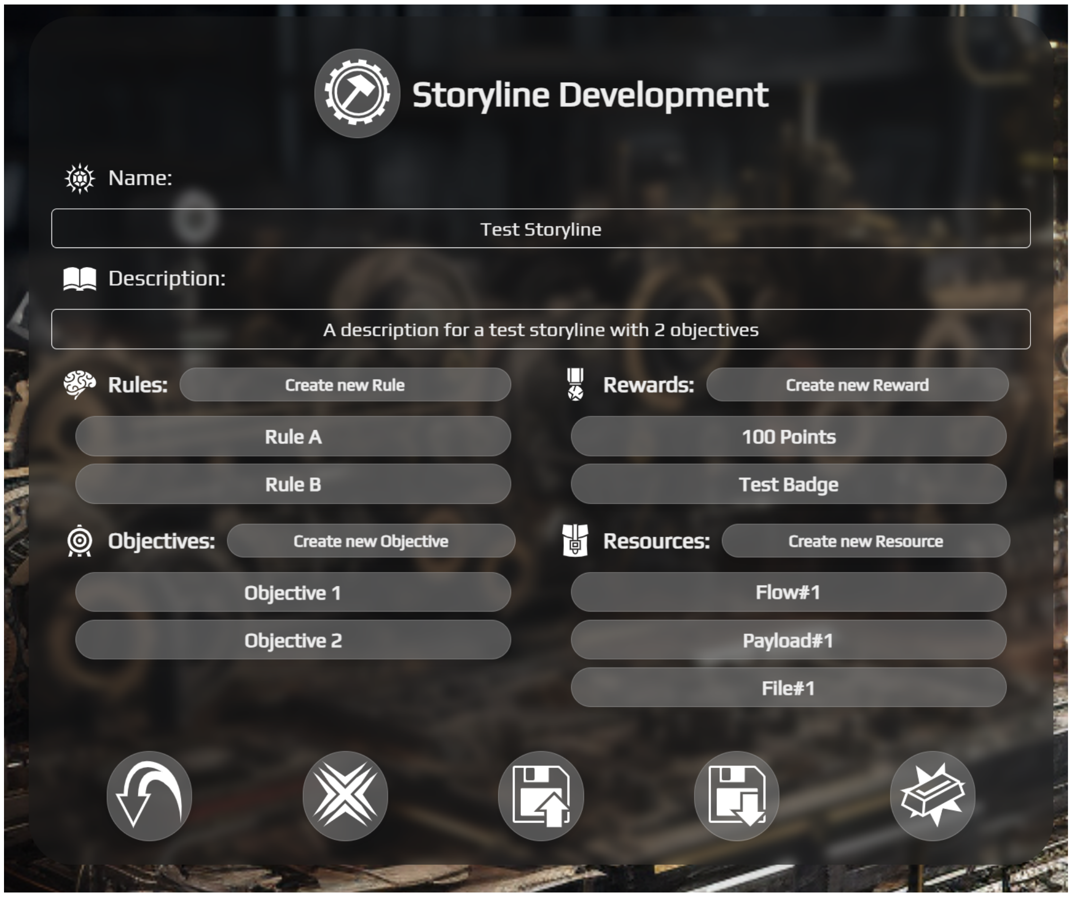
- Name: Name of the developed storyline.
- Description: Description of the developed storyline.
- Rules: An array of the game logic mechanisms, rules determine the condition by which an objective can be considered complete or by which a player is eligible for a reward.
- Rewards: An array of rewards such as points, achievements, badges or unlockables.
- Objectives: An array of objectives, defining the training tasks of the storyline. These objectives can be accompanied by steps on how to complete them and hints that can cost points to unlock.
- Resources: An array of resources required by the storyline such as flows, subflows, module, data or files.

Through the "Storyline Development" option (Figure 152), the user is able to create and testplay storylines. These storylines can then be exported into their respective.yaml files and bundled with any resources that have been defined. The development of the storyline also includes a testplay operation for the user to test their storyline by playing through it to see how it turns out.

5.5. PHYSICS Storylines

In the context of the PHYSICS project, a storyline was developed for the 2nd PHYSICS HUA Hackathon using the OpenWhisk (OW) Sliding Window subflow as shown in Figure 153. It included two objectives which allowed the user to interact with the OW Sliding Window and were guided through the steps on how to import and use the subflow. The purpose of this storyline was to enable the user to play with the subflow through game-based objectives to make the interaction more hands-on and engaging.

As for all other patterns and subflows found in this document, it is possible to develop and create similar storylines for the purposes of training users on how to use and exposing them to the project's artefacts in a gamified manner. These storylines could be shared using “Online Mode” enabling other users connected to play them, collaborate to develop new ones or create competitions with each other.


Storyline Development

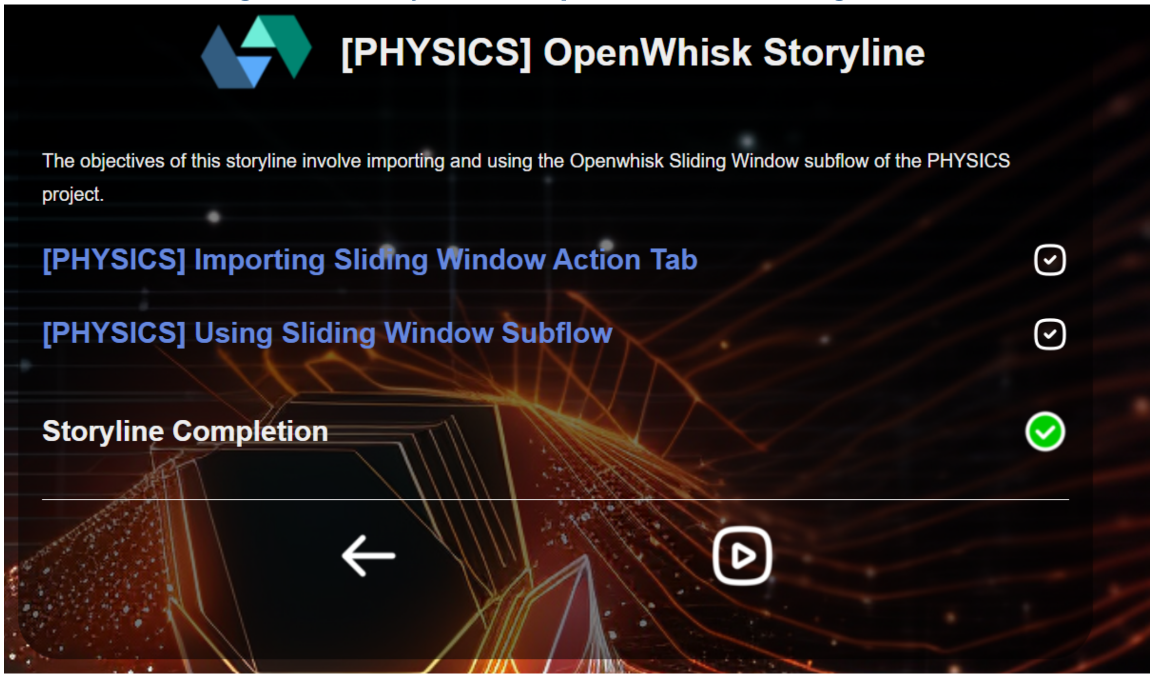
Name:

Test Storyline

Description:

A description for a test storyline with 2 objectives

Figure 152: Storyline Development UI in the Gaming Server


[PHYSICS] OpenWhisk Storyline

The objectives of this storyline involve importing and using the Openwhisk Sliding Window subflow of the PHYSICS project.

[PHYSICS] Importing Sliding Window Action Tab
[PHYSICS] Using Sliding Window Subflow
Storyline Completion

Figure 153: Execution of the Gaming Server for the Defined Storylines

5.6. Future Steps

Users can access Local Mode in the gaming server's current implementation, where they can play beginner storylines with simple objectives, interact with Node-RED tutorials and template nodes, and construct their own storylines. The next phase of the gaming server is to enable Online Mode so that users can interact, exchange, and trade storylines with each other, in which a flow sharing marketplace is a planned feature for the next iteration. Users could benefit by trading and sharing their storylines and flows through a digital marketplace in the form of Non-Fungible Tokens (NFTs), allowing for monetization of the knowledge, expertise and skills of the Node-RED environment. Next steps for this gamification strategy include testing and proofing the game mechanics, developing and deploying storylines, and evaluating the efficacy of the employed game mechanisms. These evaluations will clarify whether this gamified approach is effective for the PHYSICS project and could be utilised as an alternative training method.

6. Adaptive Elasticity Controllers Implementation and Incorporation

6.1. Introduction- Scope

The elasticity controllers are components integrated at the infrastructure layer by extending the existing Kubernetes API. The main target for these controllers is to account for additional application requirements that are not simply based on CPU and Memory metrics, but also consider/support metrics that help with response time (event driven computation), green computing (using less energy) or saving deployment cost (using cheaper instances). To accomplish that, the elasticity controllers need to use a broader view of the cluster and look at a wider set of metrics than in the standard Kubernetes auto-scalers where only CPU and Memory was being used. In addition, they need to provide interfaces for the other WP3 components, so that they can be configured accordingly for each use case.

In the initial phase of the project we evaluated the existing mechanisms, specially focusing on Kubernetes Vertical and Horizontal pod auto-scalers[33]. These controllers were looking at the existing known per pod metrics and could take decisions based on the load on the specific pod instances. However, this was not sufficient for modern workloads, specially for FaaS ones, and in PHYSICS we looked into extending it 3 ways:

- Add per pod custom metrics to monitor application specific metrics which can affect the elasticity policy. This is based on the K8s Scaling on custom metrics[34] feature.
- Add scaling which is based on multiple metrics, not only one. This way we can evaluate multiple configurations and take the maximal one. This is based on the K8s Scaling on multiple metrics[35] feature. Note: This policy may not suit our goal directly since it takes the "maximal" configuration, so we may need to extend it or use one of the alternatives.
- Add a global view which can affect the elasticity policy based on all the information in the monitoring system, and on cluster deployment metadata (such as energy consumption, price, topology and network usage). This additional metadata is modelled in the semantic model so it can be gathered and accessed by any PHYSICS component (such as the elasticity controllers). This is based on the K8s Support for metrics APIs[36] feature.

After starting working on what metrics to make available (mainly from the OpenWhisk side) and how to consume this, we discovered the upstream project Kubernetes-based Event Driven Autoscaler (KEDA)[37]. This project is much more aligned with what PHYSICS wanted to address, as it focuses on Event-Driven actions, which aligns better with FaaS use cases. With KEDA the container scaling is based on the number of events to be processed, rather than CPU or memory thresholds. In addition, it is fully integrated with Kubernetes (through CRDs, the same approach we followed for PHYSICS components, as detailed in D5.2), lightweight, and works alongside the standard

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

[34] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#scaling-on-custom-metrics

[35] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#scaling-on-multiple-metrics

[36] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#support-for-metrics-apis

[37] https://keda.sh/

Kubernetes components like the HPA. It provides a nice way of extending its functionality without overwriting or duplication.

We studied this new project evolution, including evaluation of its community (https://keda.sh/community/) and features, and decided to adopt it as our API. The main benefits are:

- Don't reinvent the wheel, and reuse what is already done. This makes us saving a lot of time and effort, as well as increasing the chances of making a bigger impact by contributing to it.
- KEDA allows explicitly mapping the apps you want to manage (i.e. scale) in an event-driven way.
- KEDA provides a catalogue of autoscalers, ready to be used and or to be (re)configured for specific use cases. This means we have a large set of autoscalers already available for us. See section 6.4 for more details.

6.2. Relation to requirements

One of the goals of the PHYSICS project is to enable a simple environment for developing SaaS applications which can run efficiently across the cloud continuum from the core of the cloud to the edge. The efficiency can be measured in multiple axis, for example:

- Time - the application (or the workflows) should complete the execution according to some time constraints (99.9% of the flows should complete within 800 ms). In some cases, this may not leave too much freedom in selecting the resources on which the flow executes, but in some cases it means we should not use the fastest possible resources and instead use other resources which consume less power, cost less, reduce network bandwidth or a combination of all -- this highlights the need for using other metrics different than cpu and memory for taking the scaling decisions.
- Power - when we want to optimise flows for power consumption, we can use resources which use less power as explained above, but other techniques may involve reducing network bandwidth (which can be a power consuming hog, especially when using cellular networks in rural areas which are not covered in high density -- this also highlights the need for other inputs, as well as the work in Kepler related to energy metrics detailed in D5.2.
- Cost - in some cases (especially when using public clouds) we can choose resources with different pricing schemes and different QoS plans - in cases where high QoS is not mandatory (for example for very short executing flows or functions) we can select cheaper resources which reduce the cost of ownership and keep the QoS at the user needs -- this again highlights the need for different metrics and specific purpose scalers that account for them.

The goal of the adaptive elasticity controller is to be able to take all such considerations into account when deciding to scale the applications and even the system itself (i.e., scaling the Kubernetes cluster to for instance save energy or cost).

6.3. Component/Subsystem Design

The main flow of operations of elasticity controllers is depicted in Figure 154, where we detect (or predict) changes in the workload or the system (Event-Drive) which triggers the autoscaler logic.

This logic is in charge of deciding the actions to be taken in response to those events, and trigger the new configuration -- either directly or through other controllers.

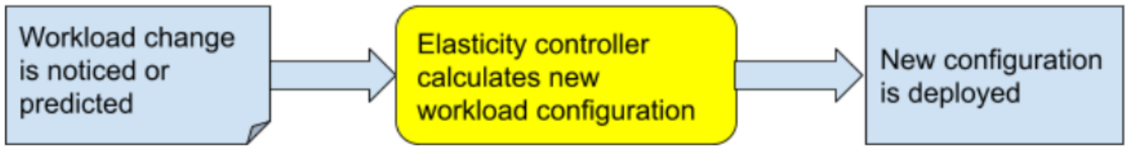

Workload change is noticed or predicted
Elasticity controller calculates new workload configuration
New configuration is deployed

Figure 154: Elasticity Controller Flow of Operations

As described, the elasticity controllers need to have access to several data sources. The initial design (depicted in Figure 155) already accounted for this, where the next different sources of information were being used by the elasticity controller to create the new deployment plan and leverage the K8s horizontal pod autoscaling APIs.

- Existing cluster configuration and the workloads on each node. This comes from K8s cluster manager and from the statistics in Prometheus.
- Time estimation for running functions on different platforms. This information is a combination of information from the semantic model and the statistics gathered by the platform
- Power estimations of different platforms (compute and network). This information comes mainly from the semantic models, obtained though Kepler (more information in D5.2)
- Cost of different nodes in the cloud - this information would come from the cloud provider.

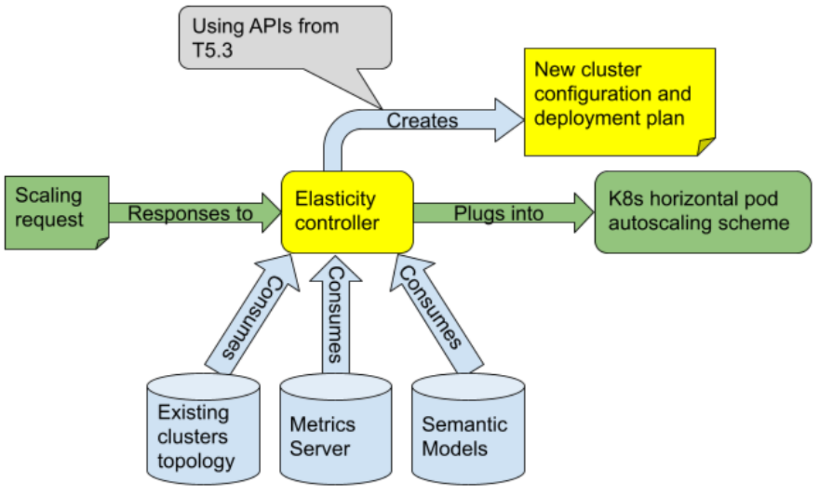


Figure 155: Elasticity Controller Relation to PHYSICS Architecture

As highlighted before, in the second phase of the project we shifted the design and adopted the KEDA project as the main framework for elasticity controllers. We took this decision due to:

- Project being perfectly aligned with PHYSICS intention about supporting
    - Event driven scaling decisions
    - Kubernetes integration

- Configurable options for each use case -- provided through the Kubernetes integration
- The architecture of KEDA (see Figure 156) mapping really well with our initial design (Figure). As you can see the concepts are very similar:
    - There is an external event which triggers the scaler.
    - It is integrated with Kubernetes through the ScaleObject CRD
    - It grabs (and provides) metrics that integrate (fed) the Horizontal Pod Autoscaler

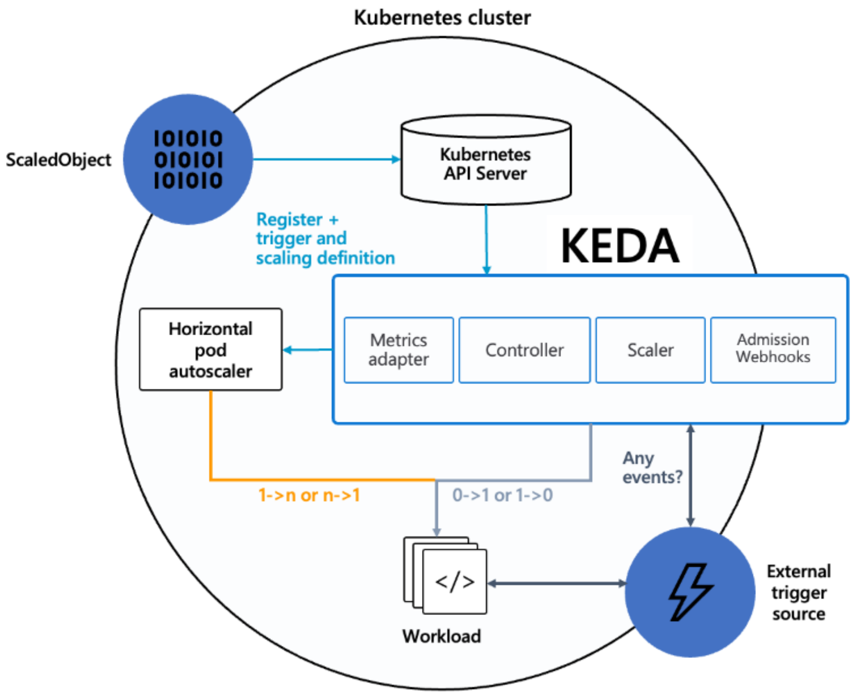


Figure 156: KEDA Architecture (taken from https://keda.sh/docs/2.11/concepts/#architecture)

6.3.1. KEDA overview

KEDA extends the Kubernetes API through Custom Resource Definitions. This new object, named ScaledObject, is offered as the new API for the end users to configure the scaler to use and the resources to scale based on it.
KEDA has 4 main components which perform the key roles within Kubernetes to manage scaling:

- The "Scaler" component is the specific scaler to use.
- The "Controller" (which corresponds to the keda-operator) is in charge of scaling to and from zero when there are no events.
- The "Metrics adapter" is in charge of exposing rich event data (e.g., queue length or stream lag) to the Horizontal Pod Autoscaler to drive the scale out from 1 to N.
- The “Admission Webhooks” are in charge of ensuring proper configuration is applied such as preventing multiple Scale Objects pointing to the same target.

6.3.2. Benefits from using it

There are several benefits due to using KEDA as the base for our elasticity controllers:

- It is event-driven
- It has a large catalogue[38] (50+) of build-in scalers, and it is easily extensible to add more
- It allows to manage different type of workloads as it can manage deployment, jobs and even custom resources
- It allows scaling to 0, with it is a plus for saving energy
- Great community behind it

6.3.3. Elasticity controllers options

Thanks to the above explained features and benefit from KEDA, we designed a different set of autoscalers that can be implemented thanks to the information made available by other PHYSICS components.

Table 27: Indicative Scaler Strategies

Scaler	Optimization function	Description	Scaling Object	Metrics used
Scaler 1	Energy	All KEDA scalers allow scaling to 0, which optimises energy consumption when there is no events	Pods (deployments)	No events
Scaler 2	Energy	If the amount of energy usage is high but resource usage is not that high, it can be scaled down to save some energy and compact the pods, as usually higher cpu usage is more efficient than low (base on KEDA Prometheus scaler[39]	Nodes	Kepler, Prometheus, Semantic
Scaler 3	Cost	Scale the number of nodes of the cluster depending on the cost of the (spot) instances. Set a target maximum and automatically scale between a min and max amount, depending on cost over time	Node	External API (cloud provider costs)
Scaler 4	Performance	Scale the number of nodes of the cluster depending on the number of functions over time (based on KEDA kubernetes workload scaler [])	Node	Number of functions
Scaler 5	Performance	Scale the number of OpenWhisk invokers depending on the number of functions over time (based on KEDA kubernetes workload scaler too)	OpenWhisk Invokers	Number of functions

[38] https://keda.sh/docs/2.11/scalers/
[39] https://keda.sh/docs/2.11/scalers/prometheus/

Scaler 6	OpenWhisk waiting (queue) time	Scale the number of OpenWhisk invokers depending on the length of the OpenWhisk kafka queue, so that there are more consumers creating pods for functions (based on KEDA Apache Kafka scaler).	OpenWhisk Invokers	Kafka messages
Scaler 7	OpenWhisk waiting (queue) time	Similar to the previous one. Scale the number of OpenWhisk invokers depending on the OpenWhisk kafka delay start metric, so that there are more consumers creating pods for functions or more nodes for them (based on KEDA Prometheus scaler).	OpenWhisk Invokers or Nodes	OpenWhisk metrics from Prometheus[40]

6.4. Component Implementation

Out of the above list, we have implemented the next 2 as part of the PHYSICS project (Scaler 4 and 6), due to being more OpenWhisk specific. Note the algorithms themselves for the other could be pretty straightforward though some more advanced techniques could be implemented for more specific purpose applications -- such as using the available data together with Machine/Reinforce Learning techniques for taking more predictive actions (instead of reactive).
KEDA was installed in the cluster through the Custom Metrics Autoscaler[41] which is built on top of the OpenShift Container Platform horizontal pod autoscaler (HPA).

6.4.1. Scaling Kubernetes cluster depending on number of OpenWhisk functions (Scaler 4)

This scaler is based on the Kubernetes Workload but targeting nodes instead of pods as the entity to scale. The main idea of this scaler is to scale the number of Nodes depending on the number of functions. More specifically, we want to make sure the memory is sufficient, so it is adjusted in such a way that the image side used for the nodes is considered.
In our (AWS) environment, based on OKD, we use ClusterAPI to manage/create/scale the cluster nodes. This defines the nodes as a set of machines, which can be easily scaled, in the same way as you scale the number of pods of a replica set:

```
$ oc get machinesets -A
NAMESPACE              NAME                                DESIRED   CURRENT   READY   AVAILABLE
```

[40] https://github.com/apache/openwhisk/blob/master/docs/metrics.md
[41]https://docs.openshift.com/container-platform/4.13/nodes/cma/nodes-cma-autoscaling-custom-rn.html

```
AGE
openshift-machine-api   ocphub-t4rh8-submariner-gw-eu-north-1a  1        1        1        1
         643d
openshift-machine-api   ocphub-t4rh8-worker-eu-north-1a          4        4        4        4
         647d
```

Code 8: Replica Set Details

Thus, in our case the object to scale is the machines belonging to the machineset ocphub-t4rh8-worker-eu-north-1a. Note the information about the CPUs and Memory for the AWS flavour used are also in there

```
$ oc get machinesets ocphub-t4rh8-worker-eu-north-1a -n openshift-machine-api -oyaml | head
apiVersion: machine.openshift.io/v1beta1
kind: MachineSet
metadata:
  annotations:
        machine.openshift.io/GPU: "0"
        machine.openshift.io/memoryMb: "16384"
        machine.openshift.io/vCPU: "4"
```

Code 9: Target of Scaling

Then, the trigger specification[42] must be configured appropriately based on the default memory associated with functions (OW default configuration parameter) and the memory available per node -- also considering other workloads/status of the cluster to better accommodate. For instance it is not the same cluster that is dedicated for FaaS, that another one that is shared for different use cases. Here we focused on the first one.

The main value to adjust is "value", which is the relation between the number of pods which match a specific selector (i.e., label) and the number of pods of the workload to scale. In our case the podSelector/labelSelector points to the OpenWhisk generated pods, and the workload to scale is the above mentioned machineset (i.e., a Kubernetes CRD).

So, in our case we forced a specific label in the OpenWhisk created pod. Note this is directly applicable for Knative ones too. In fact, it can even be applied to both if the same label is used -- the scaler will account for functions/pods created by either OpenWhisk and Knative.
Then we defined the scaleTargetRef (the object to scale), pointing to the desired machine set as the scaler supports scaling custom resources as long as they implement the scale subresource, and this is the case for machinesets.

Finally, we decide on the right value for "value" and the scale limits. We want to always have machines ready (so no scaling to 0), as there are other workloads/pods in the system, such as PHYSICS components, and we also wanted to have a limit (as we are paying for the AWS servers and

[42] https://keda.sh/docs/2.11/scalers/kubernetes-workload/#trigger-specification

we want to limit the costs). In our case we set a limit between 3 and 6. And then selected the right target value for us, which was around 15. This means we target to have up to 15 functions per node, to ensure we don't hit memory issues.

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: function-node-scaler
spec:
  scaleTargetRef:
    name: ocphub-t4rh8-worker-eu-north-1a
    apiVersion:  machine.openshift.io/v1beta1
    kind: MachineSet
  minReplicaCount: 3
  maxReplicaCount: 6
  triggers:
  - type: kubernetes-workload
    metadata:
      podSelector: 'function=physics'
      value: '15'
```

Code 10: Setting of Functions per Node Elasticity Metric

6.4.2. Scaling OpenWhisk invokers or Kubernetes cluster depending on Kafka queues (Scaler 6)

The main target metric within the KEDA Apache Kafka scaler is the queue lag (or queue size), the number of messages the consumer is behind the producer. Let us picture a scenario in which some OpenWhisk Kafka producers produced 10000 messages into a partition and some OpenWhisk Kafka consumers consumed 9800 of them. In that case, the consumer lag is 200 messages.
In this scenario the KEDA Apache Kafka scaler trigger specification[43] (when to scale) points to the OpenWhisk Kafa instance and corresponding topic. The lag threshold can be adjusted to make the auto-scaling more or less granular and reactive. The scale target reference (what to scale) points to OpenWhisk invokers deployment. By setting the minimum replica count to 0 the deployment will consume no resources in the absence of function invocations to favour energy savings.
By default, the number of replicas will not exceed:

- The number of partitions on a topic when a topic is specified;
- The number of partitions of all topics in the consumer group when no topic is specified

The resulting KEDA Scaled Object. Note that the topic is intentionally unspecified, in order for total offset lag to be calculated considering all topics within the consumer group. By default, OpenWhisk creates one topic per invoker.

```
apiVersion: keda.sh/v1alpha1
```

43

```
kind: ScaledObject
metadata:
  name: kafka-openwhiskinvoker-consumer-scaledobject
spec:
  minReplicaCount: 0
  maxReplicaCount: 5
  cooldownPeriod: 5
  pollingInterval: 10
  scaleTargetRef:
    name: owdev-invoker
     apiVersion: apps/v1
     kind: StatefulSet
 triggers:
  - type: kafka
    metadata:
    ootstrapServers: owdev-kafka.openwhisk.svc:9092
    consumerGroup: invoker0
    lagThreshold: '5'
    offsetResetPolicy: 'latest'
```

Code 11: Setting of Elasticity Metric based on Queue Size

Note developing Scaler 7 will be pretty similar to this one, but looking at delay at the kafka queue, obtained through prometheus (openwhisk.histogram.kafka_<topic name>.delay_start - Time delay between when a message was pushed to Kafka and when it is read within a consumer), instead of being triggered by the queue length.

7. Conclusions

This deliverable presents the work done in WP3 from M4 to M33. The main goal of the first period (M4-M13) was to go from the initial design in WP2 to a detailed set of processes that are needed for establishing the Visual Design Environment, the entry point for the application development in PHYSICS, and the first component prototypes of WP3. This includes the ability to support and implement the main design, development and deployment process of an application, starting from the Node-RED visual editor and concluding to producing the needed deployment artefacts as well as the definition of the application graph to be forwarded to WP4 for the actual deployment. The specific process has been supported by relevant semantics as well as decoupled DevOps process pipelines, in order to create a modular and extensible environment covering aspects such as image building, custom image uploading, performance analysis of functions.

The goal of the second period (M14-M33) included the testing and stabilisation of the environment, as well as incorporation of user feedback and extension to new features and functionalities. At the end of the project, the PHYSICS Design Environment includes many new functionalities, including centralised login, enhanced visualisations and testing modes, increased logging and build management processes, along with a migration to a more centralised, SaaS-like design.

Furthermore, the PHYSICS Design Environment offers a set of extended functionalities, coming from the semantic domain, in order to empower the developer to easily declare a number of parameters and options that may affect the placement, deployment, operation and configuration of their applications. Two different means of embedding semantics have been designed, through code or flow level annotators, that accompany the main semantic structure of the workflow specification of Node-RED. A relevant ontology has been also defined in order to model the PHYSICS related concepts at the application side, integrating with the Reasoning Framework of T4.1.

The Design Environment comes with a package of PHYSICS developed patterns that can be reused and parameterized based on the developer needs and application goals. The patterns aim to aid the developers in wrapping code around the Openwhisk specification, utilising the Node-RED environment as an application level orchestrator, handling context in different ways, optimising function execution, providing anonymization and security functionalities among others, as well as doing an easier transition to a function-oriented implementation principle. The patterns have also incorporated aspects like routing capabilities, cluster monitoring means, automated annotation processes as well as optimization functions available at the application level. Pattern creation and development has been performed by taking into account from early on the application needs and expressed scenarios in D6.3, so that there is a higher level of guarantee that they will be valuable to the use cases of the project, as well as feedback from them following the end of the first iteration of the project in M18. Furthermore, presentation, configuration and usage examples for each pattern are given as a guide for the developers, as well as limitations, potential problems and performance tradeoffs. These patterns can be used not only within the PHYSICS project but also in the wider Node-RED community, since they represent functionalities that are needed in multiple scenarios and can be dragged and dropped in any Node-RED environment.

For the adaptive elasticity controllers, an approach based on the KEDA tool has been extended and adapted to the FaaS paradigm, including the ability to define scalability actions based on different metrics and events. As a conclusion the PHYSICS Design Environment is a solution that can significantly abstract and automate a number of processes and tasks relevant to the usage and operation in a FaaS manner. The level of abstraction achieved is considered critical for interacting with the envisioned end users of the tool, targeting primarily professionals that may be in the IT domain but do not have a strong development background (i.e. not targeting full-stack or back-end developers but focusing more on roles like data scientists, scientific engineers etc). Furthermore, the integration and streamlining of different processes (e.g. Annotations, performance analysis etc) aids in giving the necessary information to the following WPs (platform and infrastructure management) in order to successfully address the high level objectives of the project, in terms of optimised placement, management and operation of a FaaS cluster.

REFERENCES

[1] D. Neri, J. Soldani, O. Zimmermann, and A. Brogi, “Design principles, architectural smells and refactorings for microservices: a multivocal review,” *SICS Softw.-Intensive Cyber-Phys. Syst.*, vol. 35, no. 1–2, pp. 3–15, Aug. 2020, doi: 10.1007/s00450-019-00407-8.

[2] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice Architecture: Aligning Principles, Practices, and Culture*. O’Reilly Media, Inc., 2016.

[3] martinekuan, “Cloud design patterns - Azure Architecture Center.” Accessed: Sep. 22, 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/patterns/

[4] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakaroha, “A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms,” in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Dec. 2017, pp. 162–169. doi: 10.1109/CloudCom.2017.15.

[5] C. Abad, I. T. Foster, N. Herbst, and A. Iosup, “Serverless Computing (Dagstuhl Seminar 21201),” *Dagstuhl Rep.*, vol. 11, no. 4, pp. 34–93, 2021, doi: 10.4230/DagRep.11.4.34.

[6] G. Kousiouris and D. Kyriazis, “Functionalities, Challenges and Enablers for a Generalized FaaS based Architecture as the Realizer of Cloud/Edge Continuum Interplay:,” in *Proceedings of the 11th International Conference on Cloud Computing and Services Science*, Online Streaming, --- Select a Country ---: SCITEPRESS - Science and Technology Publications, 2021, pp. 199–206. doi: 10.5220/0010412101990206.

[7] P. Leitner, E. Wittern, J. Spillner, and W. Hummer, “A mixed-method empirical study of Function-as-a-Service software development in industrial practice,” *J. Syst. Softw.*, vol. 149, pp. 340–359, Mar. 2019, doi: 10.1016/j.jss.2018.12.013.

[8] F. Amato and F. Moscato, “Exploiting Cloud and Workflow Patterns for the Analysis of Composite Cloud Services,” *Future Gener. Comput. Syst.*, vol. 67, pp. 255–265, Feb. 2017, doi: 10.1016/j.future.2016.06.035.

[9] P. Giampa and M. Dibitonto, “MIP An AI Distributed Architectural Model to Introduce Cognitive computing capabilities in Cyber Physical Systems (CPS).” arXiv, Mar. 29, 2020. doi: 10.48550/arXiv.2003.13174.

[10] “Cloud-native, event-based programming for mobile applications | Proceedings of the International Conference on Mobile Software Engineering and Systems.” Accessed: Sep. 21, 2023. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/2897073.2897713?casa_token=7GsWSFzLbv4AAAAA:n3A5hCazri-EjEl5Or7Zh1pa1NWH5t3Dd6adTrzHPuVvxUT2U9sBzlgsKl9HSRpwD_e3qgN3ng

[11] “Apache OpenWhisk is a serverless, open source cloud platform.” Accessed: Sep. 21, 2023. [Online]. Available: https://openwhisk.apache.org/

[12] D. Barcelona-Pons, P. García-López, Á. Ruiz, A. Gómez-Gómez, G. París, and M. Sánchez-Artigas, “FaaS Orchestration of Parallel Workloads,” in *Proceedings of the 5th International Workshop on Serverless Computing*, in WOSC ’19. New York, NY, USA: Association for Computing Machinery, Dec. 2019, pp. 25–30. doi: 10.1145/3366623.3368137.

[13] S. Sengupta, “Faas-flow - Function Composition for OpenFaaS.” Sep. 18, 2023. Accessed: Sep. 21, 2023. [Online]. Available: https://github.com/s8sg/faas-flow

[14] E. Bisong, *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners*. Berkeley, CA: Apress, 2019. doi: 10.1007/978-1-4842-4470-8.

[15] “Workflows Tutorial (2nd gen) | Cloud Functions Documentation | Google Cloud.” Accessed: Sep. 22, 2023. [Online]. Available: https://cloud.google.com/functions/docs/tutorials/workflows

[16] “node-red-node-openwhisk.” Accessed: Sep. 22, 2023. [Online]. Available:

http://flows.nodered.org/node/node-red-node-openwhisk
[17] “Triggerflow | Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems.” Accessed: Sep. 21, 2023. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3401025.3401731?casa_token=KOeE394nR6UAAAAA:Yul7zGbfp7ShfzhOqOy7SvGlaoTumEh0VU4tn81Yu6E4F2BJBdU1h9W4kQzVwqnHLHKP2yp8XQ
[18] “Library - Node-RED.” Accessed: Sep. 22, 2023. [Online]. Available: https://flows.nodered.org/
[19] “Node.js,” Node.js. Accessed: Sep. 22, 2023. [Online]. Available: https://nodejs.org/en
[20] I. Baldini *et al.*, “The serverless trilemma: function composition for serverless computing,” in *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, in Onward! 2017. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 89–103. doi: 10.1145/3133850.3133855.
[21] S. Eismann *et al.*, “The State of Serverless Applications: Collection, Characterization, and Community Consensus,” *IEEE Trans. Softw. Eng.*, vol. 48, no. 10, pp. 4152–4166, Oct. 2022, doi: 10.1109/TSE.2021.3113940.
[22] M. Montagnuolo *et al.*, “Supporting media workflows on an advanced cloud object store platform,” in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, in SAC ’16. New York, NY, USA: Association for Computing Machinery, Apr. 2016, pp. 384–389. doi: 10.1145/2851613.2851620.
[23] “Nx: Smart, Fast and Extensible Build System,” Nx. Accessed: Sep. 22, 2023. [Online]. Available: https://nx.dev
[24] “Serverless Computing – AWS Lambda Pricing – Amazon Web Services.” Accessed: Sep. 22, 2023. [Online]. Available: https://aws.amazon.com/lambda/pricing/
[25] G. Kousiouris and A. Pnevmatikakis, “Performance Experiences From Running An E-health Inference Process As FaaS Across Diverse Clusters,” in *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*, Coimbra Portugal: ACM, Apr. 2023, pp. 289–295. doi: 10.1145/3578245.3585023.
[26] G. Kousiouris, T. Cucinotta, and T. Varvarigou, “The effects of scheduling, workload type and consolidation scenarios on virtual machine performance and their prediction through optimized artificial neural networks,” *J. Syst. Softw.*, vol. 84, no. 8, pp. 1270–1291, Aug. 2011, doi: 10.1016/j.jss.2011.04.013.
[27] A. Evangelinou, M. Ciavotta, D. Ardagna, A. Kopaneli, G. Kousiouris, and T. Varvarigou, “Enterprise applications cloud rightsizing through a joint benchmarking and optimization approach,” *Future Gener. Comput. Syst.*, vol. 78, pp. 102–114, Jan. 2018, doi: 10.1016/j.future.2016.11.002.
[28] V. Ivanov and K. Smolander, “Implementation of a DevOps Pipeline for Serverless Applications,” in *Product-Focused Software Process Improvement*, M. Kuhrmann, K. Schneider, D. Pfahl, S. Amasaki, M. Ciolkowski, R. Hebig, P. Tell, J. Klünder, and S. Küpper, Eds., in Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 48–64. doi: 10.1007/978-3-030-03673-7_4.
[29] A. Pogiatzis and G. Samakovitis, “An Event-Driven Serverless ETL Pipeline on AWS,” *Appl. Sci.*, vol. 11, no. 1, Art. no. 1, Jan. 2021, doi: 10.3390/app11010191.
[30] K. Figiela, A. Gajek, A. Zima, B. Obrok, and M. Malawski, “Performance evaluation of heterogeneous cloud functions,” *Concurr. Comput. Pract. Exp.*, vol. 30, no. 23, p. e4792, 2018, doi: 10.1002/cpe.4792.
[31] R. Cordingly *et al.*, “The Serverless Application Analytics Framework: Enabling Design Trade-off Evaluation for Serverless Software,” in *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, Delft Netherlands: ACM, Dec. 2020, pp. 67–72. doi: 10.1145/3429880.3430103.
[32] R. Pellegrini, I. Ivkic, and M. Tauber, “Function-as-a-Service Benchmarking Framework,” in

Proceedings of the 9th International Conference on Cloud Computing and Services Science, 2019, pp. 479–487. doi: 10.5220/0007757304790487.
[33] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, “Cold Start Influencing Factors in Function as a Service,” in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, Dec. 2018, pp. 181–188. doi: 10.1109/UCC-Companion.2018.00054.
[34] D. Jackson and G. Clynch, “An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions,” in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, Dec. 2018, pp. 154–160. doi: 10.1109/UCC-Companion.2018.00050.
[35] G. Fatouros *et al.*, “Knowledge Graphs and interoperability techniques for hybrid-cloud deployment of FaaS applications,” in *2022 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, IEEE, 2022, pp. 91–96.
[36] “openwhisk/docs/reference.md at master · apache/openwhisk,” GitHub. Accessed: Sep. 21, 2023. [Online]. Available: https://github.com/apache/openwhisk/blob/master/docs/reference.md
[37] S. Ristov, S. Pedratscher, J. Wallnoefer, and T. Fahringer, “DAF: Dependency-Aware FaaSifier for Node.js Monolithic Applications,” *IEEE Softw.*, vol. 38, no. 1, pp. 48–53, Jan. 2021, doi: 10.1109/MS.2020.3018334.
[38] “openwhisk/docs/actions-docker.md at master · apache/openwhisk,” GitHub. Accessed: Sep. 22, 2023. [Online]. Available: https://github.com/apache/openwhisk/blob/master/docs/actions-docker.md
[39] Y. Wang, H. Zhu, J. Wang, J. Liu, Y. Wang, and L. Sun, “XLBoost-Geo: An IP Geolocation System Based on Extreme Landmark Boosting.” arXiv, Oct. 26, 2020. doi: 10.48550/arXiv.2010.13396.
[40] G. Cretella and B. Di Martino, “A semantic engine for porting applications to the cloud and among clouds,” *Softw. Pract. Exp.*, vol. 45, no. 12, pp. 1619–1637, 2015, doi: 10.1002/spe.2304.
[41] A. V. Dastjerdi, S. K. Garg, O. F. Rana, and R. Buyya, “CloudPick: a framework for QoS-aware and ontology-based service deployment across clouds,” *Softw. Pract. Exp.*, vol. 45, no. 2, pp. 197–231, 2015, doi: 10.1002/spe.2288.
[42] P. Jamshidi, C. Pahl, and N. C. Mendonça, “Pattern-based multi-cloud architecture migration,” *Softw. Pract. Exp.*, vol. 47, no. 9, pp. 1159–1184, 2017, doi: 10.1002/spe.2442.
[43] C. Pahl, P. Jamshidi, and O. Zimmermann, “Architectural Principles for Cloud Software,” *ACM Trans. Internet Technol.*, vol. 18, no. 2, p. 17:1-17:23, Feb. 2018, doi: 10.1145/3104028.
[44] A. Hachemi, “Software Development Process Modeling with Patterns,” in *Proceedings of the 2nd World Symposium on Software Engineering*, in WSSE ’20. New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 37–41. doi: 10.1145/3425329.3425339.
[45] “A Process Pattern Model for Tackling and Improving Big Data Quality | Information Systems Frontiers.” Accessed: Sep. 21, 2023. [Online]. Available: https://link.springer.com/article/10.1007/s10796-017-9822-7
[46] D. Taibi, N. El Ioini, C. Pahl, and J. R. S. Niederkofler, *Patterns for serverless functions (Function-as-a-Service) : A multivocal literature review*. Science and Technology Publications (SciTePress), 2020. doi: 10.5220/0009578501810192.
[47] G. Kousiouris *et al.*, “Parametric Design and Performance Analysis of a Decoupled Service-Oriented Prediction Framework Based on Embedded Numerical Software,” *IEEE Trans. Serv. Comput.*, vol. 6, no. 4, pp. 511–524, Oct. 2013, doi: 10.1109/TSC.2012.21.
[48] A. Iosup, S. Kounev, and K. Sachs, “SPEC Research Group’s Cloud Working Group: RG Cloud Group,” in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, 2016, pp. 127–128.
[49] N. Mahmoudi and H. Khazaei, “SimFaaS: A Performance Simulator for Serverless Computing Platforms.” arXiv, Feb. 17, 2021. doi: 10.48550/arXiv.2102.08904.
[50] “OpenWhisk.” The Apache Software Foundation, Aug. 31, 2023. Accessed: Aug. 31, 2023.

[Online]. Available: https://github.com/apache/openwhisk
[51] G. Kousiouris, “A self-adaptive batch request aggregation pattern for improving resource management, response time and costs in microservice and serverless environments,” in *2021 IEEE International Performance, Computing, and Communications Conference (IPCCC)*, Oct. 2021, pp. 1–10. doi: 10.1109/IPCCC51483.2021.9679422.
[52] S. Matsubara *et al.*, “Digital Annealer for High-Speed Solving of Combinatorial optimization Problems and Its Applications,” in *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2020, pp. 667–672. doi: 10.1109/ASP-DAC47756.2020.9045100.
[53] “Get Started EN,” Fujitsu Deutschland. Accessed: Sep. 22, 2023. [Online]. Available: https://www.fujitsu.com/de/themes/digitalannealer/get-started/get-started-en.html
[54] “node-red-contrib-crypto-blue.” Accessed: Sep. 22, 2023. [Online]. Available: http://flows.nodered.org/node/node-red-contrib-crypto-blue
[55] “Crypto-Blue: Digital Signatures Pattern (flow) - Node-RED.” Accessed: Sep. 22, 2023. [Online]. Available: https://flows.nodered.org/flow/d3df2f5b85e2a44837b7dfb99c30f84b
[56] “node-red-contrib-web3-blue.” Accessed: Sep. 22, 2023. [Online]. Available: http://flows.nodered.org/node/node-red-contrib-web3-blue
[57] “Web3-Blue: Smart Contracts Pattern (flow) - Node-RED.” Accessed: Sep. 22, 2023. [Online]. Available: https://flows.nodered.org/flow/0590dd348fcde72632333e1912aa0d2b#

Disclaimer

Copyright Message