PHYSICS

OPTIMIZED HYBRID SPACE-TIME SERVICE CONTINUUM IN FAAS

D4.2 – Cloud Platform Services for a Global Space-Time Continuum Interplay Scientific Report and Prototype Description V2

Lead Beneficiary	ATOS
Work Package Ref.	WP4 – Reference Work package Name
Task Ref.	T4.1 – Reasoning Framework for Semantic Matching and Runtime Adaptation T4.2 – Cloud Services and Edge Devices Performance Evaluation T4.3 – Global Continuum Patterns Placement T4.4 – Distributed In-memory State Services for Data Interplay T4.5 – Adaptive Platform Deployment, Operation & Orchestration
Deliverable Title	D4.2 – Cloud Platform Services for a Global Space-Time Continuum Interplay Scientific Report and Prototype Description V2
Due Date	2023-09-30
Delivered Date	2023-09-30
Revision Number	3.0
Dissemination Level	Public (PU)
Type	Report (R)
Document Status	Final
Review Status	Internally Reviewed and Quality Assurance Reviewed
Document Acceptance	WP Leader Accepted and Coordinator Accepted
EC Project Officer	Mr. Stefano Foglietta

H2020 ICT 40 2020 Research and Innovation Action

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

Contributing Partners

Partner Acronym	Role[1]	Name Surname[2]
UPM	Contributor	Marta Patiño Ainhoa Azqueta
HUA	Contributor	George Kousiouris T. Stamati V. Katevas C. Giannakos
RHT	Contributor	Luis Tomas
RYAX	Contributor	Yiannis Georgiou
INNOV	Contributor	George Fatouros
BYTE	Contributor	Yiannis Poulakis
ATOS	Lead Beneficiary	Carlos Sánchez Roi Sucasas
FUJITSU	Internal reviewer	Isabelle Ehresmann
REDHAT	Internal reviewer	Luis Tomas
DFKI	Quality assurance	Carsten Harms Maciej Kolek

Revision History

Version	Date	Partner(s)	Description
0.1	2023-06-30		ToC Version
0.2	2023-09-15		1st draft Version
1.0	2023-09-21		Version for Peer Reviews
2.0	2023-09-25		Version for Quality Assurance
3.0	2023-09-29		Version for Submission

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

[2] Can be left void

LIST OF ABBREVIATIONS

API	Application Programming Interface
CI/CD	Continuous Integration / Continuous Delivery
CNCF	Cloud Native Computing Foundation
CPU	Central Processing Unit
CRUD	Create, Read, Update and Delete basic operations of an entity
CSP	Cloud Service Provider
DevOps	Development Operations
DMS	Distributed Memory Service
DoA	Descriptions of Action
Dx.y	Deliverable number y corresponding to WP x
EC	European Commission
EKS	Elastic Kubernetes Service
ETL	Extract, Transform and Load
FaaS	Function as a Service
GCP	Global Continuum Placement
GPU	Graphics Processing Unit
HPC	High Performance Computing
IaaS	Infrastructure as a Service
IaC	Infrastructure as Code
JSON-LD	Javascript Object Notation – Linked Data
K8S	Kubernetes containers orchestrator software
KG	Knowledge Graph
MILNP	Mixed Integer Nonlinear Programming
ML	Machine Learning
Mx	Project Month x
OAS	Open API Specification
OCM	Open Cluster Management
OTA	Over-the-air distribution of software at the edge
OW	OpenWhisk FaaS platform
PaaS	Platform as a Service
PEF	Performance Evaluation Framework
PKI	Primary Key Indicator

QoE	Quality of Experience
QoS	Quality of Service
RA	Reference Architecture
RAM	Random Access Memory
RDF	Resource Description Framework
REST	Representational State Transfer
RPC	Remote Procedure Call
SaaS	Software as a Service
SFG	Serverless Function Generator
SLA	Service Level Agreement
Tx.y	Task number y belonging to WP x
UI	User Interface
UML	Universal Modelling Language
URI	Uniform Resource Identifier
WP	Work package

EXECUTIVE SUMMARY

This document (D4.2) has been built on top of the V1 version (D4.1) produced in M13. It includes the previous content, updated where necessary, so that there is one overall document that describes the entire WP4 outcomes at the end of the project. The Summary of Changes table details the sections that have been changed in relation to the V1 version and which ones are new or have been heavily updated since V1. This version presents all the work done in the different WP4 tasks. The main objective of this release is to describe the final software components from this technical WP aligned with the result of the other technical WPs of the PHYSICS project (WP3 and WP5).

The scope of this deliverable consists of a functional and technical description of the prototype of every component of the WP4 toolkits. This includes functionalities, components architecture details, overall data models, public interfaces and some demonstrations and/or experimentation outcomes.

Each component in this document has a dedicated chapter with a common structure to describe the work done. WP4 aims to bridge the gap between the WP3 development environment and the WP5 infrastructure. For this a Reasoning Framework collecting the information from the overall application and platform is described in chapter 2. The Performance Evaluation Framework (PEF in Chapter 3) helps in integrating performance analysis of functions and FaaS platforms in order to aid, among others, in function development, placement and operation. In Chapter 4, the Global Continuum Placement serves for optimal allocation of functions across the available clusters, while Chapter 5 provides state handling services for functions. All the previous components require a final step to attain application deployment in the infrastructure. This work is attained by the Orchestrator whose details are described in Chapter 6. New APIs have been defined in order to support the processes and interconnect platform components. Data on the detailed experimentation has been supplied when suited to defend specific choices and additions. Finally, the document ends with general conclusions for the work done in this WP.

Summary of Changes

CHAPTER	UPDATE	SECTION(s)
Chapter 1	Updated text to reflect the presentation of this second version of the WP4 deliverable	1.3
Chapter 1	Update Overview diagram	
Chapters 2-6	Removed (the info is planned to be included in D6.2)	Installation and usage Next steps (last deliverable)
Chapter 2	Additional features related to Runtime Adaptation; Updates (text and figures) on the Reasoning Framework based on the 2nd period developments	2.1.1, 2.1.2, 2.2
Chapter 2	New Experimental Results	2.3
Chapter 3	Minor or no updates (Sections can be skipped)	3.1.1, 3.1.2 ,3.1.3, 3.2.3b and 3.2.3c, 3.2.5, 3.3.1-3.3.5
Chapter 3	Major updates. Reengineering of PEF around a more FaaS based architecture and PEF APIs.	3.2.1, 3.2.2, 3.2.3a, 3.2.6
Chapter 3	New Sections	3.2.4, 3.3.6, 3.3.7
Chapter 4	Extended the technical description to describe the developed extensions and optimization algorithms and extended the demonstration and experimentation to capture the changes and the new evaluations	4.2 and 4.3
Chapter 5	Description of the new implementation of the component, the functions created to integrate with the new implementation and the new evaluation.	
Chapter 6	New features and overall architecture	
Chapter 7	New section that summarizes the installation and usage	Previously, each chapter had a section for this topic.

CONTENTS

EXECUTIVE SUMMARY 5
Summary of Changes 6
1. Introduction 14
1.1. Objectives of the Deliverable 16
1.2. Insights from other Tasks and Deliverables 16
1.3. Structure 16
2. Reasoning Framework for Semantic Matching and Runtime Adaptation 18
2.1. Functional description 18
2.1.1 Main objective and functionalities 18
2.1.2 Relationship with other components 19
2.1.3 Requirements matrix 20
2.2. Technical description 21
2.2.1 Baseline technologies and dependencies 21
2.2.2 Component internal architecture 21
2.2.3 Interfaces and Integrations 23
2.2.4 Data Model 24
2.3. Demonstration or Experimentation outcomes 25
2.3.1 Reasoning Framework Latency 25
2.3.2 Runtime Adaptation 28
3. Cloud Services and Edge Devices Performance Evaluation 34
3.1. Functional description 34
3.1.1 Main objective and functionalities 34
3.1.2 Relationship with other components 36
3.1.3 Requirements matrix 36
3.2. Technical description 37
3.2.1 Baseline technologies and dependencies 37
3.2.2 Component internal architecture 38
3.2.3 Interfaces, adaptations and integrations 39
3.2.4 Function Design Approach Details 45
3.2.5 Specific workload parameters and experiments consideration 54
3.2.6 Data Model 60
3.3. Demonstration or Experimentation outcomes 61
3.3.1 Batch Request Aggregator Performance Model 61
3.3.2 Cold vs Warm vs Hot Executions 67
3.3.3 Node-RED versus Node.js runtimes for Hello world single function application 68
3.3.4 Node-RED flow orchestration to OW function orchestration delays comparison 70
3.3.5 Execution Model transformation for model querying results 76
3.3.6. Benchmarking of ehealth across different clusters 77

3.3.7. Split Join Pattern Performance Experimentation 93
4. Global Continuum Patterns Placement 99
4.1. Functional description 99
4.2. Technical description 101
4.2.1. Generic component and first best fit algorithm with multi-objective capabilities 101
4.2.2. Optimization algorithms 103
4.3. Demonstration and experimentation 105
4.3.1 Demonstration of the implemented algorithm 105
4.3.2. Simulation-based experimentation and evaluation of the optimized multi-objective algorithms 108
4.4. Discussion 110
5. Distributed In-memory State Services for Data Interplay 112
5.1. Functional Description 112
5.1.1 Main Objective and Functionalities 112
5.1.2 Relationship with Other Components 112
5.1.3 Requirements Matrix 112
5.2. Technical Description 113
5.2.1 Interfaces and Integration 113
5.2.2 Usage 115
5.3. Experimentation Outcomes 116
5.3.1 KeyDB as a Service 116
5.3.2 DMS deployed in a Kubernetes cluster 119
6. Adaptive Platform Deployment, Operation & Orchestration 124
6.1. Functional description 124
6.1.1. Background 124
6.1.2. Multicluster platform 125
6.1.3. Connecting applications across clusters 125
6.1.4. Alarm component when cluster shows limitations 125
6.2. Technical description 126
6.2.1 Baseline technologies and dependencies 126
6.2.2 Component internal architecture 127
6.2.3 Interfaces and integrations 134
6.3. Demonstration and experimentation 136
6.3.1. Scenario or experimentation description 136
7. Integration with Physics platform 140
8. Conclusions 141
9. References 142

LIST OF TABLES

- Table 1: Structure of the document
- Table 2: Reasoning Framework API Description
- Table 3: Reasoning Framework's Response Latency
- Table 4: Cost per Experiment
- Table 5: API operations for PEF process execution
- Table 6: PEF DB API operations for test results, storage acquisition and filtering
- Table 7: PEF component main entities data model
- Table 8: Cost Estimation of GB-second based on AWS Lambda pricing model if multiple rows are aggregated following the Request Aggregator pattern
- Table 9: Function sequence hot/cold execution times with the different runtime modes
- Table 10: Testbed Data for Various Test Configurations
- Table 11: Effective Rate and Request Acceptance Probability for test name = ehealthHUA
- Table 12: Effective Rate and Request Acceptance Probability for test name = ehealthAWS
- Table 13: Effective Rate and Request Acceptance Probability for test name =ehealthAZURE
- Table 14: Comparison of Testbed Data for Various Test Configurations with baseline HUA32, AWS16, AZURE4
- Table 15: Input Ranges for the Split-Join Experiment
- Table 16: DMS interface
- Table 17: Orchestrator component dependencies
- Table 18: Components required for the Orchestrator once the Deployment graph has been received.
- Table 19: Update of fields for default parameters in the Manifestwork due to the reallocation of interconnected functions.
- Table 20: Translator unique POST endpoint that received the JSON deployment graph.
- Table 21: Endpoints for the Alarm microservice API.
- Table 22: Endpoints of the Openwhisk proxy API.

LIST OF FIGURES

- Figure 1: WP4 tasks logical components relationship extracted from D2.5 Physics architecture.
- Figure 2: Overview of Graphs used in PHYSICS platform
- Figure 3: Relationship with other components
- Figure 4: Reasoning Framework architecture
- Figure 5: Forecaster's internal Architecture and Components
- Figure 6: Resource Graph without INSERT queries
- Figure 7: Resource Graph after updating it with INSERT queries
- Figure 8: Deployment Graph: The edges "Allocatable" indicate the graph nodes of type "Flow" that could be hosted at the cluster with id "5653...72d".
- Figure 9: Architecture of RuntimeAdaption Evaluation setup
- Figure 10: Mean Response and Execution Latency between different experiments
- Figure 11: Wait and response latency with Fibonacci function and performance adaptation
- Figure 12: Wait and response latency with Fibonacci function and 50-50 routing policy
- Figure 13: Standard Deviation of Response Latency between different experiments
- Figure 14: PEF Run Test Use Case diagram
- Figure 15: PEF Evaluate Performance Use Case diagram
- Figure 16: Pattern Modelling using PEF Use Case
- Figure 17: Performance Evaluation Framework component internal architecture
- Figure 18: Stack layers needed for the adaptation of the modelling image for function execution
- Figure 19: Node-RED wrapper flow for adapting the model inference process to a serverless execution
- Figure 20: Dockerfile needed for the creation of the baseline function image
- Figure 21: Model creation process of the PEF
- Figure 22: PEF overall process
- Figure 23: PHYSICS Load Generator Subflow Implementation
- Figure 24: PHYSICS Load Generator Subflow wrapped around an Openwhisk Interface
- Figure 25: Input Specification for the Load Generator
- Figure 26: Set rate system measurements point analysis
- Figure 27: Load generator Output Result of a Benchmark Run
- Figure 28: Function Chain Pattern Logic
- Figure 29: Load Generator Function Chain Inner Structure
- Figure 30: PEF FaaS Load Generator Adapter Flow
- Figure 31: Node-RED flow
- Figure 32: Input (Left) and Output (Right) JSON specification for the Clustering Function
- Figure 33: Profiling Clusters Creation Cron Job

- Figure 34: Classification Node-RED flow
- Figure 35: Example inputs to the Classification function a) profile trace for given function b) current clustering centroids per resource
- Figure 36: Example output of the Classifier Function
- Figure 37: Cold/Warm/Hot containers variations of executions
- Figure 38: Jmeter user interface
- Figure 39: Mixed executions that can defect the measurement process and mitigation without the usage of a synchronizing barrier at the end of each loop
- Figure 40: Potential mixed executions that can defect the measurement process and mitigation through the usage of a synchronizing barrier at the end of each loop
- Figure 41: Example of synchronisation solving, and problem inserted by maximum allowed containers
- Figure 42: Model Error in final validation cases
- Figure 43: Surface Plot of Model Predictor in the normalized range (Input1: Frequency, Input2: Batch Size, Output: Response Time)
- Figure 44: Client side request generation frequency measured per sample
- Figure 45: Client side Response Times per sample
- Figure 46: System side measured frequency per 1 second interval
- Figure 47: Active (running) containers in the system
- Figure 48: Batch Size Setting by ANN model and according system side application of the setting
- Figure 49: Cold/Warm/Hot Jmeter workloads response time of "Hello World" node.js function execution time
- Figure 50: Cold/Warm/Hot Jmeter workloads standard deviation of "Hello World" node.js function execution time
- Figure 51: Node-RED workflow of "Hello World" Node.js function (for Node-RED runtime)
- Figure 52: Node-RED execution times of "Hello World" Node.js function (with Node-RED runtime)
- Figure 53: Node.js OpenWhisk function flow execution (many containers) vs. Node-RED (same container)
- Figure 54: Artificial Delay flow to mimic dynamic sequence numbers in Node-RED
- Figure 55: Difference in Orchestration Times between OW and Node-RED compared to the function numbers
- Figure 56: Interfunction Delay Measurement
- Figure 57: IFC mode comparison
- Figure 58: Hot Function sequence execution
- Figure 59: Function estimated total execution for different function numbers and delays
- Figure 60: Orchestration Overheads
- Figure 61: Proposal ANN model in a similar fashion as the RA model
- Figure 62: PEF transformation to serverless model measurement impact

- Figure 63: Overview of a FaaS System Architecture
- Figure 64: Performance plot of ehealthHUA
- Figure 65: Performance plots of ehealthAWS
- Figure 66: Behavior of the ehealthAWS with setRate=60 and memory=512
- Figure 67: Performance plots of ehealthAZURE
- Figure 68: Inter-arrival time (seconds) between processed request samples in ehealthHUA
- Figure 69: Inter-arrival time (seconds) between processed request samples in ehealthAWS
- Figure 70: Inter-arrival time (seconds) between processed request samples in ehealthAzure
- Figure 71: Distribution of Execution Time for Memory 256
- Figure 72: Distribution of Execution for Memory 512 across Cluster
- Figure 73: Distribution of Execution for Memory 1024 across Clusters
- Figure 74: Distribution of Waiting Time for Memory 256
- Figure 75: Distribution of Waiting Time for Memory 512
- Figure 76: Distribution of Waiting Time for Memory 1024
- Figure 77: Average Total User Side Response Time
- Figure 78: Average Waiting Time for Function Invocation
- Figure 79: Split Join Experiment Automation Flow
- Figure 80: Average Response Time and Cost compared to Input Size for 2 Max Containers
- Figure 81: Average Response Time and Cost compared to Input Size for 8 Max Containers
- Figure 82: Average Response Time and Cost compared to Input Size for 32 Max Containers
- Figure 83: Global Continuum Placement high-level view
- Figure 84: Global Continuum Placement component internal architecture
- Figure 85: FOA's algorithm step by step
- Figure 86: FOA's linear program trade-off between makespan (x-axis) and amount of downloaded data (y-axis). The color of the points represents the iteration-id of the linear program binary search. Each one is one solution of the linear program.
- Figure 87: KeyDB vs Redis Ops/Sec performance
- Figure 88: KeyDB vs. Redis performance
- Figure 89: KeyDB and Redis performance evaluation (Ops/sec)
- Figure 90: KeyDB and Redis performance evaluation (Latency)
- Figure 91: KeyDB evaluation with the client and the database in different nodes
- Figure 92: Get latency with and without persistence in 1 node
- Figure 93: Set latency with and without persistence in 1 node
- Figure 94: Get latency without persistence, without replication, master-replica and active replication in 3 nodes
- Figure 95: Set latency no persistence, no replication, master-replica and active

replication in 3 nodes
- Figure 96: Diagram of WP4 components and Orchestration pipeline with runtime adaptation based on OpenWhisk metrics collected in Prometheus.
- Figure 97: Plan, Do, Check, Act cycle in the Orchestrator
- Figure 98: Deployment flow

LIST OF SOURCE CODE EXAMPLES

- Code 1: Reasoning Framework API specifications
- Code 2: Cluster data send to the Placement Optimizer
- Code 3: Registering the image with OW using the create/update command
- Code 4: Configuration file
- Code 5: Available memory
- Code 6: Configuration
- Code 7: Dynamic configuration
- Code 8: Code snippet showing platform information.
- Code 9: Code snippet to register the platform information using an API call
- Code 10: Code snippet showing demands and constraints per task
- Code 11: Code snippet to realize the placement by global continuum engine.
- Code 12: Code snippet with output of the result from the locations
- Code 13: functionWrite.py
- Code 14: functionRead.py
- Code 15: Functions and Sequence creation
- Code 16: Sequence invocation results
- Code 17: Python function
- Code 18: Code snippet with JSON Deployment Graph describing the application.
- Code 19: Code Snippet with YAML Manifestwork that incorporates the Workflow CRD from the previous Deployment
- Code 20: Code snippet showing QoS definition example
- Code 21: Code Snippet for Prometheus metric processing
- Code 22: Code snippet of the Alarm notification example
- Code 23: Code snippet showing an example of a Deployment Graph
- Code 24: Code snippet of the output of the manifestworks in the cluster
- Code 25: Code snippet of the QoS object/definition
- Code 26: Code snippet of the QoS violation notification

1. Introduction

This second version of WP4 deliverable describes the main components of the WP, the success concerning the main objectives and functionalities proposed in the first version of the deliverable, and the relation with other components in the same WP and other WPs. Moreover, the component architecture is explained in detail together with the overall data model and state of the development for the release of the second prototype of the WP4 toolkit before the final integration.

The main objective of WP4 is to implement an abstraction layer to allow management and optimization of applications consisting of flows or functions that are obtained using WP3 infrastructure and allow these functions connected to whole applications to be distributed in multiple clusters. These clusters could be from different CSPs (Cloud Service Providers) or located in different sites of the same CSP. This does also include possible hybrid edge/cloud scenarios with different possible distributions of Kubernetes.

Additionally, we provided the capability to run the function in multiple FaaS (Functions as a Service) engines or platforms by using workflow abstractions from WP5 which facilitates the adaptation and future proof of the platform to possible upcoming FaaS alternatives. Figure 1 shows the WP4 logical components (T4.1, T4.2, T4.3, T4.5) and their relationship with the rest of the PHYSICS system. These components are connected in a pipeline that transforms, enriches, and forwards information about the application and its constituent functions to obtain an optimized deployment of the overall application considering all the nuances of both the multicluster platform as well as the functions and their specific constraints.

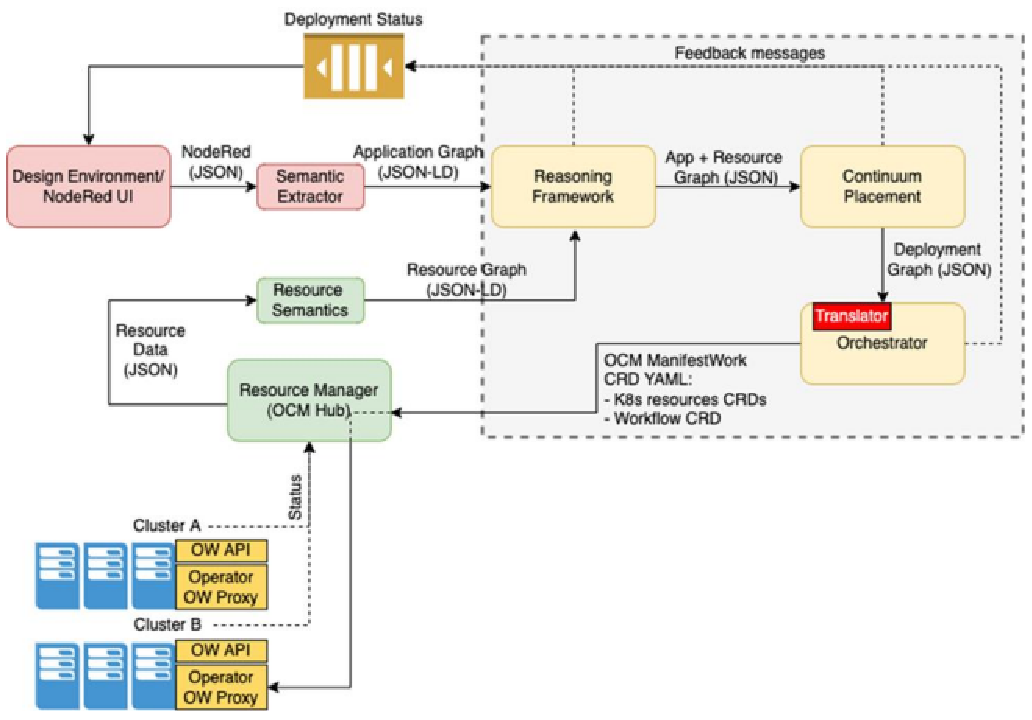


Figure 1: WP4 tasks logical components relationship extracted from D2.5 Physics architecture.

The main components of this WP4 are the following:

- **Inference Engine / Reasoning Framework:** this framework is responsible for obtaining candidate services to be used for the deployment, recommending available design patterns and retrieving the necessary options for the deployment configuration.
- **Performance Evaluation:** this component test tool collects performance evaluation data from the execution of designated workloads on top of the available services and infrastructures.
- **Global Continuum Placement:** decision making component that makes decisions based on various aspects of the platform and application execution to select the right compute resources for the placement of the different tasks of the application.
- **Distributed Memory Service:** backend service to allow sharing of data between functions invocations.
- **Adaptive Platform Deployment, Operation and Orchestration (Orchestrator):** the component that allows the dynamic deployment, reconfiguration and adaptation of the application in the global continuum edge-cloud.

We will finish this document with achievements and identified challenges for the final integration of the project.

1.1. Objectives of the Deliverable

The main objective of this deliverable is to describe the work done until M30, which includes the reasoning behind the design and development of the final prototype of the WP4 toolkit. This document describes WP4 components, covering both functional and design aspects, including technical and implementation decisions. This includes the corresponding information and control flows between components of this WP4, and the boundaries with WP3 and WP5 components.

This deliverable is the current understanding and development of the components of WP4. While the communication between components is custom for this project, the use of open standards such as JSON would allow to leverage each of the component functionalities in a different scenario and the potential for independently developed products is not to be disregarded.

1.2. Insights from other Tasks and Deliverables

This deliverable follows the global architecture defined in **D2.5.** "Physics Reference Architecture Specification V2" as a guide to introduce the functional and technical description of the components belonging to the WP4 toolkit.

We also use the requirements defined in **D2.3.** "State Of The Art Analysis And Requirements V2" to define and prioritize the prototype of the WP4 toolkit.

At the same time this deliverable was written, other two interrelated technical deliverables were working in progress (**D3.2**. Functional And Semantic Continuum Services Design Framework Sci. Report And Prototype Description V2 and **D5.2.** Extended Infrastructure Services With

Adaptable Algorithms Scientific Report And Prototype Description V2) that were synchronized with this deliverable to make PHYSICS an integrated system

1.3. Structure

We have divided the deliverable into one chapter per component of this WP4 (Table 1). Every chapter/component will have a functional and a technical section to provide a complete overview of the functionalities of the components, and it will cover the design and implementation details.

Table 1: Structure of the document

Chapter	Description
Chapter 1. Introduction	Short introduction to this deliverable, objectives and scope.
Chapter 2. Reasoning Framework for Semantic Matching and Runtime Adaptation	Functional and technical description of the Inference Engine component.
Chapter 3. Cloud Services and Edge Devices Performance Evaluation	Functional and technical description of the Benchmarking component.
Chapter 4. Global Continuum PatternsPlacement	Functional and technical description of the ContinuumPlacement component.
Chapter 5. Distributed In-memory State Services for Data Interplay	Functional and technical description of the Distributed In memory State (DMS) component.
Chapter 6. Adaptive Platform Deployment, Operation & Orchestration	Functional and technical description of the deployment Orchestrator component.
Chapter 7. Integration with Physics platform	Installation and usage of the components within the Physics infrastructure.
Chapter 8. Conclusion	The results of the work described in the deliverable and the next steps and challenges of these components.

2. Reasoning Framework for Semantic Matching and Runtime Adaptation

2.1. Functional description

2.1.1 Main objective and functionalities

The **Reasoning Framework for Semantic Matching and Runtime Adaptation** of T4.1, abbreviated Reasoning Framework, provides a central registry/datastore with the appropriate APIs for storing and retrieving applications and resources metadata within the concept of graphs [1]. In the context of PHYSICS platform graphs are divided into the following four types illustrated in Figure 2:

1. Application graph (created by the Application Semantics component, see D3.1): Describes a given application that can consist of functions, flows, sub-flows, patterns and services.
2. Resource graph (created by the Resource Semantics component, see D5.1): Describes given computational resources such as public clouds and clusters that can consist of compute nodes, storage services, edge devices, etc.
3. Global graph (created by the Reasoning Framework): Includes a given application graph with its nodes connected with the nodes of the **available** resource graphs that **could** deploy each node of the application graph.
4. Deployment graph (created by the Global Continuum Patterns Placement, see section 4): Includes a given application graph with its nodes connected with the **optimal** nodes from the available resource graphs that **will be used** to deploy each node of the application graph.

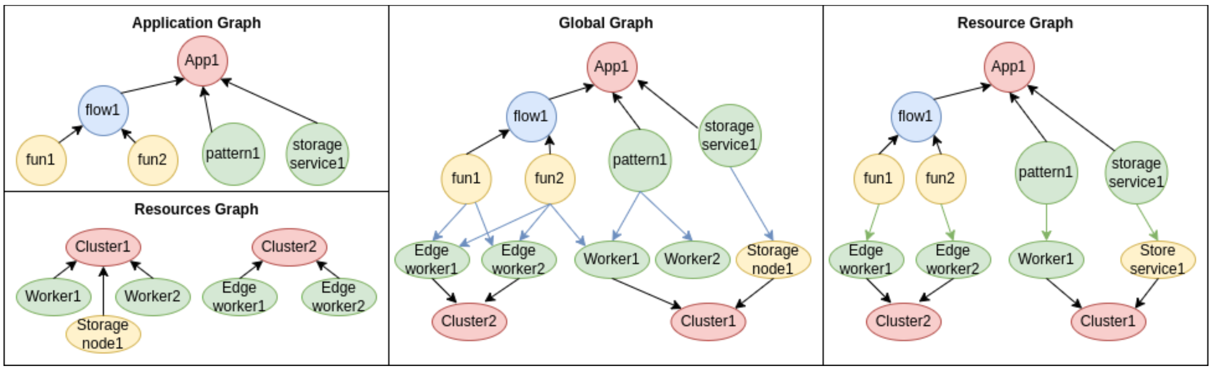


Figure 2: Overview of Graphs used in PHYSICS platform

In this way, this component also performs semantic reasoning on the input metadata to infer relationships between application and resource graphs. As a result, the Reasoning Framework creates a global graph by filtering out the available compute resources that can serve the input application. It is noted that each component of an application includes functions, flows and services that could be deployed on a different computing node according to its requirements provided by WP3 as annotations such as CPU, RAM, GPU, location, and latency.

Furthermore, to enhance the placement of functions, pertinent performance data is integrated into the Reasoning Framework. This data falls into three categories:

1. Data derived from benchmarks executed during cluster onboarding time, as outlined in D5.2.

2. Function-specific benchmarks provided by T4.2 and WP3 performance pipelines.

3. Forecasted runtime metrics, which are estimated during the application's runtime, especially when there is a deterioration in the deployed function's performance, facilitating on-the-fly updates to the global graph.

Thus, the main functionalities of this component are the following:

- Repository for application and computational resources metadata.
- API for registering, querying and updating resource and application graphs.
- Semantic matching between application requirements and available resources (e.g., K8s clusters) to deploy the given application.
- Intermediate component between WP3, WP4 and WP5.
- Facilitates the runtime adaptation by updating the performance metrics.

2.1.2 Relationship with other components

As mentioned above, the **Reasoning Framework** could be perceived as an interface between the different layers of the PHYSICS platform serving also as a central repository for application and resource metadata. Hence, it interacts with various components that either provide or request data (see Figure 3).

As far as the input data is concerned, it should be structured into data models according to the Linked Data principles [2] before entering the Reasoning Framework. This will allow interoperability between the different components of the platform, seamless data sharing and reasoning/inference on the data.

To that end, the PHYSICS ontology is being developed to describe the different types of utilized data and their relationships. This data could be divided into application and resource metadata. The former generated in WP3 is processed by the **Application Semantics** component to be transformed according to the ontology descriptions and annotations. Accordingly, resource metadata available from WP4 and WP5 is injected into the **Resource Semantics** component, which is responsible for "translating" the different resource data to the common language defined by the ontology. Thus, the **Application** (Resource) Semantics component creates an individual application (Resource) graph for each new application (Resource) registered on the platform that can be imported into the Reasoning Framework and stored in its quad-store.

The Reasoning Framework, on the other hand, outputs data to other components based on their requirements. Specifically, when an application needs to be deployed, its global graph (i.e., the application graph connected with the relevant resource nodes) created by the reasoner of T4.1 is passed to **Global Continuum Patterns Placement** (T4.3) for further optimization. In addition, WP3 may retrieve and/or update an application graph from the Reasoning Framework during the application design leveraging the provided API. Furthermore, the Design environment (T3.1) queries the Reasoning Framework's graph datastore to retrieve stored reusable patterns and flows to be recommended to the developer.

Additionally, this component is integrated with the Runtime Adaptation **Forecaster** and the **Cluster Availability Monitor**, both developed in the context of T4.1 during the 2nd period of the project. The former, triggered by an alarm service of T4.5 (section 6.1.4), predicts the clusters' performance based on recent OpenWhisk metrics and dictates the Reasoning Framework to redeploy the WP4 pipeline using the forecasted data. The Cluster Availability Monitor continuously checks the health of OpenWhisk in each registered cluster and updates accordingly the availability scores of each cluster that are stored in the reasoning framework (see also D6.2).

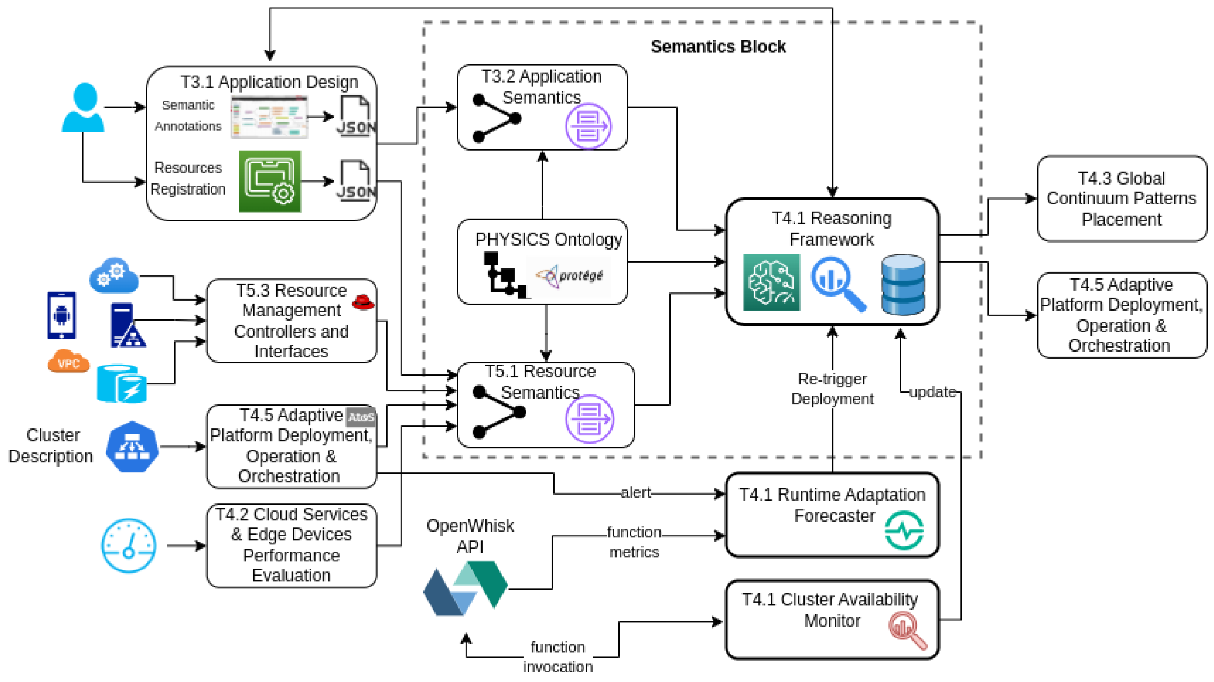

Semantics Block
T3.1 Application Design
Semantic Annotations
Resources Registration
JSON
JSON
T3.2 Application Semantics
PHYSICS Ontology
protégé
T5.1 Resource Semantics
T4.1 Reasoning Framework
T4.3 Global Continuum Patterns Placement
T4.5 Adaptive Platform Deployment, Operation & Orchestration
T5.3 Resource Management Controllers and Interfaces
VPC
Cluster Description
T4.5 Adaptive Platform Deployment, Operation & Orchestration
Atos
T4.2 Cloud Services & Edge Devices Performance Evaluation
Re-trigger Deployment
update
alert
T4.1 Runtime Adaptation Forecaster
OpenWhisk API
function metrics
T4.1 Cluster Availability Monitor
function invocation

Figure 3: Relationship with other components

Finally, several other query endpoints available through this Framework facilitate PHYSICS components' needs. For instance, it allows updating specific resource data such as cluster scoring and sizing information, and retrieving cluster node data or application annotations and characteristics.

2.1.3 Requirements matrix

With relation to the expressed requirements in D2.2 and the functionalities described in the introduction of this chapter; this component is involved in the following:

- Req-4.1-Inputs
  - Domain coverage by ontology. The ontology should be able to describe the characteristics, properties and annotations of the input application and resource data.
- Req-4.1-Reasoning
  - Required time for reasoning (Number of SPARQL queries, query mean response time)
- Req-4.1-Adaptation

- Given the performance evaluation of the deployed applications, the Reasoning Framework should update accordingly during runtime the global graph.

2.2. Technical description

2.2.1 Baseline technologies and dependencies

The Reasoning Framework component is a microservice incorporating the functionalities described in subsection 2.1.1 based on the Flask Web framework [3], which features a lightweight and modular design enabling fast development without compromising performance [4]. This service relies on AllegroGraph[3], a horizontally Distributed, multi-model (document and graph), entity-event knowledge graph (KG) technology that enables the extraction of sophisticated decision insights and predictive analytics from highly complex, distributed data that cannot be answered with conventional databases. AllegroGraph provides an architecture through the REST protocol, while there are APIs for various programming languages including Python [5]. This facilitates the enhancement of analytical models, which are typically served by Python-based applications, with features retrieved from the KB. Furthermore, a crucial dependency of the Reasoning Framework is the PHYSICS ontology based on its classes, attributes, annotations, and properties, the Reasoning Framework becomes functional.

2.2.2 Component internal architecture

Reasoning Framework's internal architecture along with the technologies mentioned above are illustrated in Figure 4. The component is responsible for exposing specific REST endpoints so that other platform components (i) ingest application and resource data, (ii) retrieve required information and (iii) infer insights from the AllegroGraph for optimizing application design and deployment in terms of cost, latency, performance and more. Specifically, the design environment posts the application graph to the Flask service that forwards it to the KB. In a similar way, utilizing the resource semantics component presented in [6], each cluster registered in the platform sends its description in the Reasoning Framework. Depending on the type of the input data, Flask guides AllegroGraph to create further relationships at both individual and default graph level. As further analyzed in the next subsection, this facilitates the timely retrieval of specific data needed by the platform as well as provides inference on the possible function allocations.

The overall architecture follows the microservices approach as both components (i.e., KG and Reasoning Framework) are containerized (using Docker) and integrated as a single service allowing additional services to be added without affecting the existing component.

[3] https://allegrograph.com/products/allegrograph/

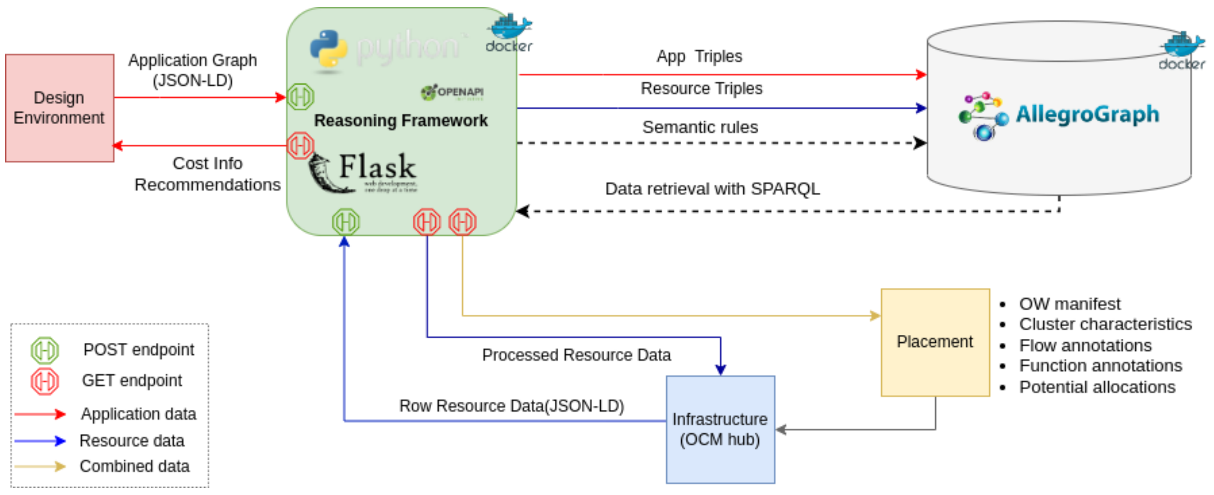
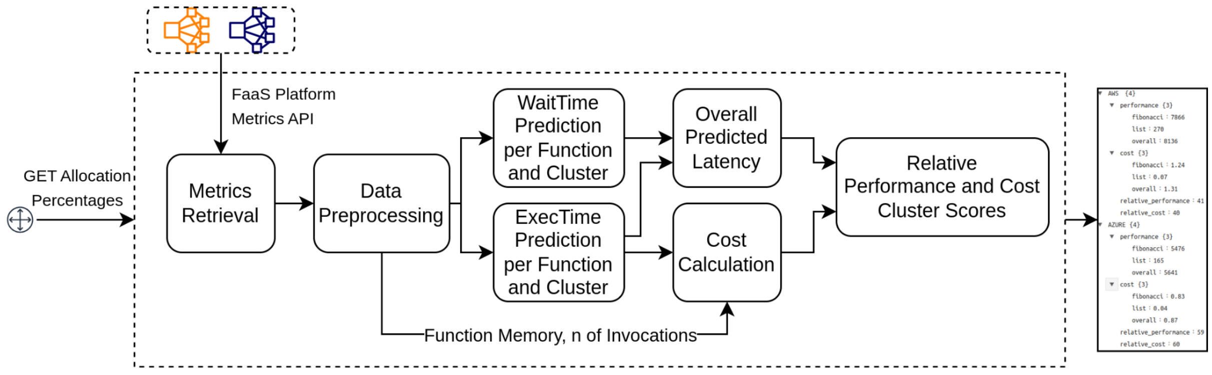


Figure 4: Reasoning Framework architecture

In the scope of Runtime Adaptation, an additional Flask Service, named **Forecaster** was developed and integrated that uses recent metrics to predict the expected function latency across each cluster. It computes the percentage of requests to be routed to each cluster, optimizing for either latency (i.e., sum of wait and execution time) or cost (i.e., execution time as the main billing factor of the FaaS model [7]).

The Forecaster is structured as a REST API developed using Flask with Gunicorn serving as the WSGI server [8]. Additionally, it features integration with a message broker, such as RabbitMQ, which enables event-driven triggering. This API consists of a single endpoint, which, when triggered asynchronous or by messages from the message broker, initiates the process of data retrieval from the OpenWhisk metrics API[4] of each registered cluster. Data retrieval is performed in parallel through multi-threading, covering the recent 15-minute period, considering that FaaS platforms typically retain containers alive for 10 to 15 minutes. Forecaster's internal architecture and components are illustrated in Figure 5.


FaaS Platform Metrics API
GET Allocation Percentages
Metrics Retrieval
Data Preprocessing
WaitTime Prediction per Function and Cluster
ExecTime Prediction per Function and Cluster
Overall Predicted Latency
Cost Calculation
Relative Performance and Cost Cluster Scores
Function Memory, n of Invocations

```
▼ AWS {4}
  ▼ performance {3}
      fibonacci : 7866
      list : 270
      overall : 8136
  ▼ cost {3}
      fibonacci : 1.24
      list : 0.07
      overall : 1.31
    relative_performance : 41
    relative_cost : 40
▼ AZURE {4}
  ▼ performance {3}
      fibonacci : 5476
      list : 165
      overall : 5641
  ▼ cost {3}
      fibonacci : 0.83
      list : 0.04
      overall : 0.87
    relative_performance : 59
    relative_cost : 60
```

Figure 5: Forecaster's internal Architecture and Components

The initial step in the processing phase is data preprocessing, where the data are streamlined to a usable format. This involves retaining only relevant fields such as activationId, name, start,

[4] https://github.com/apache/openwhisk/blob/master/docs/metrics.md

end, waitTime, duration, and memory. Subsequently, the data are aggregated with a frequency of one minute, followed by inferring the required details, including the number of activations per minute.

After the preprocessing, each function's data are fed into an exponential smoothing model [9], employing separate model instances for duration and waitTime metrics, with a smoothing level set to 0.5. This model predicts the execution and wait time for each function executed in every cluster within the last 15 minutes.

To estimate performance, the predicted duration and wait time values are summed, resulting in a performance score (predicted total latency) measured in milliseconds for each cluster. When estimating cost, the prediction involves considering the function execution time, mean number of invocations and the memory used by each function, with the score being expressed in dollars.

Following this, the absolute scores are converted to relative scores. This is achieved by dividing the inverted metric of the current cluster by the sum of the inverted metrics across all clusters as shown in the below equation. The result is then multiplied by 100 to obtain a percentage value, facilitating a direct comparison of relative scores between different clusters.

$$\text{relative\_scores}[c][m] = \left(\frac{\text{inverted\_metric}[c][m]}{\sum_{c' \in C} \text{inverted\_metric}[c'][m]} \right)$$

In the equation above, relative_scores[c][m] calculates the relative score for a given cluster c and metric m. This is achieved by dividing the inverted metric of the current cluster by the sum of the inverted metrics across all clusters, denoted as c' in the set of all clusters C. The result is then multiplied by 100 to obtain a percentage value, facilitating a direct comparison of relative scores between different clusters.

2.2.3 Interfaces and Integrations

As mentioned in the previous subsections, the Reasoning Framework's API provides various REST endpoints to interact with other platform components. To that end, the component will be compatible with the OpenAPI Specification (OAS) standard [10] ensuring seamless integration and ease of use. The specifications used in a sample of the developed REST API are given in the Code 1:

Code 1: Reasoning Framework API specifications

```
{info:
  version: "1.0.0"
  title: "physics_semantics_block"
basePath: "/api/v1"
schemes:
  - "http"
consumes:
  - "application/json"
produces:
  - "application/json"
paths:
  /add_resource/cluster:
    post:
      summary: "Save new resource of type cluster in the quadstore"
      operationId: "parse_resource_input"
      responses:
        200:
          description: "$cluster_instance.get_name() saved"
  /available_clusters:
    get:
      summary: "Returns information about the available resources of type cluster"
      operationId: "get_available_clusters"
      responses:
        200:
          description: "Clusters successfully retrieved"
}
```

In a similar manner, upon an alarm from the Adaptive Platform component (see subsection 6.1.4), the Forecaster predicts the performance score per cluster of a given application/function; updates the cluster's performance score in the Reasoning Framework; and re-triggers the deployment pipeline of WP4 via proper requests to the Reasoning Framework.

2.2.4 Data Model

Given that the Reasoning Framework deals with linked-data and semantic queries, internally, it utilizes the RDF graph model [11]. An RDF graph model is composed of nodes and arcs. An RDF graph notation or a statement is represented by a set of triples, each containing a node for the subject, a node for the object, and an arc for the predicate. A node may be left blank, a literal and/or be identified by a Uniform Resource Identifier (URI). An arc may also be identified by a URI. A literal for a node may be of two types: plain (untyped) and typed. A plain literal has a lexical form and optionally a language tag. A typed literal is made up of a string with a URI that identifies a particular data type [12].

Furthermore, since Reasoning Framework interacts with several platform services, data is exchanged leveraging JSON schema. For instance, Code 2 illustrates part of the data sent to the Optimizer (T4.3) when an application is deployed.

Code 2: Cluster data sent to the Placement Optimizer

```
{ ⊟
   "platform" : { ⊟
      "awshub" : { ⊟
         "id" : "ee228ef8-457d-11ee-a300-0a580a830017",
         "type" : "Cloud",
         "resources" : { ⊟
            "nb_cpu" : 32,
            "memory_in_MB" : 121915
         },
         "architecture" : "x86_64",
         "objective_scores" : { ⊟
            "Energy" : 1,
            "Availability" : 100,
            "Performance" : 66
         }
      },
      "azure4" : { ⊟
         "id" : "ee228ef8-457d-11ee-a300-0a580a830017",
         "type" : "Cloud",
         "resources" : { ⊞ },
         "architecture" : "x86_64",
         "objective_scores" : { ⊞ }
      }
   }
}
```

2.3. Demonstration or Experimentation outcomes

2.3.1 Reasoning Framework Latency

After ingesting data into the Reasoning Framework, these can be processed as graphs. With SPARQL queries to the KG, the required information, such for example the workflows included in an application or the sizing of each cluster, are made available as graph patterns [11]. However, some of these data require multiple queries to be retrieved and this results in increased processing time for the Reasoning Framework to respond to requests. For this purpose, specific relationships between the nodes of each input graph are implemented in the form of INSERT queries. Hence, all required information per the provided endpoint can be instantly retrieved with a SELECT query.

As an example, Figure 6 illustrates part of the available information of a target EKS cluster hosted in AWS before the INSERT query, where needed information (e.g., ram, cpu, architecture) for functions' allocations are part of each cluster node description. However, Reasoning Framework needs to aggregate such data to filter the clusters that do not meet application requirements. Such aggregations require multiple and complex queries as well as transformations performed at the Fask service. On the other hand, after the INSERT operation (see Figure 7), performed during cluster registration in the platform, all the required data (e.g., cpu cores and ram) are available at cluster level and can be retrieved with a SELECT query.

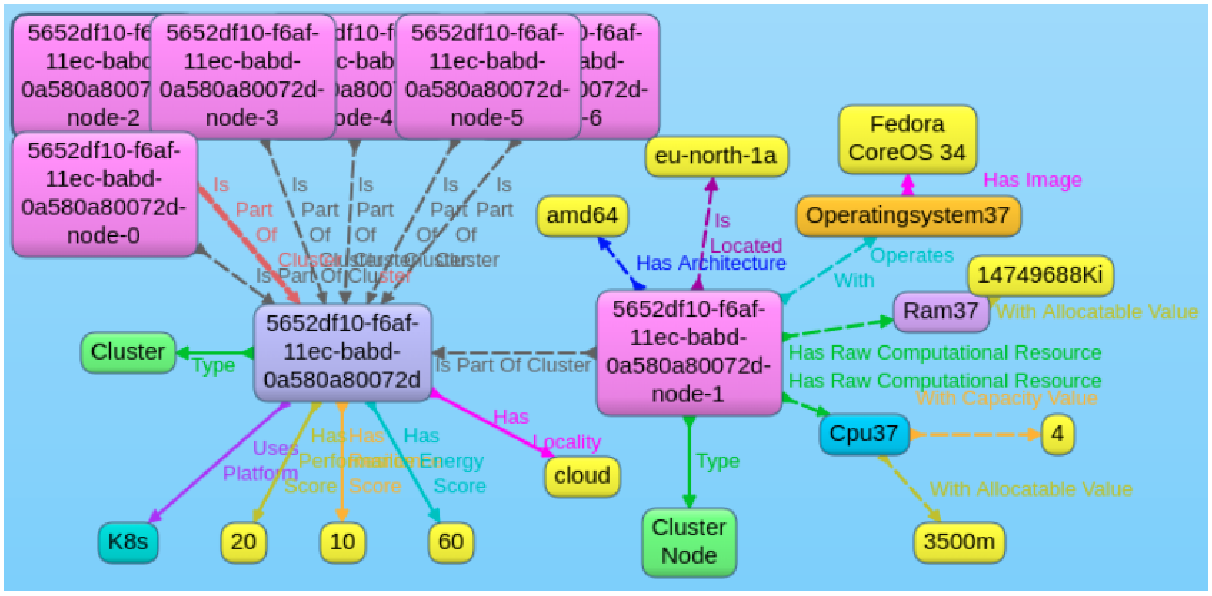

5652df10-f6af-11ec-babd-0a580a80072d-node-2
5652df10-f6af-11ec-babd-0a580a80072d-node-3
5652df10-f6af-11ec-babd-0a580a80072d-node-4
5652df10-f6af-11ec-babd-0a580a80072d-node-5
5652df10-f6af-11ec-babd-0a580a80072d-node-6
5652df10-f6af-11ec-babd-0a580a80072d-node-0
5652df10-f6af-11ec-babd-0a580a80072d
5652df10-f6af-11ec-babd-0a580a80072d-node-1
Is Part Of Cluster
Cluster
Type
K8s
Uses Platform
20
10
60
Has Performance Score
Has Resilience Score
Has Energy Score
Has Locality
cloud
amd64
eu-north-1a
Is Located
Has Architecture
Fedora CoreOS 34
Has Image
Operatingsystem37
Operates With
14749688Ki
Ram37
With Allocatable Value
Has Raw Computational Resource
Has Raw Computational Resource With Capacity Value
Cpu37
4
With Allocatable Value
3500m
Type
Cluster Node

Figure 6: Resource Graph without INSERT queries

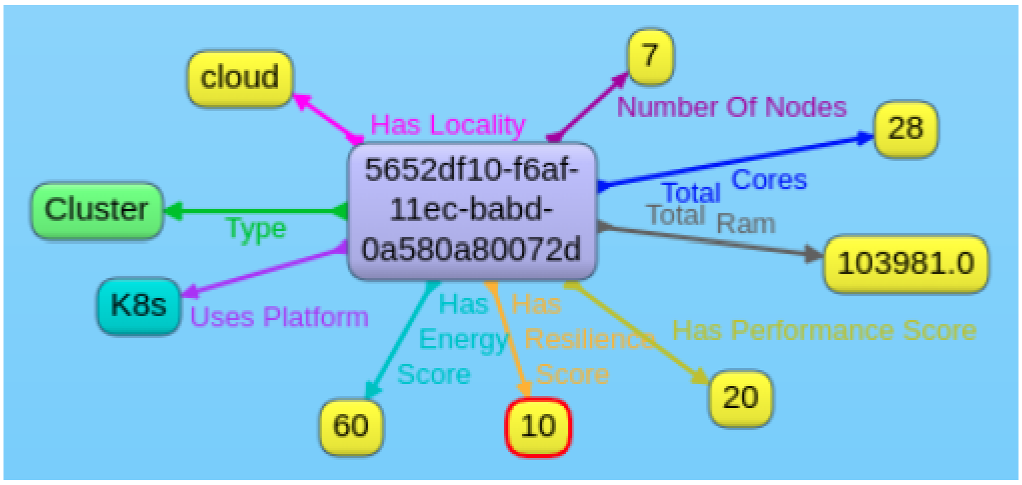


Figure 7: Resource Graph after updating it with INSERT queries

Furthermore, although AllegroGraph's built-in reasoner automatically infers some types of relationships between the nodes of the ingested graphs, they do not suffice to assign connections between application and resource graphs. To this end, several rules have been implemented to automate this process and create connections in the form of edges that link each flow with candidate target clusters. These rules enable the Reasoning Framework to infer triples of the form *?flow :allocatable ?cluster* by comparing the requirements of the input application graph with the characteristics of the available resources. The latter allows retrieving all the information required to deploy an application with a simple SPARQL query that can be executed promptly. Figure 8 depicts part of a deployment graph where the flows of the given application have been connected to the target cluster, as indicated by the "Allocatable" edges of the graph.

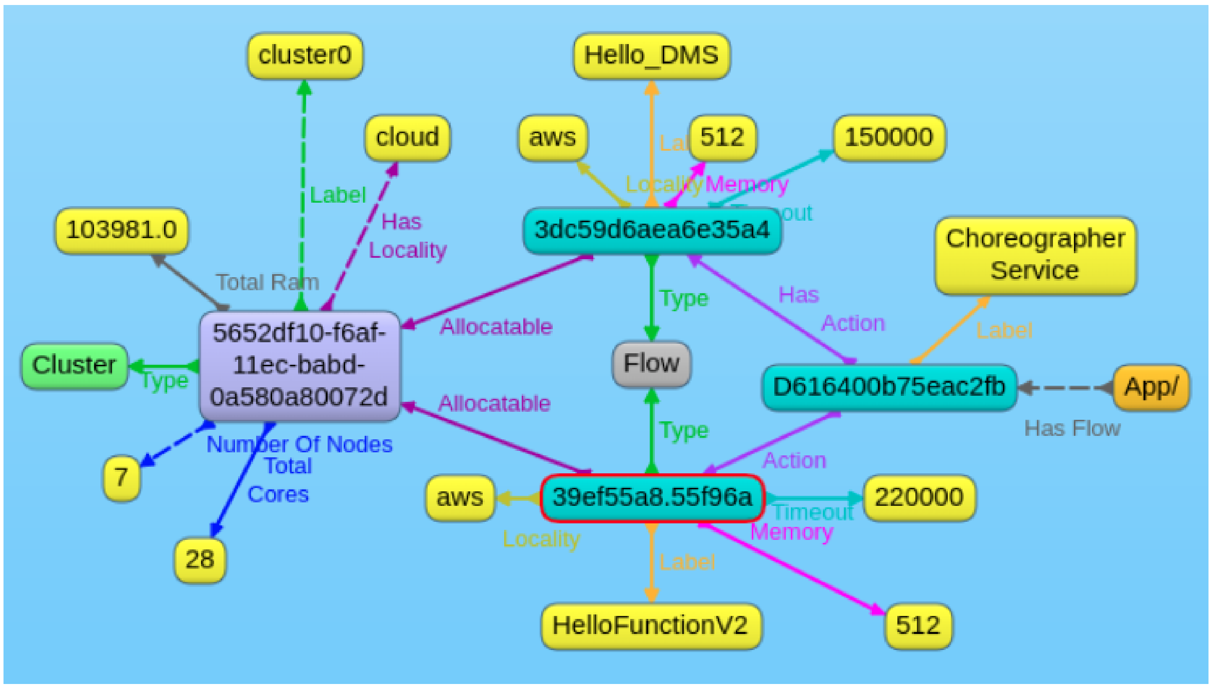


Figure 8: Deployment Graph: The edges "Allocatable" indicate the graph nodes of type "Flow" that could be hosted at the cluster with id "5653...72d".

In Table 2 can be observed the mean response time per request for the most used endpoints of the Reasoning Framework (Table 3). According to these experimental results, the optimized setup (leveraging semantic rules) requires more time for storing the input resource and application data by 45.59% and 15.57%, respectively. This is expected behavior since, during data ingestion, we perform additional operations to the AllegroGraph. On the other hand, query answering latency was significantly reduced for all the “GET” endpoints when utilizing KG’s capabilities.

Table 2: Reasoning Framework API Description

ID	Method	Path/URI	Description
EP1	POST	/cluster	Store a new cluster
EP2	GET	/cluster	Specs of stored clusters
EP3	POST	/app	Store a new app
EP4	GET	/app	Descriptions of all stored apps
EP5	GET	/app/run/<app_id>	Workflow allocations and manifest

Table 3: Reasoning Framework's Response Latency

ID	Baseline (ms)	Optimised (ms)	Change (%)
EP1	**68.99**	100.45	45.59
EP2	28.32	**6.58**	-76.76
EP3	**155.06**	179.20	15.57
EP4	26.81	**21.74**	-18.91
EP5	166.42	**110.71**	-33.48

It is noted that platform components will mainly call Reasoning Framework for requesting data (i.e., GET), while the posting of information happens in sparse intervals. For instance, cluster registration on the platform typically happens once and can be updated at some point in time if the cluster size is altered. On the other hand, requests for cluster specifications occur at each application deployment. Hence, the latency of the GET endpoints is of importance and can be further reduced by implementing additional rules following the requested data.

2.3.2 Runtime Adaptation

Towards evaluating the Forecaster component for runtime adaptation, OW Monitor and Router patterns developed in the scope of WP3 were employed in building an adaptive routing service to dynamically route function requests in hybrid environments, optimizing for function latency. The Monitor, Forecaster, and Router operate in tandem, continuously adapting to the environment's dynamic nature. The architecture of this setup is depicted in Figure 9. The project's testbeds (i.e., AWS and AZURE OpenWhisk clusters) were used for serving the evaluation experiments.

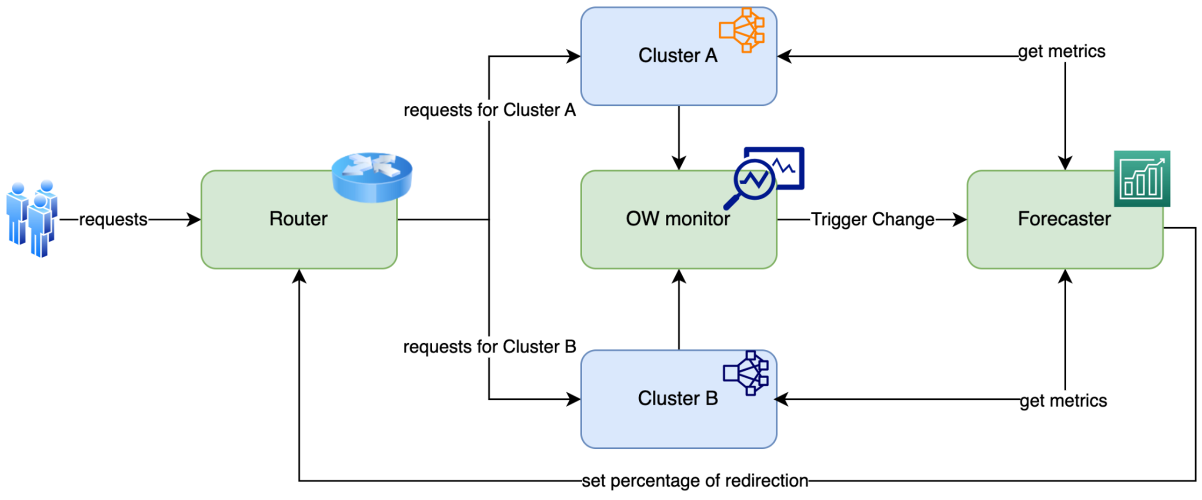


Figure 9: Architecture of RuntimeAdaption Evaluation setup

Moreover, the evaluation process utilized two Python-native functions with differing computational demands: a function to calculate Fibonacci numbers and another to generate a large list. The Fibonacci function is tasked with determining the n-th number in the sequence, while the List function generates a list with a length equal to the given n. The functions operate with default input parameters: n=40 for the Fibonacci function and a list length of 10 million.

The experimental setup is devised to scrutinize both the performance and cost-efficiency of the forecaster and the provided adaptability at runtime. The setups are detailed as follows:

Performance-based Routing

1. Deployment of the Fibonacci function with an initial routing strategy favoring AWS (100-0) coupled with a 10-minute monitoring window interval, polled every 30 seconds.
2. A baseline experiment deploying the Fibonacci function with a static 50-50 routing strategy.
3. A setup involving both the Fibonacci and List functions, initiated with a 100-0 routing strategy and a 10-minute monitoring window interval.
4. Establishing a baseline with both functions and a fixed 50-50 routing strategy.

Cost-based Routing

1. An evaluation of the cost implications utilizing both functions, a 100-0 initial routing strategy, and a 10-minute monitoring window interval. The cost was calculated based on the following rates: for AWS, $0.0000166667 per GB-second and $0.2 per 1M requests; for Azure, $0.000016 per GB-second and $0.2 per 1M requests, based on available rates of the Amazon Lambda[5] and Azure Functions[6] (pay-as-you-go) services respectively. The results of this approach were juxtaposed against the fifth setup in the performance-based routing.

Alarms from the monitor to the forecaster towards re-estimating the routing strategy are triggered when the overall waitTime in a cluster exceeds 3 seconds. Throughout all experimental scenarios, we maintained a consistent request rate of 10 invocations per minute for every function, to facilitate a controlled evaluation of the system's capabilities.

2.3.2.1. Average Latencies Analysis

The experimental results, illustrated in Figure 10, elucidate the performance metrics accrued from different routing strategies implemented during the testing phase involving Fibonacci and combined (Fibonacci and List) functions. It is noticeable from the outcomes that the scenarios utilizing both functions generally exhibit lower average latencies compared to those relying solely on the Fibonacci function. This phenomenon is attributed to the significantly lower execution latency of the List function, thereby positively influencing the average latency values in the combined setup.

[5] https://aws.amazon.com/lambda/pricing/
[6] https://azure.microsoft.com/en-us/pricing/details/functions/

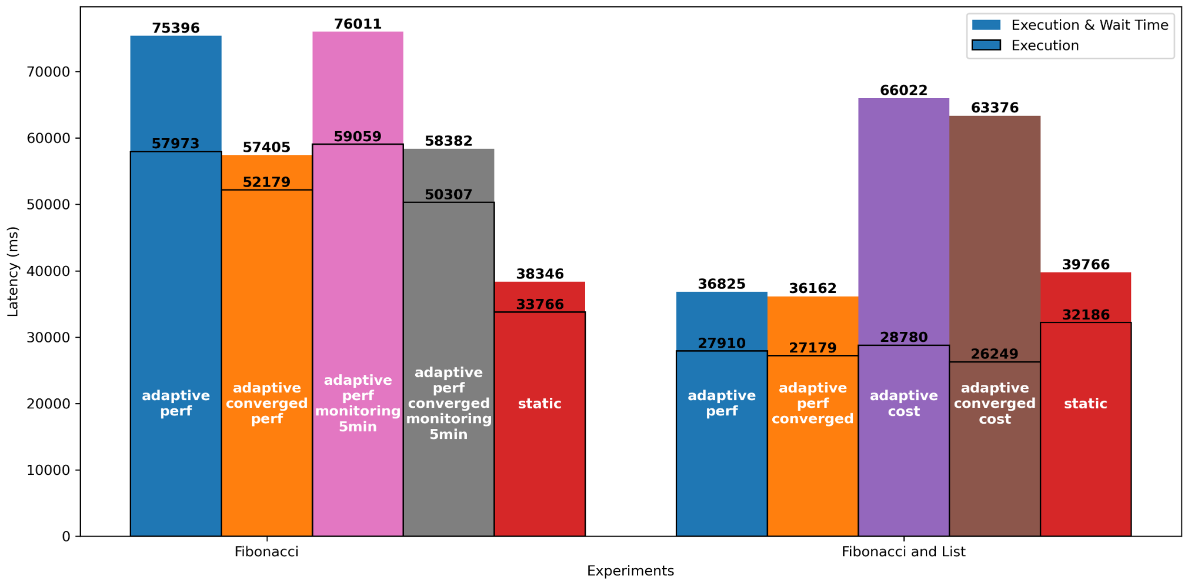


Figure 10: Mean Response and Execution Latency between different experiments

Focusing initially on the Fibonacci function deployment, we observe the performance-based adaptive strategy, commencing with a 100-0 routing, encountering elevated latencies initially, with average response latency and execution time noted at 75396 ms and 57973 ms, respectively. Meanwhile, the static strategy, operating on a 50-50 routing configuration, showcased lower initial latencies, albeit without the benefits of adaptive optimization (Figure 11 and Figure 12).

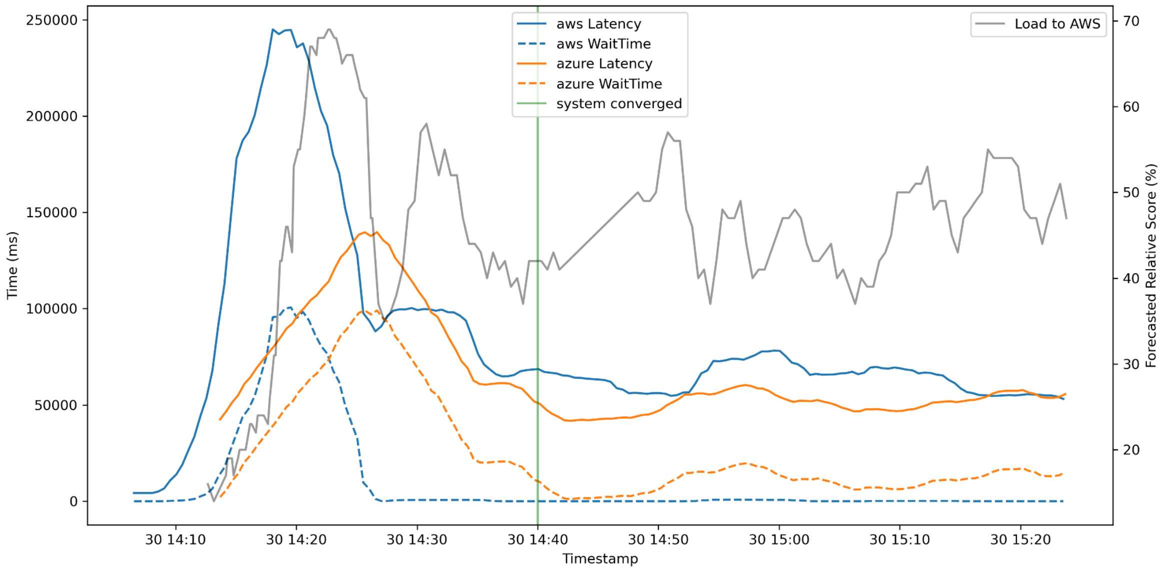


Figure 11: Wait and response latency with Fibonacci function and performance adaptation

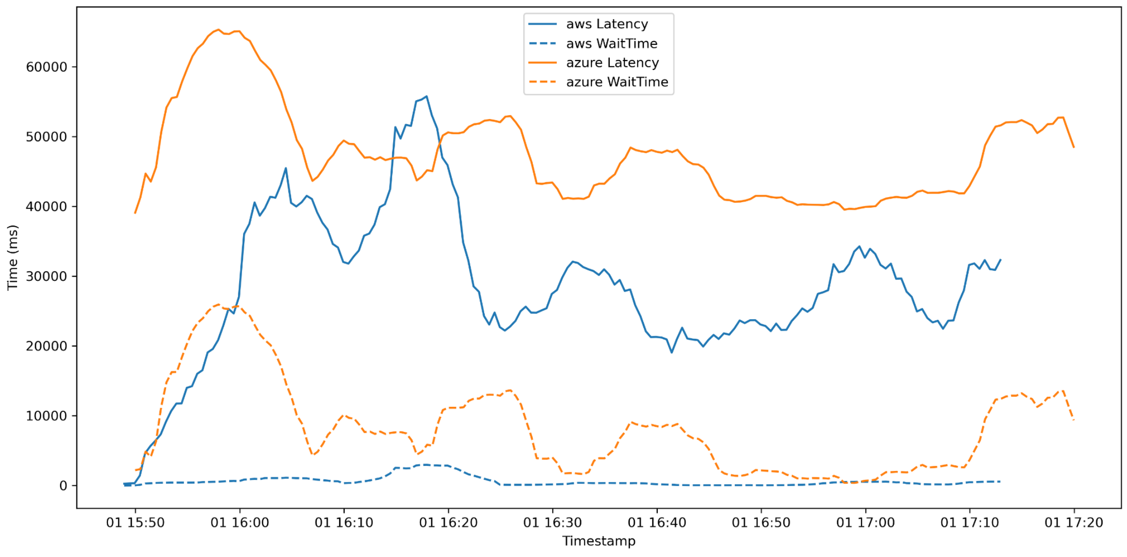


Figure 12: Wait and response latency with Fibonacci function and 50-50 routing policy

Extending the deployment to include both the Fibonacci and List functions, the performance-based adaptive strategy persisted in demonstrating higher introductory latencies (36825 ms response latency and 27910 ms execution time). Yet, it exhibited notable performance improvements upon reaching convergence with 36162 ms and 27179 ms in respective categories (i.e., 9% lower latency compared to static routing.

2.3.2.2. Variance Analysis

The standard deviation (STD) provides insights into the variability of the performance metrics, which, in the context of latency, can be critical for applications demanding consistency. Furthermore, as mentioned in [12], the stability in execution times is critical given the nature of the FaaS cost model depending on execution time. The results, depicted in Figure 13, furnish insights into the standard deviation of the response latency across different setups.

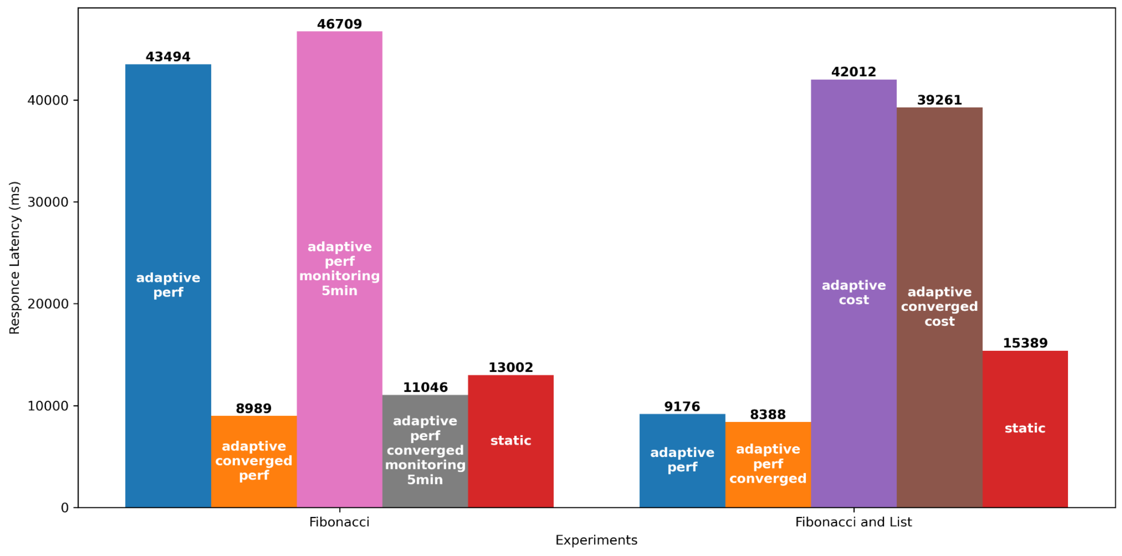


Figure 13: Standard Deviation of Response Latency between different experiments

For the Fibonacci function deployment:

- The adaptive strategy exhibited high variability initially with an STD of 43494 ms for response latency. However, this drastically reduced after convergence to 8989 ms.
- The static strategy, in this scenario, noted around 30% higher variability in latency compared to the converged version of the first setup.

For the combined Fibonacci and List functions deployment:

- The adaptive strategy had an STD of 9176 ms (response latency) which upon convergence, this value reduced slightly to 8388 ms.
- The static strategy, in this scenario, noted around 45% higher variability in latency compared to the converged version of the performance-optimized adaptive setup.

2.3.2.3. Cost Analysis

Table 4 illustrates the cost analysis for each experimental setup, denoting both the initial and converged costs. The costs are presented in terms of the mean cost per minute, which is calculated based on the average execution time and the number of requests processed each minute in each cluster for functions' memory of 512Mb.

Table 4: *Cost per Experiment*

Experimental Setup	Cost	Cost-Converged
Fibonacci & List, Performance Opt	**2.12**	2.05
Fibonacci & List, Cost Opt	2.81	**1.86**
Fibonacci & List, 50-50	2.25	2.25

The results affirm that the adaptive system, particularly in the cost-optimized setup, achieves the most cost-effective operation, reducing the mean cost per minute substantially (up to 17%) post-convergence. This aspect accentuates the adaptive system's capacity to enhance performance and ensure a cost-efficient operational framework.

Having evaluated the performance across different configurations, a couple of key insights emerge. It is noticeable that the adaptive strategies, whether prioritizing performance or cost-efficiency, tend to offer advantages over a static approach, highlighting their flexibility.

Upon convergence, the adaptive strategies demonstrate notably reduced latencies and costs, albeit at the expense of higher initial latencies. However, starting from a 50-50 routing would allow the system to converge faster, eliminating the initial performance/cost variances.

Furthermore, the deviation in latencies, as analyzed, points to a greater consistency achieved post-convergence, especially in the performance-optimized setups. While the adaptive setups exhibit pronounced fluctuations initially, indicative of the system's efforts to find an improved state, the post-convergence period witnesses a stabilization, thereby underscoring the adaptive system's efficacy in mitigating inconsistencies during runtime.

In light of the cost analysis, it is apparent that adaptive strategies steer towards enhanced performance metrics and drive down the costs significantly when compared to a rigid 50-50 strategy, thereby illustrating a dual advantage.

Conclusively, the results underscore the potential of adaptive strategies in orchestrating a system that is not only agile and responsive but also cost-efficient, paving the way for runtime adaptation in multi-cluster FaaS deployments that harmonize performance optimization with economic viability.

3. Cloud Services and Edge Devices Performance Evaluation

3.1. Functional description

3.1.1 Main objective and functionalities

The Performance Evaluation Framework (PEF) of T4.2 aims to offer functionalities to investigate performance issues in the context of the PHYSICS platform. As an example, it can be used to evaluate diverse locations or modes for the execution of a function in the context of a FaaS platform. To this end it incorporates tools that include load injectors, automating their launching, configuration, monitoring, result retrieval and presentation of the acquired tests, as well as performs several performance analyses in relation to the way to deploy and orchestrate functions. It also provides support services for the other processes of the PHYSICS platform to support performance, like in the case of the Performance Pipeline.

Another set of functionalities has been considered and related to the need for other components like patterns to be configured and operated during runtime. As mentioned in D3.2, patterns are ready-made, reusable flows offered to the developer that aim to address a specific problem in a parametric manner. However, the parameters used to instantiate it may significantly affect a pattern's benefit. To this end this component will be used to investigate performance issues when applying a pattern, as well as relevant trade-offs of their parameter setting. In this case, either the typical launching of an experiment may be performed via the REST API of the tool, or a relevant performance model may be created following the measurement process, that will also be able to interpolate for cases that have not been benchmarked.

Therefore, the overall functionality of this component includes:

- The ability to launch a given test towards the FaaS platform in consideration to evaluate the effect on a given workload (user function or benchmarking function from a typical benchmark) or a given execution mode (as the ones defined in WP3). To this end, collaboration and support towards the other components of the PHYSICS platform (e.g. from the Design Environment Performance Pipeline described in D3.2) needs to be provided
- Provide custom loads or clients needed in a FaaS environment based on the specifics of this environment management and operation processes, adapting to the peculiarities of the environment. This includes for example different technologies of load injectors (blocking or non-blocking) as well as adapted parametric client implementations needed for specific experiments
- Give the ability to create arbitrary performance models that may aid in the selection, operation and runtime management of the system or the application.
- Offer the ability to other components to retrieve the performance measurements and/or query during runtime.

The overall use case diagram of the PEF appears in Figures 14, 15 and 16.

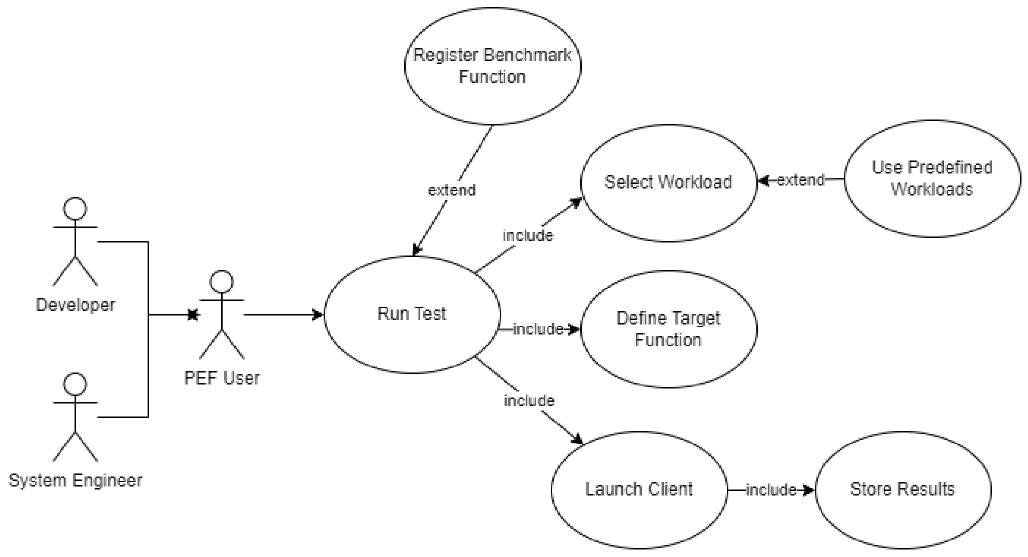


Figure 14: PEF Run Test Use Case diagram

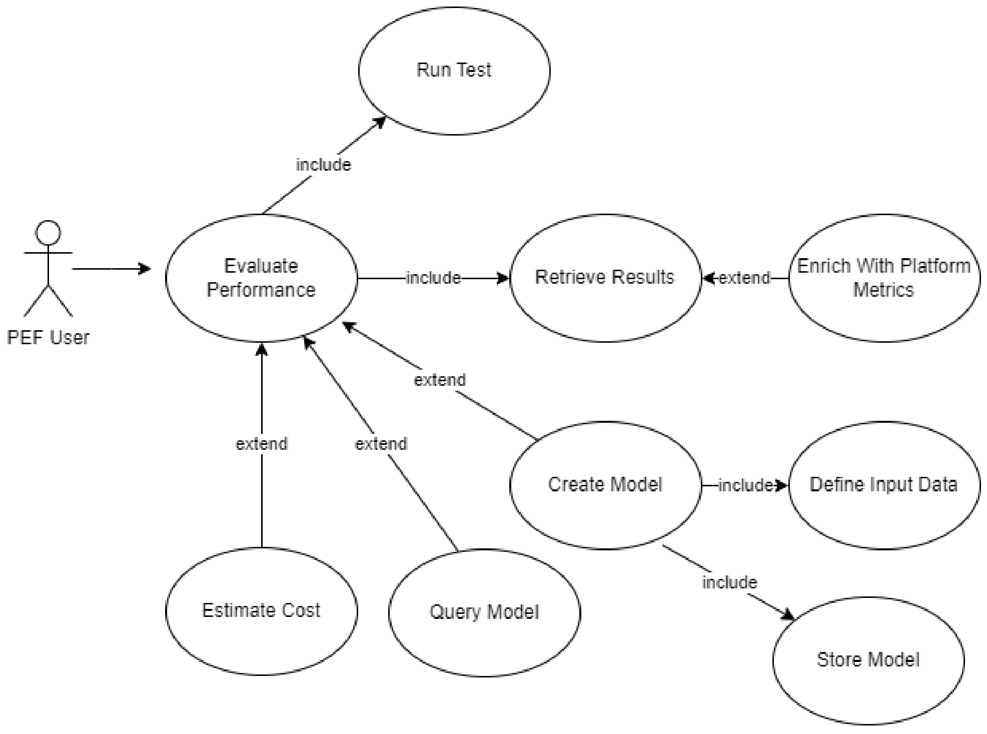


Figure 15: PEF Evaluate Performance Use Case diagram

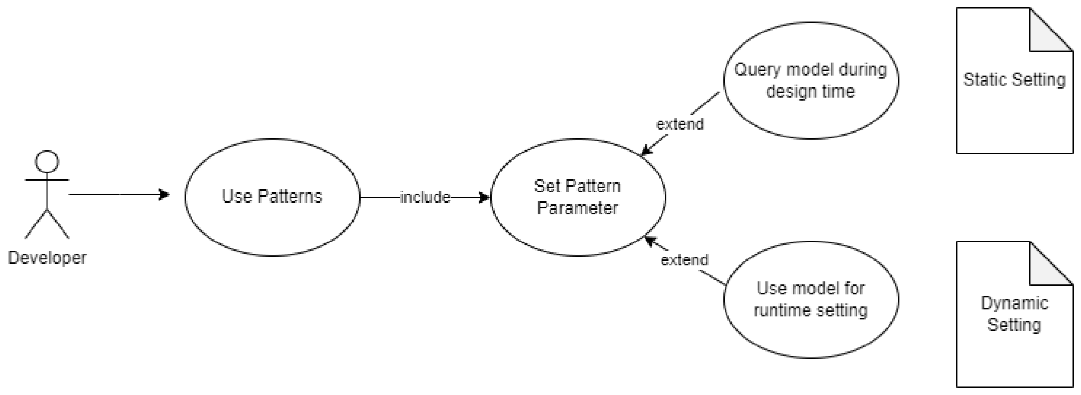


Figure 16: Pattern Modelling using PEF Use Case

Therefore, the PEF can be used as the main testing platform for load generation, given that all of its operations are offered via REST API calls and thus can be added to any arbitrary measurement and evaluation process.

3.1.2 Relationship with other components

As indicated in D2.5, the acquired evaluation information from the previous steps can be provided via REST APIs to several components (such as the Semantic Extractor of WP3 for enriching the application graph with function benchmark information or T5.1 services for a macroscopic, historical view of a cluster performance) as well as provide relevant statistical performance data towards T4.3 to be used in the global optimization process (as a resource capability identification and investigation of the performance effects of various cloud-edge interplay scenarios). PEF results have also been integrated with the Design Environment of WP3, therefore, it can be used by the developer to gain insights or by the other system components to evaluate the effect of strategies and management practices.

3.1.3 Requirements matrix

With relation to the expressed requirements in D2.3 and the functionalities described in the introduction of this chapter; this component is involved in the following:

- Req-4.2-FaaSBenchmarking
    - undertake the full lifecycle of benchmark execution (i.e. launching of the benchmark orchestration of its operation, gathering of results) in a Benchmarking as a Service style
- Req-4.2-CostAssociation
    - associate performance metrics attained through the benchmarking executions with the relevant cost models available in the FaaS domain
- Req-4.2-MeasurementPropagation
    - availability of results or predictions through a REST API

3.2. Technical description

3.2.1 Baseline technologies and dependencies

Experimentation for modeling cloud applications and service performance is time consuming. Especially when the generated load needs to scale, manual operations are needed for configuring distributed client resources, orchestrating test execution and gathering results. The main baseline technology initially used in the PEF was the Flexibench (or Application Dimensioning Workbench [13]) tool that we have developed in the context of the H2020 BigDataStack project [14]. Flexibench configured, spawned and managed virtualized container clusters serving the load injection process, in a parameterized, on-demand manner. It exploits adapted images of mainstream injection clients (Jmeter[7]) on container platforms. Given the discrete parameterization of each virtual cluster, isolation and separate test-level control, as well as REST based interfaces, benchmarking as a service is accomplished, reducing the knowledge barrier for application engineers.

However, during the second period of the project, it became apparent that having a separate managed cluster just for the load injection process was neither practical nor easily integrated with the remaining platform, given the focus on FaaS technologies. For this reason, a reengineering of PEF was performed, with the aim to offer as many functionalities as possible through functions that are easily deployed and executed within the context of the platform or can participate in project-wide functionalities such as the Performance Pipeline described in D3.2. This was also dictated by the asynchronous nature of FaaS APIs, which creates complications for synchronous, blocking clients like Jmeter.

The parts from Flexibench that were kept were only the model creation process as well as specific Jmeter clients that were used for parts of the experimentation where tighter control is needed (e.g. for the cold/warm/hot measurements or the multitenancy considerations in the collocation experiments of D5.2). The modeling process [15] aims to integrate the data acquisition with model creation and is based on a Genetic Algorithm optimized ANN generation that can create models as universal approximators. The models are stored locally and can be retrieved and queried during an online model inference process. Once a model is created, a separate function container can be easily created if we want to decouple the model querying process from the main service and achieve greater scales that are only limited by the size of the available FaaS cluster.

The PEF framework is based on the following baseline tools:

- Node-RED as the main development tool, for both the main service and the created functions
- Containers, as a means of packaging the created functions
- GNU Octave as the main model creation environment

Through this re-engineering of PEF, in the context of the PHYSICS project, the Performance Evaluation Framework has achieved:

- Easier deployment of the main component service, as well as the supporting functions

[7] https://jmeter.apache.org/usermanual/component_reference.html

- Extension of the result acquisition API, to cover for the project needs and relevant query results
- Creation of FaaS-specific load generators, both in terms of the measured quantities of the benchmarking process as well as the execution of the load generators themselves as functions, thus achieving scale that is only limited by the size of the available FaaS cluster
- Extension of the scope of PEF to offer functions for clustering relevant function profiling results to categories and classifications based on the latter
- Creation of specialized and parametric workload setups that are tailored to the FaaS concepts and execution models (e.g., cold and warm executions)
- Alteration of the performance model inference process, which has been extended to be performed through a FaaS execution model
- Application of the measurement and modeling processes to a wide set of performance issues regarding patterns, modes of execution, modes of orchestration etc.
- Integration with the PHYSICS platform components and processes

3.2.2 Component internal architecture

The overview of the PEF has been extensively updated in the second period of the project and its final form appears in Figure 17. Initially, the API layer is responsible for receiving the various requests, which can be categorized into four main operations (excluding variations of the API calls for specific purposes)

- Launch a specific test, defining options such as the target function, the desired set rate, test input, number of function clients, etc.
- Push results that are acquired from the execution of one of the provided functions (Load Generation, Classifier, Clustering). There is no direct storing from the created functions to the Performance DB to reduce the configurability needed for the functions and enable the developers to manipulate the sequence of the calls more freely (e.g. filter out from the results any null or otherwise erroneous values that are detected). Thus the Performance DB includes data from all the relevant entities handled by PEF (more information is included in Section 3.2.6. on the PEF Data Model.
- Retrieve results that have been pushed in previous executions, based on a variety of criteria (e.g. function name, location, etc.)
- Create a defined performance model, indicating experiment series names from which the baseline data will be extracted for model training.

These interfaces are exploited for example in the context of the deployment process, in which the Semantic Extractor from WP3 queries PEF in order to populate the app graph with the benchmarks and profiles of the included functions in the graph. Furthermore, they are exploited in the context of the performance pipeline, which orchestrates the execution of the various functionalities provided by the PEF (more information can be found in the according Deliverable D3.2). One interesting aspect of PEF is also the ability to incorporate WP3 subflows that have been created and can be used both at the application level as well at the platform level. Such examples can be considered the Request Aggregator, which is a service offered by PEF and has been implemented for usage by the eHealth Use Case, or other functionalities like the OpenWhisk monitor and Router subflows that have been used in the context of the experimentation described in Section 2.3.2. Given that many outcomes of PEF like the Load Generator Flow have been created as independent Node-RED flows, they can be used either within the context of PEF or from any external typical Node-RED environment, a feature that significantly decouples the dependencies in the component as well as increases the usability of the various artifacts.

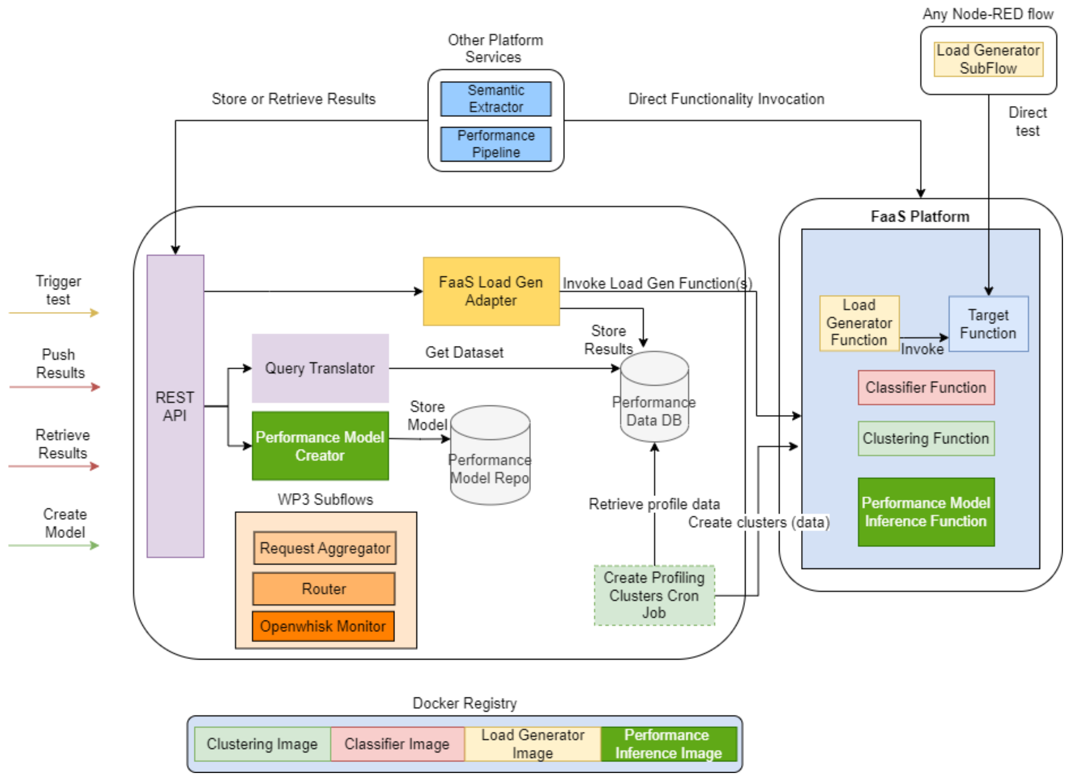


Figure 17: Performance Evaluation Framework component internal architecture

3.2.3 Interfaces, adaptations and integrations

3.2.3.1. PEF APIs

The PEF functionalities are provided in the following tables. Initially, in Table 5, the high level operations and functions are portrayed, while in Table 6, we include the detailed methods against the Performance DB data that are made available.

Table 5: API operations for PEF process execution

Method	Path/URI	Description
POST	/launchTest	Submit a test to be executed through PEF. A test may have multiple inner combinations (different inputs) or iterations
POST	ow-endpoint/ loadgen	Triggering the load generation function directly on OpenWhisk.
POST	ow-endpoint/ create-clusters	Trigger the cluster creation function directly on OpenWhisk

POST	ow-endpoint/ classifier	Trigger the classification function, providing the cluster centers and this function's profile inputs directly on OpenWhisk

Table 6: PEF DB API operations for test results, storage acquisition and filtering

Method	Path/URI	Description
POST	/loadgendata	Save load generator's output data
GET	/loadgendata	Retrieving the load generator's data
GET	/loadgendata/:flow/:branch	Retrieving the load generator's data of a specific flow in a given branch
POST	/data	Save the Prometheus trace data (average values) from the target function execution
GET	/data	Get all trace data. To be used by the cluster creation process
POST	/clusters	Used by the Clustering process to store the clustering centroids
GET	/clusters	Get the most recent clustering centroids. Used for retrieving the clusters as inputs to the Classifier function
GET	/clusters/:endpoint	Get the most recent clustering centroids for a given location. Used for retrieving the clusters as inputs to the Classifier function
GET	/clustersall	Get all the clusters saved on a database. Used to analyze the evolution of the centroids across time, as the DB gets more populated with profiles
POST	/profile	Used to store the classification for a given function
GET	/profile	Get all relevant profiles from PEF
GET	/profile/: actionName	Get classification data for a function that are stored into PEF. It returns the most recent classification for the function version that is defined in the input, grouped by available locations
GET	/profile/: actionName/:location	Get classification data for a function that is stored into PE. It returns the most recent classification for this function for the defined location in the input
GET	/flowprofile/:flowName/:branch	Get classification data for a flow name that is stored into PEF. It returns the most recent classification for this flow, grouped by available locations. The difference with the similar /profile/:actionName call is that there is no need to include the concrete actionName (which is subject to versioning within the PHYSICS platform e.g. HelloFunctionV2_george_8d3d8d55-90ef-4a17-b

		668-30bcfcb797f1.json) but the user can directly use the flow name used within their Node-RED environment (e.g. HelloFunctionV2)

3.2.3.2. Adaptation to FaaS based execution for Model Querying process

The model query mode described in the previous section was originally implemented as a REST service that raised one container per request in the context of BigDataStack [14]. Although this gives significant packaging improvements compared to a typical threaded web service, the performance degradation due to the extensive use of separate containers per request is significant and has been documented in the relevant application of the Batch Request Aggregator pattern [16] that was described in D3.1, since the latter was applied to this as a test service.

Applying the Batch request aggregator has helped improve the performance footprint, however in the case of adaptation we considered it is important to check also the "FaaSification" of this process, i.e. transforming the model querying process to a function execution. For adapting the respective implementation, the baseline-modeling image, consisting primarily of the GNU Octave environment and relevant scripts for interacting with the model creation and querying process, was enriched with the Node-RED runtime. In the initial version, the startup scripts of the image acquired the arguments for execution (model ID and needed estimation inputs) through relevant environment variables that were passed to the initializing container from the Flexibench environment through the typical Docker Service API.

In this version, the adaptation needs to be altered (Figure 18) and a supporting Node-RED flow (Figure 19) needs to be created, taking advantage of the OpenWhisk (OW) skeleton pattern described in D3.1, to pass the arguments as OpenWhisk function arguments and adapting the relevant script inputs. Now the arguments are passed as command line arguments to the script following an adaptation in the Node-RED flow, after obtaining them through the OpenWhisk argument interface (body of the POST /run call). This is needed for functional correctness, to cover also for the cases of warm container reuse, as mentioned in the Context Reuse issue described in D3.1. In this manner, each new request is based on the updated arguments that arrive in the new message towards OW invocation.

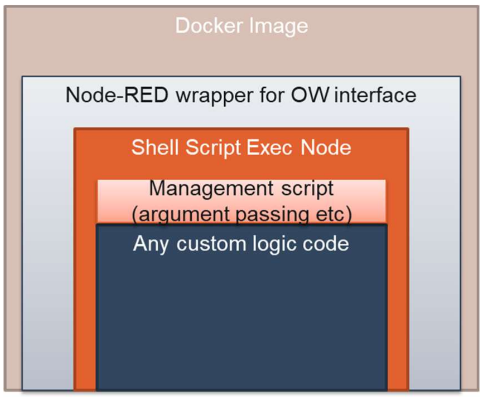

Docker Image
Node-RED wrapper for OW interface
Shell Script Exec Node
Management script (argument passing etc)
Any custom logic code

Figure 18: Stack layers needed for the adaptation of the modeling image for function execution

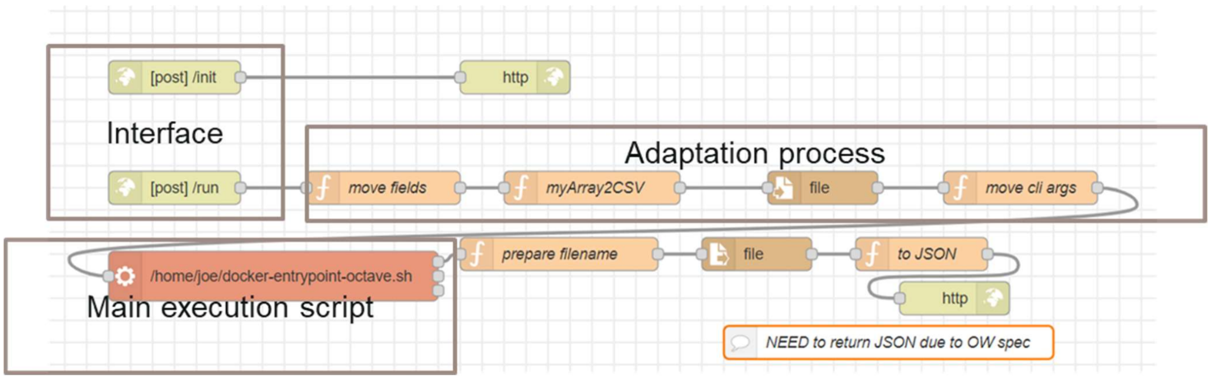
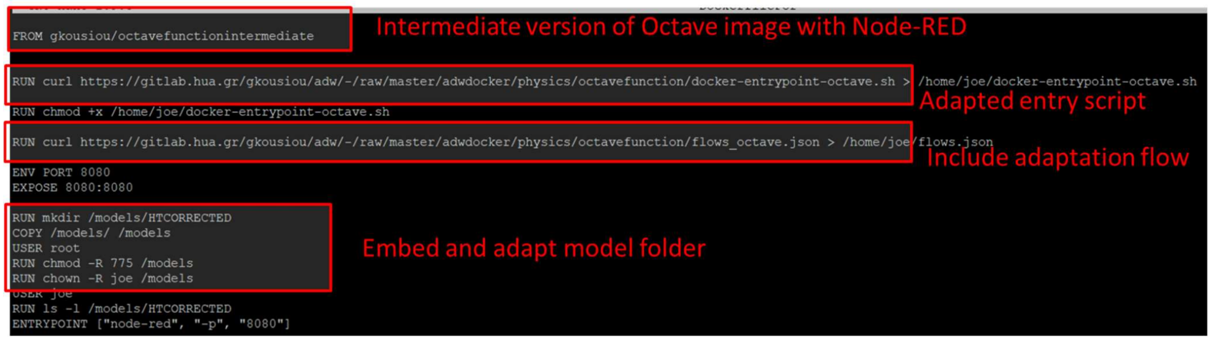


Figure 19: Node-RED wrapper flow for adapting the model inference process to a serverless execution

The relevant Dockerfile is needed for the creation of the baseline function image (Figure 20) appears in the following snapshot as an example.



```
FROM gkousiou/octavefunctionintermediate

RUN curl https://gitlab.hua.gr/gkousiou/adw/-/raw/master/adwdocker/physics/octavefunction/docker-entrypoint-octave.sh > /home/joe/docker-entrypoint-octave.sh
RUN chmod +x /home/joe/docker-entrypoint-octave.sh
RUN curl https://gitlab.hua.gr/gkousiou/adw/-/raw/master/adwdocker/physics/octavefunction/flows_octave.json > /home/joe/flows.json
ENV PORT 8080
EXPOSE 8080:8080

RUN mkdir /models/HTCORRECTED
COPY /models/ /models
USER root
RUN chmod -R 775 /models
RUN chown -R joe /models
USER joe
RUN ls -l /models/HTCORRECTED
ENTRYPOINT ["node-red", "-p", "8080"]
```

Intermediate version of Octave image with Node-RED
Adapted entry script
Include adaptation flow
Embed and adapt model folder

Figure 20: Dockerfile needed for the creation of the baseline function image

After the creation and pushing of this image to a registry, the image needs to be registered with OW with the following command (Code 3):

Code 3: Registering the image with OW using the create/update command

```
wsk action create/update action_name --docker account_name/image_name
```

The example image has been uploaded on the Docker registry [17].

3.2.3.3. Integration of performance models in patterns- the example of the Batch Request Aggregator

The issue of performance management in cloud environments has been extensively analyzed in recent years. Approaches around auto-scaling, through direct Proportional Integral Derivative (PID) controllers [18], ANN based application prediction models [19], feedback loop-based Quality of Service (QoS) prediction through optimization techniques such as Particle Swarm Optimization [20] focus on increasing application related resources to enhance Key Performance Indicators (KPIs). Increasing resources is of course a valid choice, however it is associated with extended costs. Other works examine interference caused by concurrency, either at the Virtual

Machine (VM) [21] or the container level [22] (including intra or inter container overheads) as well as the introduction of microVMs [23]. Alternative approaches try strategies such as dependency packages aware affinity scheduling [24] or library sharing [25]. Approaches such as the Request Aggregator pattern described in D3.1 target primarily at better utilizing existing resources in order to improve application performance, without having to increase them.

As mentioned in D3.1, the Batch Request Aggregator pattern is an approach that assembles incoming requests, merges them and forwards them as one array request to the backend. This affects the number of containers used to serve the request, thus alleviating the back-end stress. How the batch is released is determined by the release logic. Simple timeout strategies can be applied or static settings of batch sizes. However, it is evident that the frequency of request arrivals plays a vital role in this decision. For example, if 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 and back-end contention. Thus, one needs to regulate the parameters of the pattern based on the current conditions of execution to optimize the overall result. To provide such a regulation, the model creation process of the PEF was used for creating a relevant model, as shown in Figure 21. This takes as input the measured current request frequency (provided by the pattern) and the set batch size to predict the response time. Based on a set of multiple predictions, the model can determine the best configuration of the batch size. The target service for regulating was another instance of the PEF running on a different node and used for model inference (query model operation).

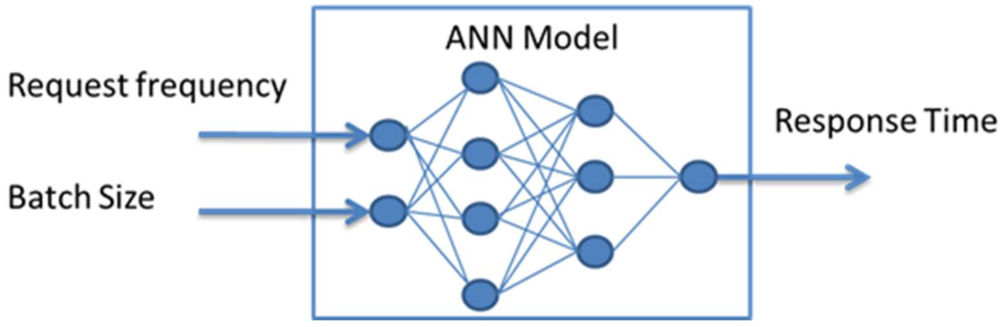


Figure 21: Model creation process of the PEF

The overall process appears in Figure 22. PEF B is the target service to model, in front of which a Batch Request Aggregator is applied. PEF A performs the tests needed to collect the dataset for training. PEF A is also used to create a model for PEF B querying process and deploys the model so that the Batch Request Aggregator can query it during runtime.

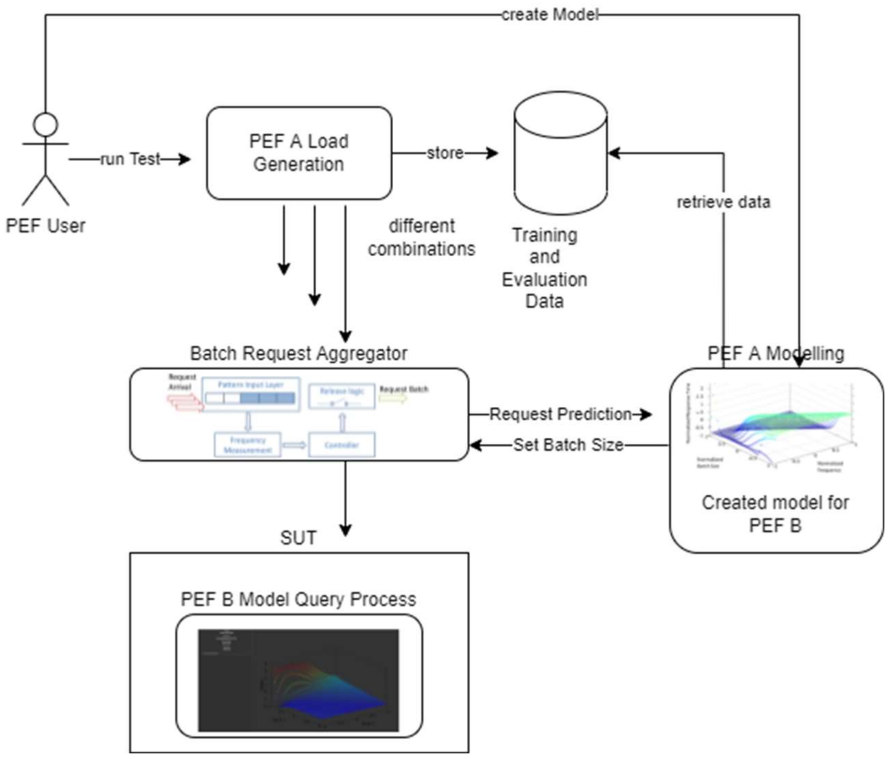


Figure 22: PEF overall process

The analysis of the data acquired from the various test executions that result in the dataset creation is included in D3.1 to demonstrate the effect of the Batch Request Aggregator on different static conditions. Model evaluation and runtime use are included in this chapter's Experimentation section.

3.2.4 Function Design Approach Details

3.2.4.1. Load generator Function/Flow

To adapt the PEF to the testbed as well as enhance the inclusion of set rate clients, the tool was extended to include load generators not based on Jmeter. Jmeter is a mature load generation tool which however has one major drawback. It is based on synchronous calls per thread, which means that the clients can not sustain a specific set rate unless the server also supports the specific rate. Thus, Jmeter clients are more directed towards simulating users but not specific requests per second. Furthermore, the peculiarities of the FaaS environment, including for example strict timeouts in blocking call invocation delays, means that synchronous calls need to be converted after a while to polling calls for the outcome of the request.

In order to avoid these situations and better adapt to the FaaS system result retrieval, a new load generator was created, based on Node-RED (Figure 23). The latter can either be used directly inside any Node-RED environment, running as a flow whenever a relevant benchmarking process is needed or can be wrapped as a function and executed on an OpenWhisk backend (Figure 24). In the former case, Node-RED is limited by a max message throughput of approximately 300 messages/sec. The adaptation to a function execution thus enables the increased scalability of the load generation process.

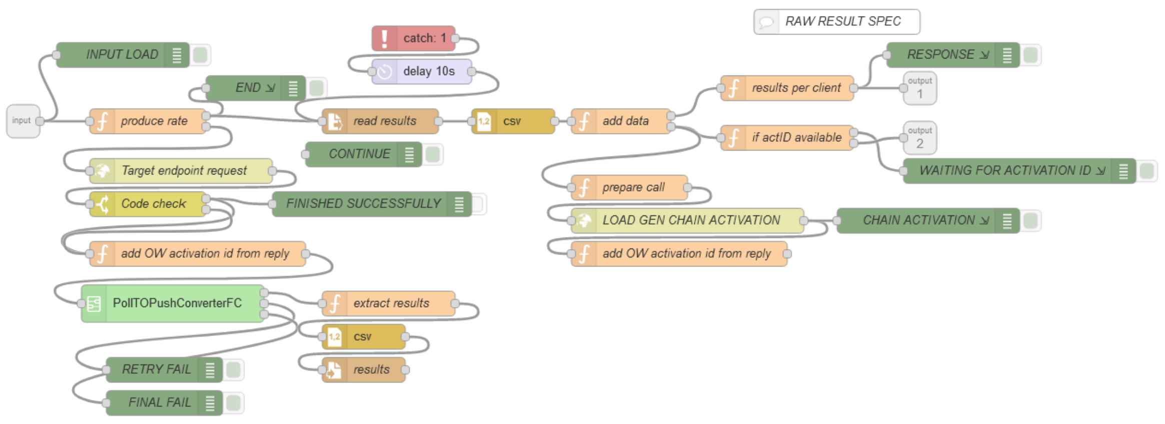


Figure 23: PHYSICS Load Generator Subflow Implementation

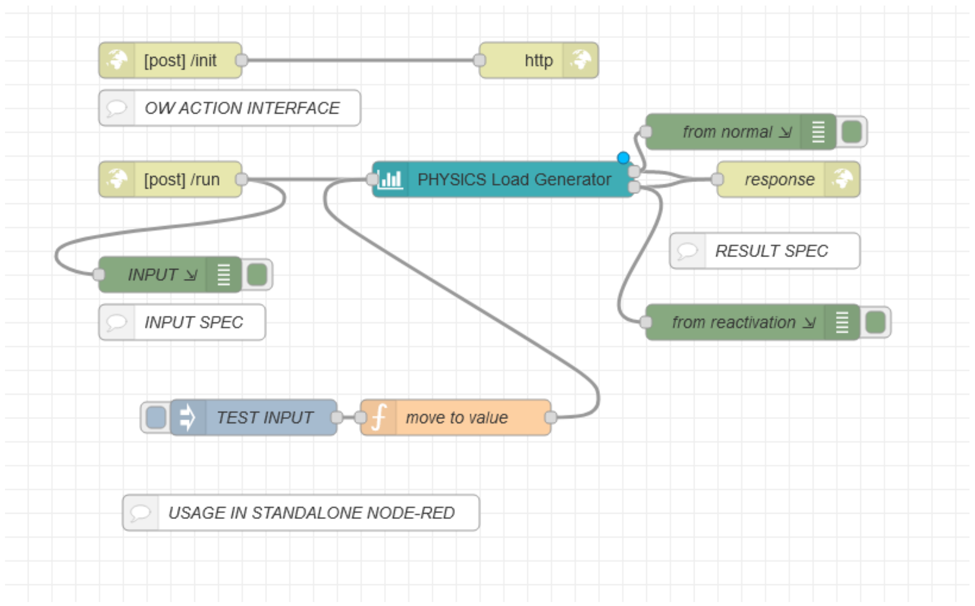
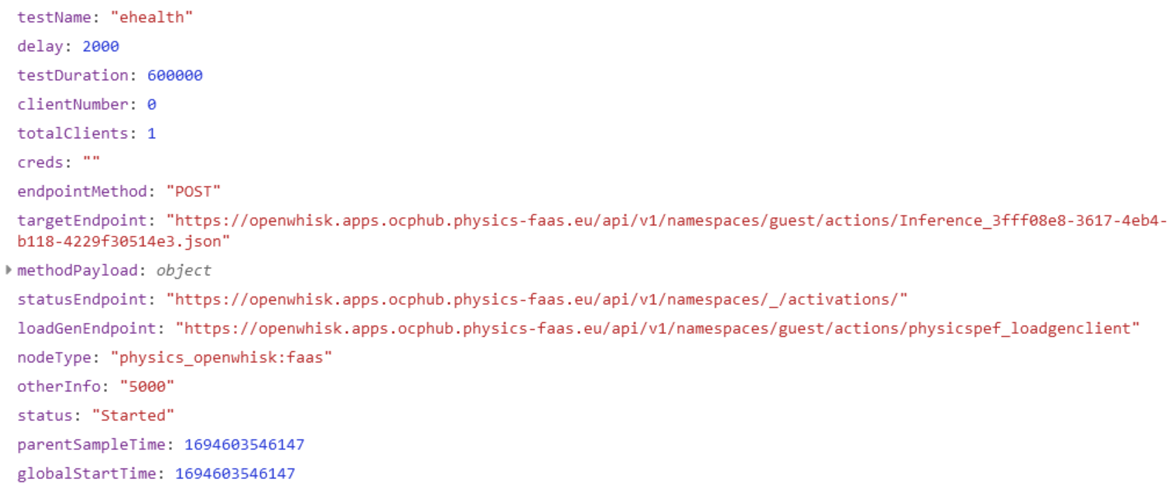


Figure 24: PHYSICS Load Generator Subflow wrapped around an OpenWhisk Interface

Input Specification for the Load Generator

The load generator requires an input JSON that specifies the main options of the execution, which appear in Figure 25.



```
testName: "ehealth"
delay: 2000
testDuration: 600000
clientNumber: 0
totalClients: 1
creds: ""
endpointMethod: "POST"
targetEndpoint: "https://openwhisk.apps.ocphub.physics-faas.eu/api/v1/namespaces/guest/actions/Inference_3fff08e8-3617-4eb4-
b118-4229f30514e3.json"
▸methodPayload: object
statusEndpoint: "https://openwhisk.apps.ocphub.physics-faas.eu/api/v1/namespaces/_/activations/"
loadGenEndpoint: "https://openwhisk.apps.ocphub.physics-faas.eu/api/v1/namespaces/guest/actions/physicspef_loadgenclient"
nodeType: "physics_openwhisk:faas"
otherInfo: "5000"
status: "Started"
parentSampleTime: 1694603546147
globalStartTime: 1694603546147
```

Figure 25: Input Specification for the Load Generator

The most important parameters include:

- Test duration: wall clock duration of the test as set by the user
- Delay: the inter-request delay. This controls the produced message rate
- Total clients used: how many function clients are expected to be used in the test, and what is the id (clientNumber) of this client
- Target endpoint: against which the load generation will be performed, as well as the HTTP method and payload to used
- Status: endpoint from which the results for the target endpoint will be received
- Load generation endpoint: where is the load generator function located (it may be different from the target endpoint OpenWhisk function)
- Nodetype: whether this execution is inside the Node-RED environment or executed as a function. If the latter, the process of function chaining will be applied (see next sections)
- Other info: A generic “other info” field that can be used to add any arbitrary information (e.g. some parameter of the target function that may affect its runtime or some metadata that the user wants to add)
- ParentSampleTime: The timestamp of starting the process (just before the call is made) as a reference point for the measurement points (see next section).

Measurement Points Analysis

In order to support a wide range of metrics and statistics, numerous measurement points have been included in the process or are extracted from the baseline OpenWhisk reports of an action invocation. With the help of these checkpoints, various conclusions can be extracted, either in relation to PEF operation, or the client operation or the target system conditions. An overview of the respective measurement collection points is presented in Figure 26.

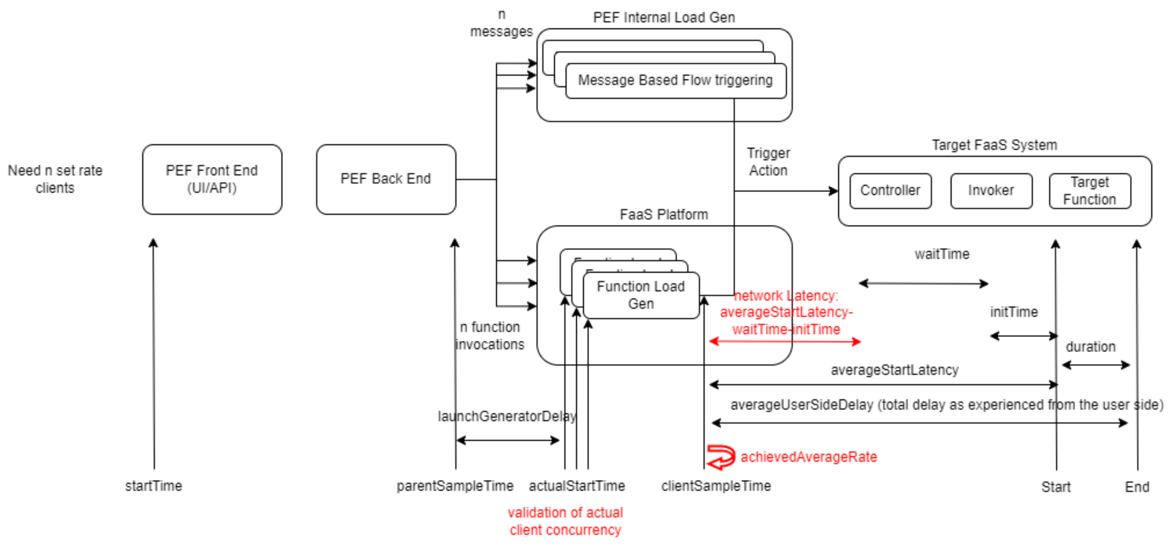


Figure 26:Set rate system measurements point analysis

The extracted output metrics are obtained from the OpenWhisk activation API as well as internal load generator sampling points. The most important of these include:

- Average user-side delay: total response time from the client call to the end response
- Wait time in OpenWhisk: time the request is placed in the queue until a relevant function container pod is available
- Initialization time: time needed to raise a function container in the case one is not already available (cold start)
- Function execution duration: the time it takes to execute a function
- Launch Generator latency: the delay between the start of the experiment and the time at which the load generator function begins to execute
- Start Latency: delay between an invocation launched by the load generator and the actual startup of the function execution
- Network Latency: it is calculated by subtracting the wait and init times from the start latency
- Success percentage: percentage of calls that successfully finished
- Number of cold starts experienced in the run

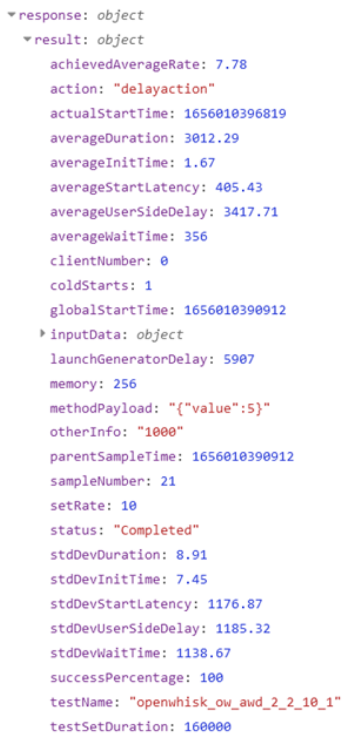
In the case of multiple load generation functions being launched, the user can ensure that all of them were indeed concurrently active in the experiment (e.g. not some of them waiting for a container slot in OpenWhisk) through checking the actual start time of each function. These diverse metrics provide valuable insight into both the function execution and the load generation processes.

Given that these results are typically used to generate e.g. performance models or function runtime predictors, further functional details are logged in order to assist in the dataset extraction for the training of these models. Such details include:

- Function name as well as memory used
- Type of node used for load generation (FaaS or local in NR)
- Target and achieved set rate for each client

- Total number of clients (if multiple load generator functions are used)
- Total samples accumulated per client including the raw data from these requests
- The main inputs for the test such as duration, the "other info", the payload of the target function invocation since this might also include some workload parameters for the target function. These are included so that they are logged together with the results of the execution

An example output of the results of the load generator is presented in Figure 27.



```
▾response: object
  ▾result: object
     achievedAverageRate: 7.78
     action: "delayaction"
     actualStartTime: 1656010396819
     averageDuration: 3012.29
     averageInitTime: 1.67
     averageStartLatency: 405.43
     averageUserSideDelay: 3417.71
     averageWaitTime: 356
     clientNumber: 0
     coldStarts: 1
     globalStartTime: 1656010390912
    ▸inputData: object
     launchGeneratorDelay: 5907
     memory: 256
     methodPayload: "{"value":5}"
     otherInfo: "1000"
     parentSampleTime: 1656010390912
     sampleNumber: 21
     setRate: 10
     status: "Completed"
     stdDevDuration: 8.91
     stdDevInitTime: 7.45
     stdDevStartLatency: 1176.87
     stdDevUserSideDelay: 1185.32
     stdDevWaitTime: 1138.67
     successPercentage: 100
     testName: "openwhisk_ow_awd_2_2_10_1"
     testSetDuration: 160000
```

Figure 27: Load generator Output Result of a Benchmark Run

Function Chain Pattern Approach For Load Generation

A challenge that arises when packaging all these processes as functions is the inherent limitations of FaaS, such as the maximum runtime a function can live. Typically FaaS systems limit that running time to e.g. 15 minutes. Thus to bypass this limitation, appropriate strategies need to be identified and employed. Among the most popular solutions is the Function Chain pattern [26]. In the case of our load generator, we employed this specific pattern, as a developer might specify a test duration that exceeds the maximum runtime of a function. The application of the Function Chain pattern is presented in Figures 28 and 29.

In this case, it is crucial that each function instance in the chain is cognizant of its own lifespan as well as the total duration of the experiment. If a function instance is nearing its end of life, it needs to initiate the successor instance and log the activation id of the successor in its results. This enables the load generation process to continue monitoring the successor until the total test duration is completed. Additionally, it needs to log the partial test data up to that point. In the final function instance, the cumulative results of the test are calculated and stored as the result of the activation.

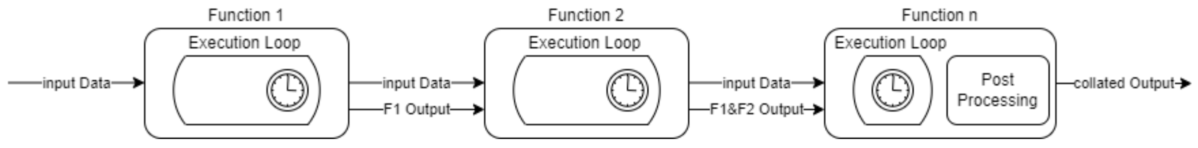


Figure 28: Function Chain Pattern Logic

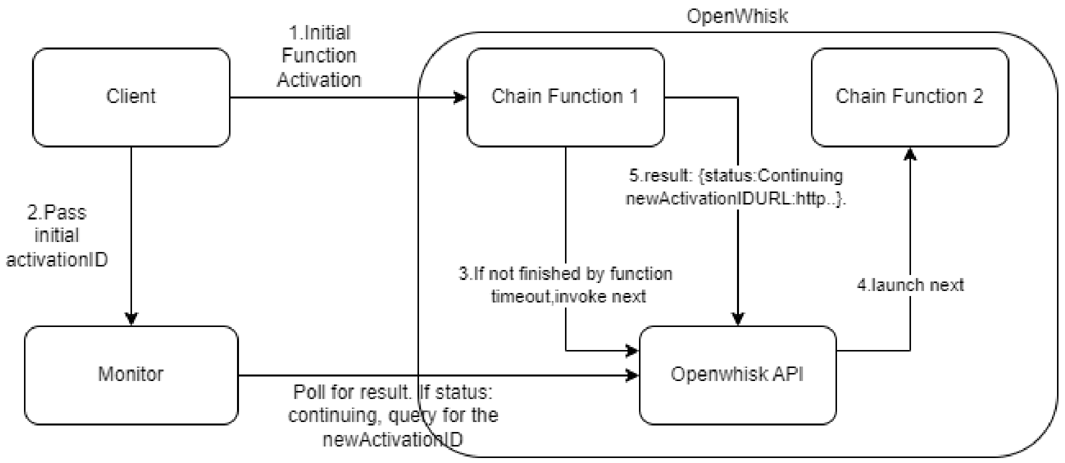

OpenWhisk
Client
1.Initial Function Activation
Chain Function 1
Chain Function 2
2.Pass initial activationID
5.result: {status:Continuing newActivationIDURL:http..}.
3.If not finished by function timeout,invoke next
4.launch next
Monitor
Openwhisk API
Poll for result. If status: continuing, query for the newActivationID

Figure 29: Load Generator Function Chain Inner Structure

PEF FaaS LoadGen Adapter

A top level coordination flow (Figure 30) is also needed (Set Rate Adapter), in order to regulate the overall process (i.e. trigger the messages that trigger the flow generation, gather the results and store them for later analysis) in the context of PEF. This is an alternative way of launching a test, managed by PEF, complementary to the usage of the load generator either as a Node-RED subflow in any Node-RED environment, or directly through invoking the according to OpenWhisk function.

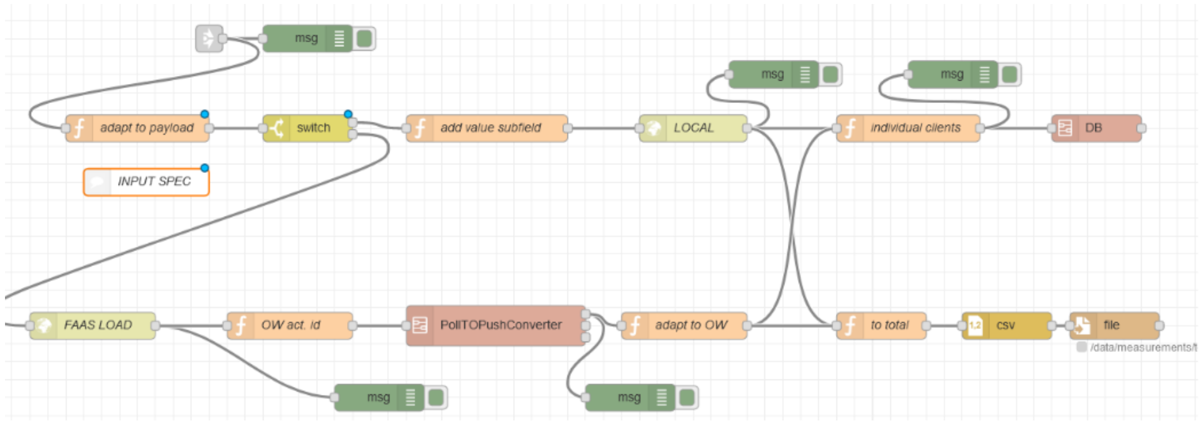


Figure 30: PEF FaaS Load Generator Adapter Flow

3.2.4.2. Clustering function

The clustering function is based on the implementation of the well known k-means clustering method (more details on this can be found in D5.2). In this deliverable, we describe the way the function was implemented. The implementation is based on the Node-RED flow that appears in Figure 31.

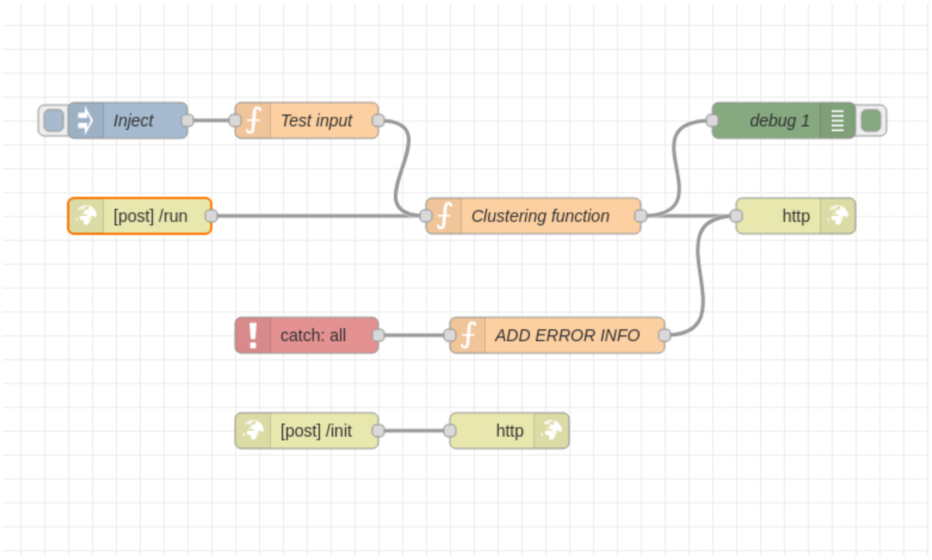
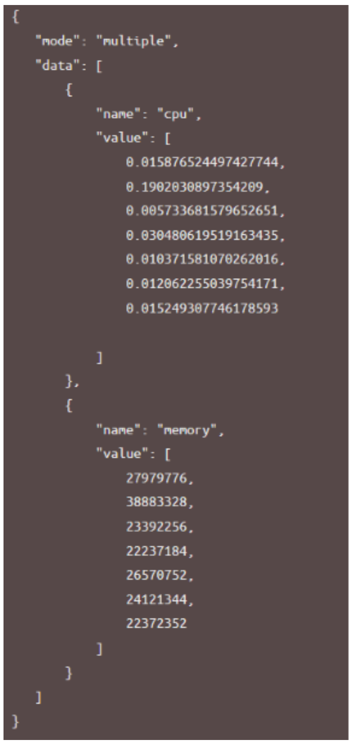


Figure 31: Node-RED flow

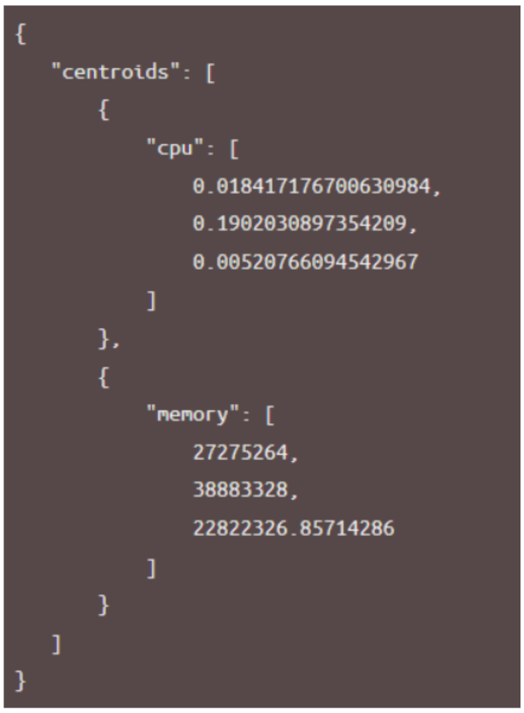
Clustering Flow/Function implementation in Node-RED

This wraps the k-means clustering operation as a service. As such it can be executed inside any Node-RED environment in a service manner. It also implements the OpenWhisk API specification so that it can be executed directly as a custom docker action of OpenWhisk. The inputs include arrays of objects and their values and the output returns clusters with three centroids for any given input, using the k-means implementation provided by the clusters npm library. Specific attention has been given to its interface so that all needed information is fed into the input. This means that clustering deployment and usage are entirely decoupled from any supporting framework, DB and service, simplifying its deployment. It does not need to be configured, for example, with the location or credentials of the REST API that includes the traces since all raw data can be an input argument. Since it is used in this manner, it is easy in the future to be replaced by any other desired machine learning method. The input and output specifications are presented in Figure 32.



```
{
    "mode": "multiple",
    "data": [
        {
            "name": "cpu",
            "value": [
                0.015876524497427744,
                0.1902030897354209,
                0.005733681579652651,
                0.030480619519163435,
                0.010371581070262016,
                0.012062255039754171,
                0.015249307746178593


            ]
        },
        {
            "name": "memory",
            "value": [
                27979776,
                38883328,
                23392256,
                22237184,
                26570752,
                24121344,
                22372352
            ]
        }
    ]
}
```



```
{
    "centroids": [
        {
            "cpu": [
                0.018417176700630984,
                0.1902030897354209,
                0.00520766094542967
            ]
        },
        {
            "memory": [
                27275264,
                38883328,
                22822326.85714286
            ]
        }
    ]
}
```

Figure 32: Input (Left) and Output (Right) JSON specification for the Clustering Function

In the input JSON, the user indicates an array of “name” variables for the various features upon performing the clustering upon. There is no limit for the amount of feature objects the data array can have. The “value” field for each feature indicates the available data points. Lastly, the “mode” option has two options: “multiple” and “single”. The “multiple” option will create separate clusters for each feature object entry (e.g. “cpu”, “memory”) and the “single” option will generate a single cluster that includes all features in the same clustering process (i.e. the clustering vector will include columns from both the “cpu” and “memory” objects). The primary use in PHYSICS is in the multiple mode, in order to have more discrete profiling information as mentioned in D5.2.

Clustering Function Triggering Cron Job

The creation of the clusters is performed by the PEF in a cron job manner with the aid of the flow presented in Figure 33. The flow is responsible for querying the performance DB in order to collect all available profile data up to this point in time. Then it calls the deployed Clustering Function in order to calculate the centroids and finally it stores the result in the performance DB.

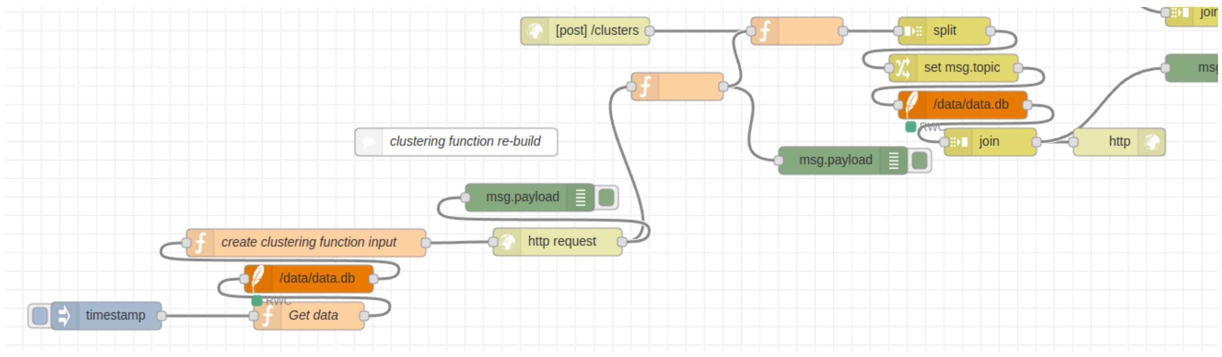


Figure 33: Profiling Clusters Creation Cron Job

3.2.4.3. Classification Flow/Function implementation in Node-RED

The classification flow/function is again implemented in Node-RED through the following flow in Figure 34.

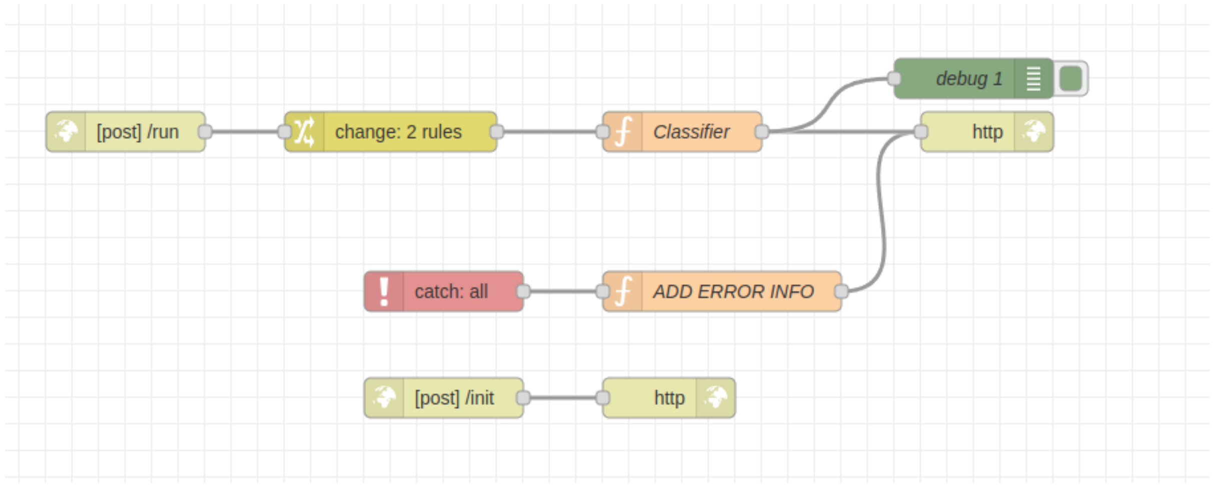
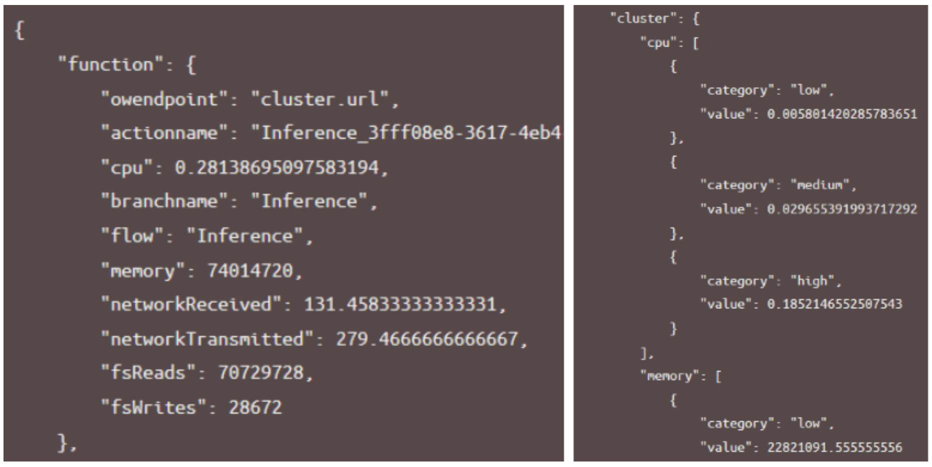


Figure 34: Classification Node-RED flow

This algorithm applies Euclidean distance and compares the distance between the three centroids given from the clustering process (Figure 35b) and the target function profile traces (Figure 35a) obtained during its execution. Both of these fields are provided in the input JSON, so that this flow can be easily replaced without dependencies from other classification algorithms. Like the clustering flow mentioned above, it's possible to run this flow either as a node red flow service or as an OpenWhisk function.

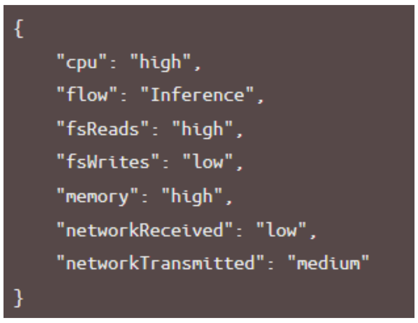


```
{
    "function": {
        "owendpoint": "cluster.url",
        "actionname": "Inference_3fff08e8-3617-4eb4
        "cpu": 0.28138695097583194,
        "branchname": "Inference",
        "flow": "Inference",
        "memory": 74014720,
        "networkReceived": 131.45833333333331,
        "networkTransmitted": 279.4666666666667,
        "fsReads": 70729728,
        "fsWrites": 28672
    },
```

```
    "cluster": {
        "cpu": [
            {
                "category": "low",
                "value": 0.005801420285783651
            },
            {
                "category": "medium",
                "value": 0.029655391993717292
            },
            {
                "category": "high",
                "value": 0.1852146552507543
            }
        ],
        "memory": [
            {
                "category": "low",
                "value": 22821091.555555556
```

Figure 35: Example inputs to the Classification function a) profile trace for given function b) current clustering centroids per resource

An example output of this function is depicted in Figure 36.



```
{
    "cpu": "high",
    "flow": "Inference",
    "fsReads": "high",
    "fsWrites": "low",
    "memory": "high",
    "networkReceived": "low",
    "networkTransmitted": "medium"
}
```

Figure 36: Example output of the Classifier Function

3.2.5 Specific workload parameters and experiments consideration

3.2.5.1. Specific blocking client design for serverless systems

Measurement experiments need always to be handled with specific care in order to ensure result validity and elimination of factors that may influence the results. While this is a challenge for all types of systems, in FaaS platforms it becomes even more dangerous due to a number of characteristics of the latter and especially the fact that the particular systems tend to hide several details around their management and execution models to abstract these from the user. This however creates an extra set of challenges for the system and performance engineer to detect the type of parameters that may affect the performance. In the following paragraphs, many issues are highlighted, to alert the performance engineer before an experiment execution, as well as handle a number of them in the context of the PEF (e.g., through designing specific workload structures to mitigate them, including them in future performance models, etc.).

Cold/Warm/Hot Executions

Due to typical resource management practices in FaaS platforms, such as proactive launching of containers (pre-warm) for various runtimes, an invoked function can be executed in three different manners latency-wise, also depending on which container is used to execute the function. Furthermore, used containers linger in the system in a paused status for a while after the function has executed, typically for around 10-12 minutes, to be available for subsequent function invocations. The variations in the execution of functions are therefore as follows (Figure 37):

- Cold start: No pre-warm container is available for this runtime, container needs to be created from scratch
- Warm start: Pre-warm Container is available for this runtime, function needs to be loaded from the platform DB
- Hot start: Paused Container is available for this runtime and function has already been loaded in this container instance

In the literature, other terms are also used, for example, the categorization may be cold/pre-warm/warm, but in effect, it refers to the same conditions of execution.

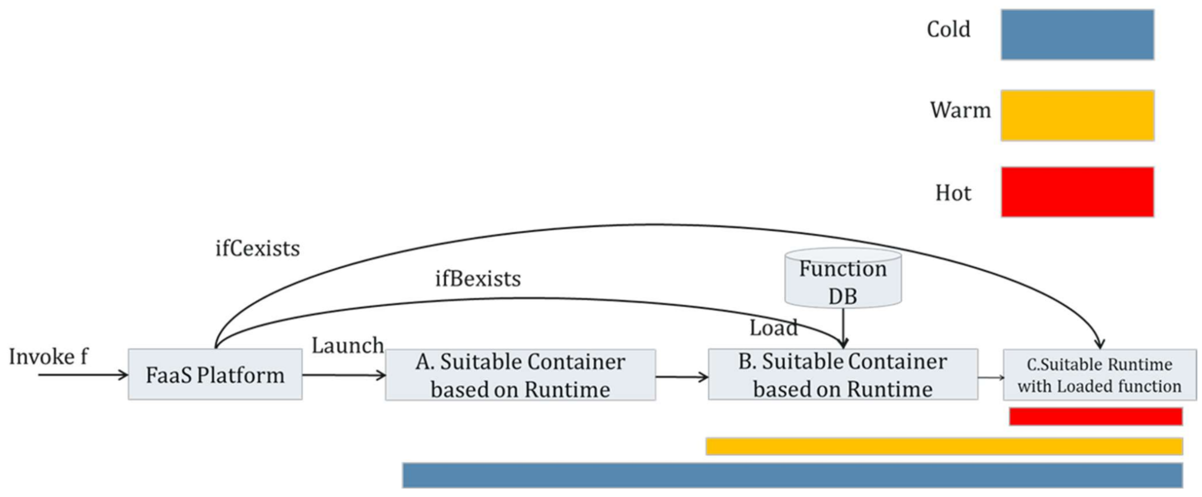
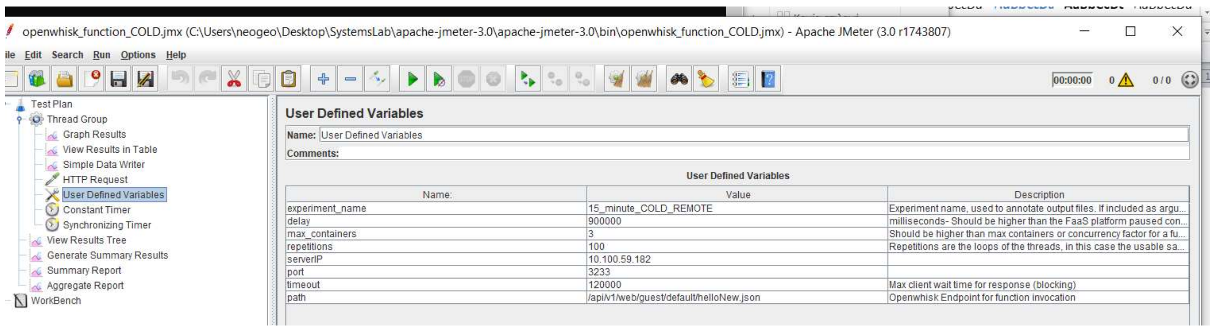

Cold
Warm
Hot
ifCexists
ifBexists
Function DB
Load
Invoke f
FaaS Platform
Launch
A. Suitable Container based on Runtime
B. Suitable Container based on Runtime
C.Suitable Runtime with Loaded function

Figure 37: Cold/Warm/Hot containers variations of executions

To ensure easy repeatability and adaptation of the client, three types of clients have been created by Jmeter that help ensure the validity of the measurement process. To enable usability and easy configuration, the overall setup is performed following user-defined variables. Each parameter is defined in a central location (User Defined Variables tab) and its use throughout the Jmeter script is based on referencing the according variable (Figure 38).



User Defined Variables

Name: User Defined Variables

Comments:

User Defined Variables

Name:	Value	Description
experiment_name	15_minute_COLD_REMOTE	Experiment name, used to annotate output files. If included as argu...
delay	900000	milliseconds- Should be higher than the FaaS platform paused con...
max_containers	3	Should be higher than max containers or concurrency factor for a fu...
repetitions	100	Repetitions are the loops of the threads, in this case the usable sa...
serverIP	10.100.59.182	
port	3233	
timeout	120000	Max client wait time for response (blocking)
path	/api/v1/web/guest/default/helloNew.json	Openwhisk Endpoint for function invocation

Figure 38: Jmeter user interface

The parameters to be defined in the user interface for the workload are:

- Experiment name: used to annotate output files and results
- Delay (milliseconds): should be higher than the FaaS platform paused hot container duration
- Max_containers (used for cold cases): should be <= the max containers or concurrency factor for a function. Used to dictate the max number of client threads
- Prewarm containers (for warm and hot cases): number of prewarm containers, used to dictate the number of user threads.
- Repetitions: loops of the threads, however for each different setup the number of usable samples has a respective rule (to be described in the following sections) serverIP and port: details of the respective API endpoint of the FaaS platform for function invocation
- Timeout: max client wait time for response (blocking call)
- Path: path of the function to invoke (the full path is defined by <serverIP>:<port>/<path>

In each case, the strategy followed by the respective platform, as well as the pause duration for the container, need to be taken into consideration to ensure that the measurements are accurate and no overlap between calls or scheduling of other types of container modes infiltrates the process. For this reason, the following measurement methodology has been followed to avoid potential caveats in sample retrieval

Cold cases

The cold cases are the most difficult to obtain. In this case, by cold we mean executions where the container image layers are present on the node, but the relevant container needs to be created from scratch. Initially, we need to launch bursts of concurrent requests so that they do not find warm or hot containers. Looping around the bursts can give the number of data points one needs. However, when launching concurrent threads, it was observed that requests had time to find hot containers. This was due to the fact that the overall number of containers for a given function may be throttled based on the configuration of the platform. Thus, when the max+1 request arrives, it does not spin a new container, but it waits until a previous one has been released. This can be identified if one checks the number of paused containers after each burst, which should be n-1 to account for the pre-warm container, where n the size of the burst. Furthermore, due to differences in request arrival and potential processing time in the first loops (mixed warm and cold starts), the later requests in the burst found containers that had finished processing the first arriving requests (Figures 39-40). In order to cope with that, a synchronizing timer needs to be applied at the end of each loop, so that all threads kickstart at the same time in the next loop (Figure 41). Another point of attention is the need to use a constant timer of over 12 minutes since this is the time limit the platform maintains the paused hot containers.

Finally, the k smallest response times from each burst need to be removed at the end, where k is the number of prewarm containers used since this refers to the warm container usage. Alternatively, the pre-warm containers can also be deactivated before the measurement. However, this is not always possible, e.g. in an online platform, it means that we need to stop OpenWhisk, change the settings and relaunch. Therefore, this option is not available for testing production deployments that cannot be stopped.

In this case, the usable samples are *{repetitions}*({max_containers}-prewarmcontainers)*.

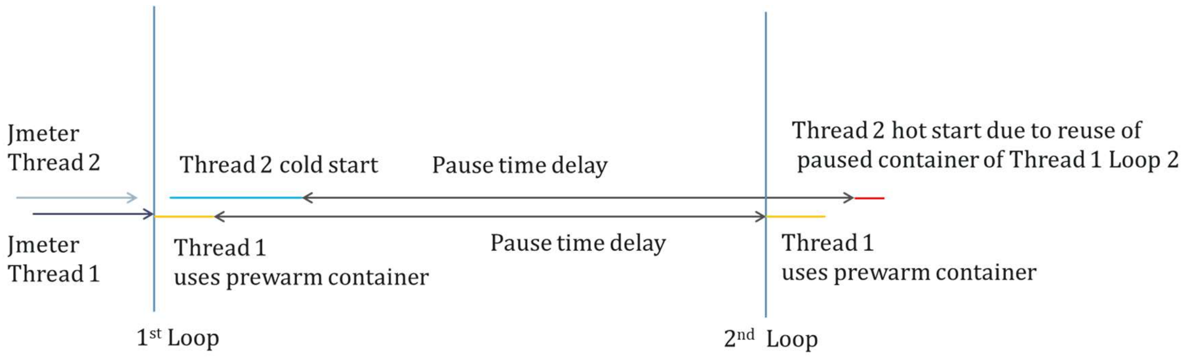

Jmeter Thread 2
Thread 2 cold start
Pause time delay
Thread 2 hot start due to reuse of paused container of Thread 1 Loop 2
Jmeter Thread 1
Thread 1 uses prewarm container
Pause time delay
Thread 1 uses prewarm container
1st Loop
2nd Loop

Figure 39: Mixed executions that can defect the measurement process and mitigation without the usage of a synchronizing barrier at the end of each loop

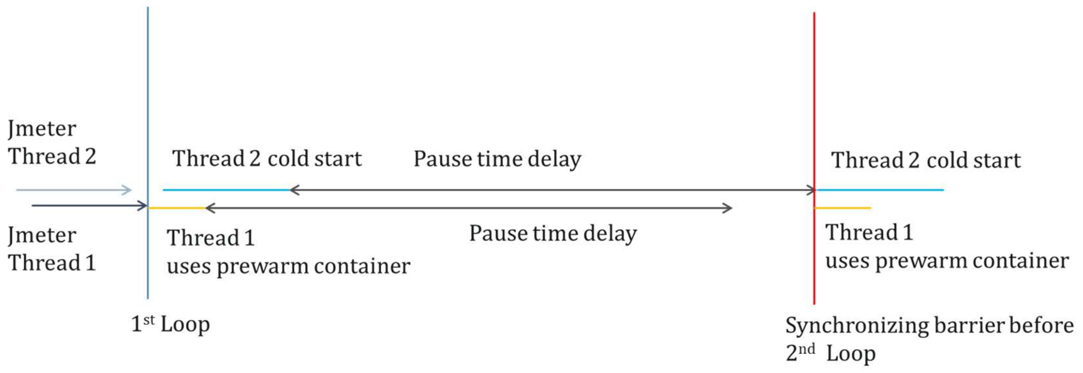
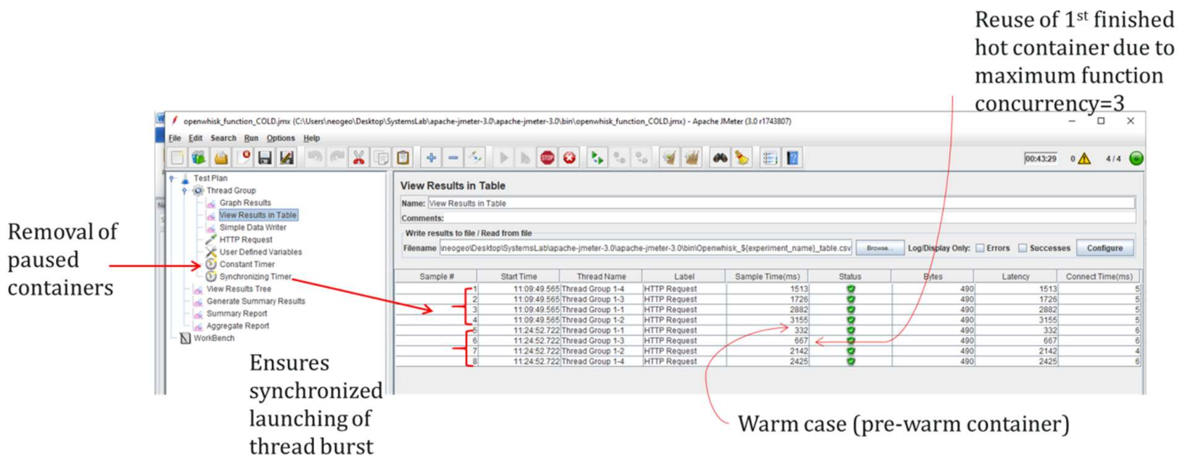


Figure 40: Potential mixed executions that can defect the measurement process and mitigation through the usage of a synchronizing barrier at the end of each loop


Reuse of 1st finished hot container due to maximum function concurrency=3
Removal of paused containers
Ensures synchronized launching of thread burst
Warm case (pre-warm container)
openwhisk_function_COLD.jmx (C:\Users\neogeo\Desktop\SystemsLab\apache-jmeter-3.0\apache-jmeter-3.0\bin\openwhisk_function_COLD.jmx) - Apache JMeter (3.0 r1743807)
File Edit Search Run Options Help
00:43:29 0 4/4
Test Plan
Thread Group
Graph Results
View Results in Table
Simple Data Writer
HTTP Request
User Defined Variables
Constant Timer
Synchronizing Timer
View Results Tree
Generate Summary Results
Summary Report
Aggregate Report
WorkBench
Browse...
Configure

View Results in Table

Name: View Results in Table

Comments:

Write results to file / Read from file

Filename \neogeo\Desktop\SystemsLab\apache-jmeter-3.0\apache-jmeter-3.0\bin\Openwhisk_${experiment_name}_table.csv

Log/Display Only: ☐ Errors ☐ Successes

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Latency	Connect Time(ms)
1	11:09:49.565	Thread Group 1-4	HTTP Request	1513	✓	490	1513	5
2	11:09:49.565	Thread Group 1-3	HTTP Request	1726	✓	490	1726	5
3	11:09:49.565	Thread Group 1-1	HTTP Request	2882	✓	490	2882	5
4	11:09:49.565	Thread Group 1-2	HTTP Request	3155	✓	490	3155	5
5	11:24:52.722	Thread Group 1-1	HTTP Request	332	✓	490	332	6
6	11:24:52.722	Thread Group 1-3	HTTP Request	667	✓	490	667	6
7	11:24:52.722	Thread Group 1-2	HTTP Request	2142	✓	490	2142	4
8	11:24:52.722	Thread Group 1-4	HTTP Request	2425	✓	490	2425	6

Figure 41: Example of synchronization solving, and problem inserted by maximum allowed containers

Warm cases

In the warm case, the number of requests per measurement loop needs to be the equal to the number of pre-warm containers set for a given runtime in the platform. This is to ensure that no cold starts will infiltrate the measurements if requests in a batch are higher than the available pre-warm containers. For safety, one can use a single user thread (assuming that the platform has at least one instance of a pre-warm container for the specific runtime) in order to avoid synchronization issues between minor thread delays from requests in the same burst (e.g., due to random network delays) that may lead to finding hot containers from just previously executed functions (as described in the cold case section). Alternatively, a synchronizing timer can be used, as in the cold case scenario. Furthermore, between each measurement loop, the client needs to wait for the interval in which used containers are in the pause mode (e.g. 10-12 minutes). This will ensure their elimination before the next request loop arrives, eliminating any hot executions infiltrating the warm case.

Therefore, the only difference from the previous case is the different delay user defined variable in the workload file (900000 milliseconds or 15 minutes) and a higher duration of experiment execution to retrieve the required data points. Usable samples in this case are {repetitions}*{prewarm_containers}.

Hot cases

The hot case is easier to measure, given that each execution can be fired immediately after the previous one has finished. Therefore, 1 client thread is needed that loops across n times of execution depending on the needed data points. Given that the container is continuously used, almost all cases of execution will fall in the hot category, apart from the first that will be in the warm state. The only point of attention in this case is not to send a new request before the previous one has finished, which would lead to a new cold container being created. Thus, the delay between calls should be set to a value that is expected to be higher than the function execution time. Even this delay can be avoided given that Jmeter threads are blocking threads, meaning that the n+1 request of the user thread is not launched until the response to the nth request is obtained.

Usable samples in this case are {repetitions}*{prewarm_containers} -{prewarm_containers}. The first {prewarm_containers} samples need to be removed since these relate to warm cases. For better result validity, in order to avoid interference from multiple concurrent containers, it would be advised to set the number of prewarm containers to 1, if one wants to measure the baseline times without any overhead inclusion.

An indicative usage of the three loads is given in Section 3.3.2.

3.2.5.2. Serverless System Parameters for consideration

Another set of parameters that may affect the measurements can be configured in the OpenWhisk environment. The most important of these options are described below. These options are ones that can affect an experimentation process.

Global and local limits per action invocation

The OpenWhisk environment has several high-level limits applied to each function, indicated in the config file options below.

Code 4: Configuration file

```
config {
    controller-instances = 1
    limits-actions-sequence-maxLength = 50
    limits-triggers-fires-perMinute = 60
    limits-actions-invokes-perMinute = 60
    limits-actions-invokes-concurrent = 30
  }
```

These options regulate for example the maximum invocation of functions per minute (60) as well as the maximum number of concurrent containers running (30). Thus, if an experiment targets more than these limits it will not be able to obtain them unless a different setting is applied.

A few other local limits (per function) exist, primarily the memory and timeout interval for an action invocation. More information about the OpenWhisk limits can be found here[3].

Number of containers limited by container pool memory

One significant aspect that regulates how many containers can be raised by OpenWhisk is the memory available for the container pool. Even though the same parameter (max number of containers) is regulated by the global concurrent action invocation limit, there is another cut-off point that is based on the available memory assigned to the container pool. This is an option again regulated at the application.conf file, as the option excerpt below demonstrates (Code 5). Based on this, the max number of containers is defined by the memory size of the container pool, divided by the memory needed by each container we need to raise (which in turn depends on the function memory size required).

Code 5: Available memory

```
container-pool {
    user-memory:8192 m
    ...
  }
```

It needs to be stressed that a launched OpenWhisk instance can include a baseline conf file that is overridden by an updated one in parts of the options. In the updated one, the keyword "include" at the beginning of the extra application.conf file can be used to import default parameters from the baseline version, whereas only the options to be overridden may be included in the extra conf file and passed to the OpenWhisk startup via the -c option.

Number of prewarm containers

The number of prewarm containers launched by the OpenWhisk environment is configured per type of runtime registered with the environment. There are two ways of configuring this number, either statically or dynamically based on the observed frequency of requests. This configuration takes place in the runtimes.json file of the OpenWhisk setup[4]. The configuration is based on a JSON construct named stemCells, appended in each runtime description. A static configuration of 2 pre-warm containers appears in the following code example (Code 6), assuming the maximum number of containers is not cut off by the container pool memory mentioned in the previous paragraph.

Code 6: Configuration

```
"stemCells": [
     {
        "initialCount": 2,
        "memory": "256 MB"
     }
     ]
```

A dynamic configuration can be set through the following stemCells structure:

Code 7: Dynamic configuration

```
"stemCells": [
     {
        "initialCount": 2,
         "memory": "256 MB",
         "reactive": {
             "minCount": 1,
             "maxCount": 4,
             "ttl": "2 minutes",
             "threshold": 2,
             "increment": 1
     }
     ]
```

In this case, if some containers experience a cold start, then the number of pre-warm containers is given by the formula:

# of prewarm containers to be created = # of cold starts/threshold * increment

limited by the min and max count parameters. The created containers will have a TTL defined by the respective parameter. if the traffic reduces, then the number of containers will eventually converge to the min count parameter.

This information is critical for example when one uses the cold/warm/hot workload files, described in the beginning of this section, since as it was described, the accurate operation of these depends on the setting of the number of pre-warm containers. Therefore, in such an experimentation phase, one needs to ensure that the number of pre-warm containers follows a static configuration, as well as their created number is not constrained by the other parameter that may influence the container launching (container pool memory size).

3.2.6 Data Model

For the main functions presented above, the relevant inputs and outputs of them have already been described. In this section, we describe the overall entities that exist:

Table 7: PEF component main entities data model

Entity	**Description**	**Component or service owner**
Load Test Data	A defined load test performed by the user. Includes information on the timestamp the test was created, the flow name from which the function originates, the branch from which the function originates, the location where the function was executed and the output of the load test (already described in the Measurements Analysis section)	PEF
Trace Data	The entity includes information on an acquired function trace. As such it includes fields for the Activation Id (of the traced function), the function input payload, the function name (and the flow from which this function originated), the OpenWhisk location of execution, the branch name that owns the flow and a field for each of the resources in the trace (cpu, memory, network in/out, filesystem read/write)	PEF
Resource	One of the available resources for which information comes in the trace data (cpu, memory, network in/out, filesystem read/write)	PEF
Profile Data	The resulting classification for a given function. It includes an auto-increment id, the function, flow and branch names as well as the classified category (one of low/medium/high) for each resource (cpu, memory, network in/out, filesystem read/write)	PEF
Resource Cluster Centroids	This includes the data from the run clustering processes. Each entry includes an auto-increment id plus the finally created centroids (low/medium/high) per resource (cpu, memory, network in/out, filesystem read/write) for this run	PEF
Model	A model created from the available results of a test series.	

3.3. Demonstration or Experimentation outcomes

3.3.1 Batch Request Aggregator Performance Model

ANN model accuracy

In D3.1, an extensive experimentation process was performed in static configurations to detect how the Request Aggregator pattern affects the system in cases of low and high traffic and how it affects the response time based on the set batch size. The results from these runs were used to train the ANN model predictor, described in the previous sections, for getting the expected average response delay (output) for different frequencies (input 1) and different batch sizes applied (input 2). ANNs were chosen since they represent black-box universal approximators and can be applied based on an available dataset, without further knowledge of the internals of the system. In contrast, other approaches such as PID controllers require a more tailored approach to determine the relevant metrics of the system as well as a controlled entity, and a calibration step.

The ANN model structure was optimized based on a genetic algorithm to determine the main model structural characteristics (number of layers, type of neurons per layer and number of neurons for hidden layers)[15] , which is also the main algorithm implemented as a model creator in PEF. The network type is based on back-propagation, and feed forward architecture and the resulting model was selected from a pool of approximately 450 candidate networks that portrayed an intermediate validation error of less than 20%, based on its performance on an intermediate test set. The finally selected network consisted of 5 layers (3 hidden). The first four were configured with the tansig transfer function while the output layer with the purelin function of Octave.

After the selection of the best model, a further acceptance test was performed to simulate runtime usage. From the available 30k experimental values, ~7k of them were reserved for this purpose. These values were not utilized in whatever manner during the training or optimization process of the model design. The Mean Absolute Percentage Error of 11.73% was achieved (Figure 42). The surface plot of the model appears in Figure 43, in the model normalized range.

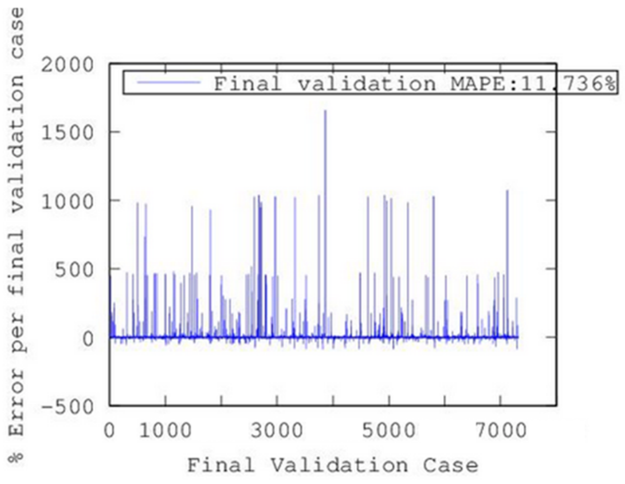


Figure 42: Model Error in final validation cases

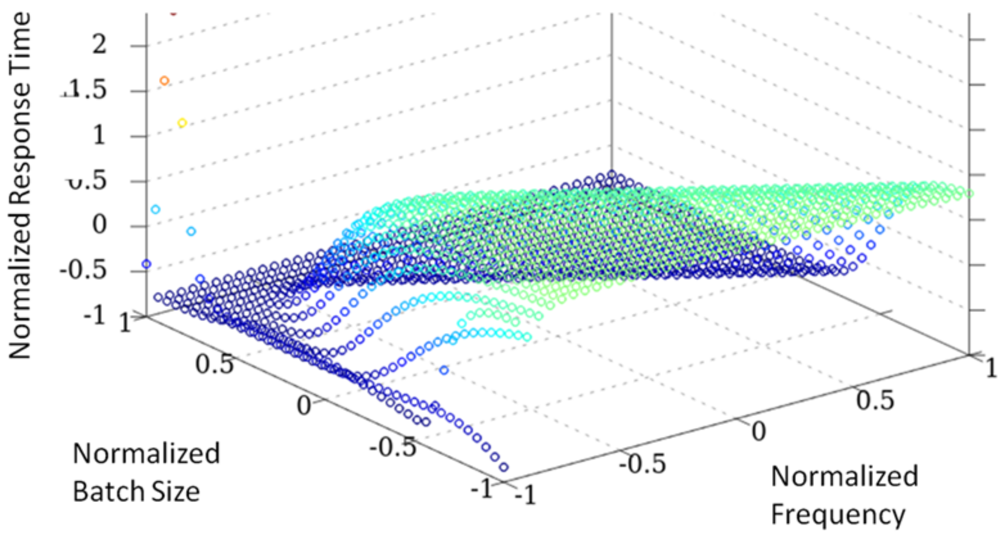


Figure 43: Surface Plot of Model Predictor in the normalized range (Input1: Frequency, Input2: Batch Size, Output: Response Time)

Dynamic Adaptation based on the ANN model

To test the model's response during runtime as well as the PEF's ability to respond in a service-oriented manner, and especially in bursts of load, an experiment was performed with varying loads including sudden peaks (Figure 44). This was performed to also observe the effect of the delay in getting the prediction response and how the system gets affected by it in actual conditions, despite the good accuracy of the model in the static validation. To decouple this delay from the conditions in the system, the pattern adaptation model was running on a separate node to ensure a timely reaction to sudden peaks of load.

The model was queried at a regular interval (every 10 seconds) based on the current conditions of execution (frequency). Given that the model takes approximately 10 seconds to respond, this results in a decision update for the batch size every 20 seconds. Predictions were obtained for different batch sizes (ranging from 1 to 70 with a step of 10). The one achieving the lowest predicted response time was selected and used to configure the pattern. Given that there is a direct relation between frequency and anticipated batch completion, one can also utilize this information to configure the timeout interval.

The results are portrayed in Figure 45 from the client side (for each individual request sample) and Figure 46, Figure 47 and Figure 48 from inside the system, by taking samples every 1 second for processed frequency, active container numbers and batch size (set and measured). From the figures it can be seen that when the increase is gradual (e.g. from 0.2 messages/sec to 4) the system has the time to adapt. Although there is an initial increase in response times, this is fixed by the increased batch size. A further increase from 4 to 8 messages per second (around sample 2000 of Figure 45) does not influence the system, given that the batch size is already high. In this period, we also observe a further decrease in response times, as would be anticipated by the average times reported in D3.1 for the static batch size measurements.

When the load peak is more sudden (from 0.2 to 20 messages/sec), the delay in adaptation leads to a very high response time (in essence reaching the 120000 milliseconds timeout) due to the high number of accumulated messages and according to containers. However, one major benefit of the pattern in this case is the fact that the system does not stall, it only experiences a temporary denial of service. This could be handled either by larger timeout times or by applying in combination with this pattern the Circuit Breaker pattern [27] to refrain the client from generating more requests temporarily.

Another interesting conclusion is that there are sudden peaks in container numbers in the transition time from high to low frequency, not justified by the amount of traffic at that time (tailing of container numbers in Figure 47 following the drop of the peak around sample 400. This indicates that there are many lingering requests, already measured in the frequency, in Node.js queues of the PEF implementation, that start to request resources. However, when the frequency is detected as reduced, the system returns to the 1 batch size, making these lingering requests raise one container each. This is an indication that a potentially improved predictor for the ANN would be the number of active containers and not the request frequency. It is also an indication that in other types of adapters (e.g. PID controllers), the past values element (i.e. Integral factor) should be strengthened to achieve a graceful, gradual reduction in batch set sizes.

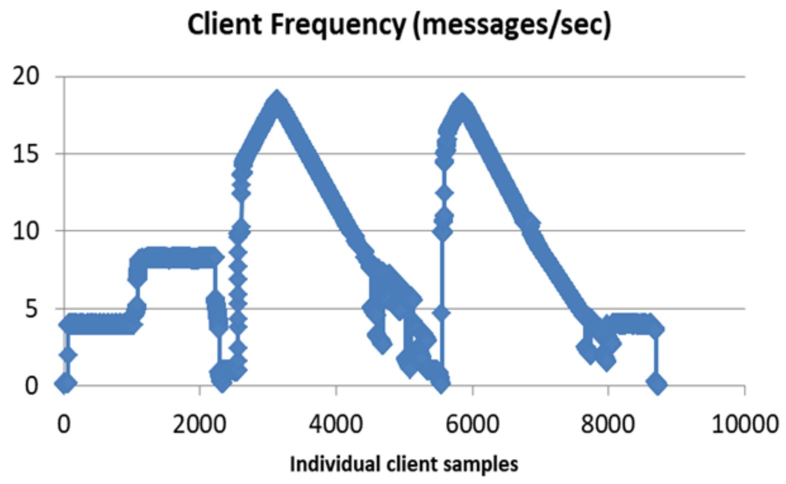


Figure 44: Client side request generation frequency measured per sample

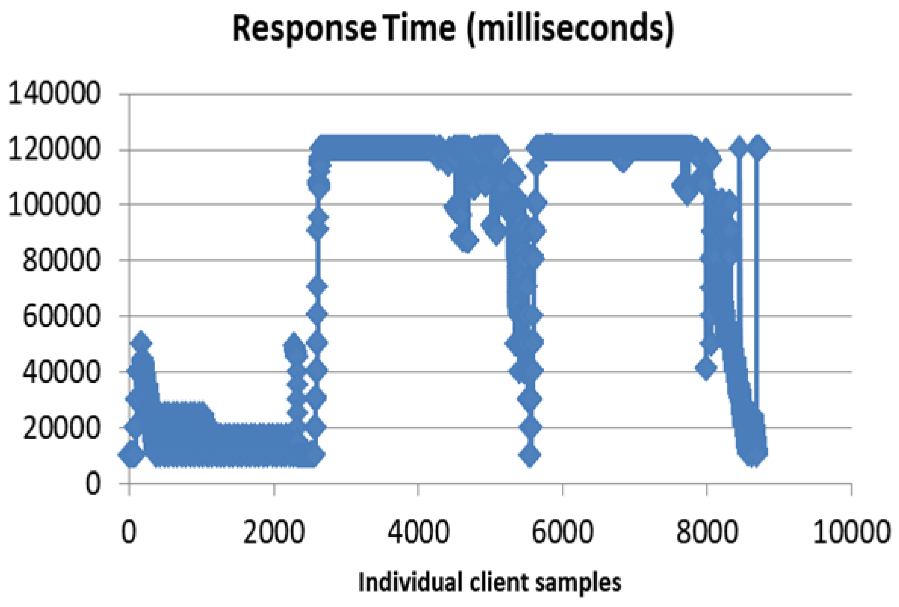


Figure 45: Client side Response Times per sample

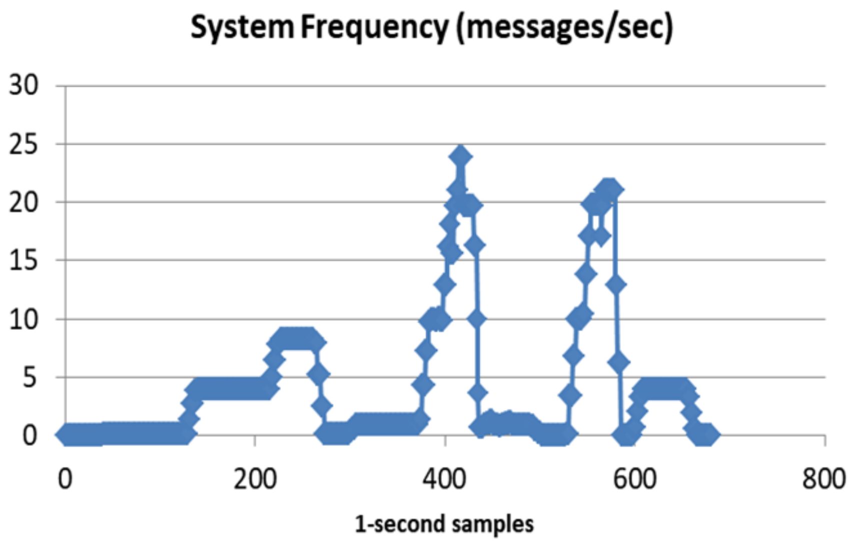


Figure 46: System side measured frequency per 1 second interval

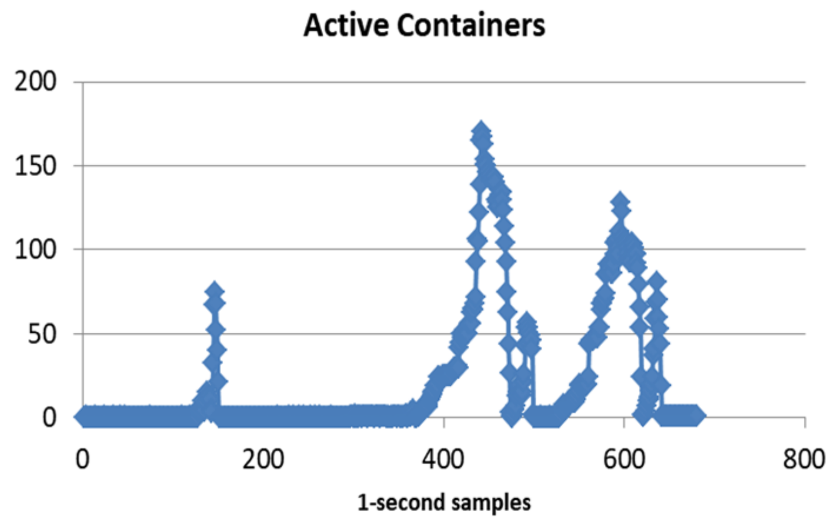


Figure 47: Active (running) containers in the system

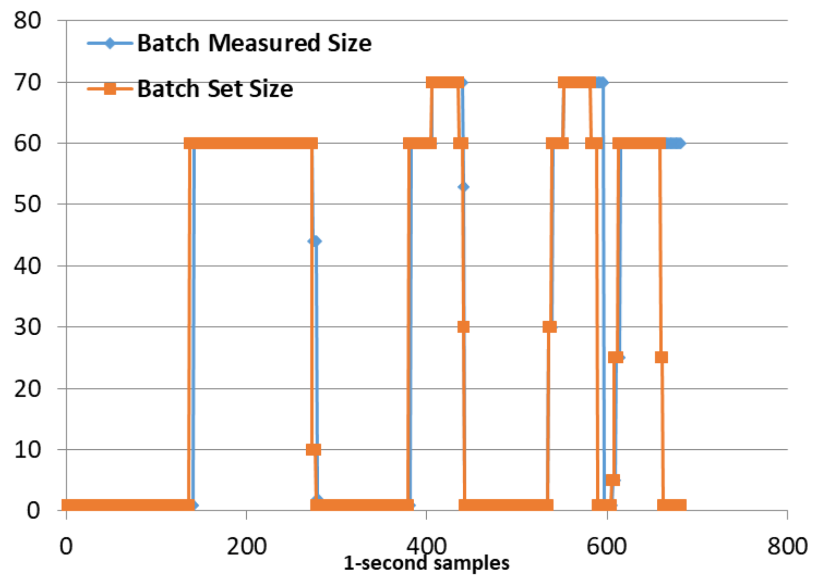


Figure 48: Batch Size Setting by ANN model and according to system side application of the setting

FaaS Cost Models association with Request Aggregation Pattern

FaaS cost models charge customers based on performance metrics such as execution time, memory or number of invocations [28]. This implies that there is a combined need as well as the incentive to tackle the performance-cost tradeoff, especially for runtime management [29] [30]. The Request Aggregator pattern reduces the number of invocations, as well as the execution time (given the lower need for environment initialization), and this is expected to aid in minimizing cost aspects. Other works that can be combined include the investigation of trade-offs after which a switch to serverless mode is beneficial for the customer [31]. Predicting accurately costs [32] can also be used in combination with this approach. This can be helpful in cases where one needs to regulate/minimize cost directly and not for example based on frequency, if performance is considered less important. In this case, a series of measurements were performed with the use of the Request Aggregator to investigate what is the overall execution time for a prediction request with multiple input lines, compared to running the model for only one input row for estimation.

The performed requests include an interval of 15 seconds between them, to investigate the baseline execution time when only one container is active in the system. Input size variations included 1, 100 and 450 input rows for which predictions were needed, simulating a relevant batch size applied by the RA pattern. Calls to the service were performed for a duration of 10 minutes per case. The results appear in the following table and are associated with the estimated costs. The estimation is based on several assumptions. All calls are considered as cold-start ones and also include the time to raise the container (which is not billed in actual FaaS offerings). However, they are intended as a rough overview of potential benefits. As a baseline, the cost model of AWS Lambda (US East-Ohio) [7] is used. The used memory was allowed to use the overall memory available (10GB). Thus, the price of 0.0000001667 dollars per 1 msec of runtime was used and multiplied by the runtime duration, while the cost per input row also appears in Table 8. This depicts a significant difference, which is also evident when taking under consideration the other billing factor (number of requests set at 0.2$ per 1 million of requests) that is also affected by the reduced overall number of calls needed for one to obtain 1 million predictions, if they apply the RA pattern.

Table 8: Cost Estimation of GB-second based on AWS Lambda pricing model if multiple rows are aggregated following the Request Aggregator pattern

Input Rows	Average Execution Delay (msec)	Estimated GB-Second Cost Per Input Row	Cost of Requests for 1M predictions
1	10051.25	0.001675	0.2
100	10082.62	0.000016807	0.002
450	10228.29	0.000003789	0.000444

3.3.2 Cold vs Warm vs Hot Executions

The goal of this experiment is to highlight and measure performance differences between the three stages of execution in typical FaaS environments, as well as to establish repeatable methodologies for acquiring measurements and testing the defined typical workloads created in the context of PHYSICS. The platform of execution was the OpenWhisk standalone version, running in a VM with 4 cores and 8GB of RAM. The cold/warm/hot typical Jmeter workloads, described earlier, were used against a target hello world node.js function, obtaining the results of Figure 49 and Figure 50.

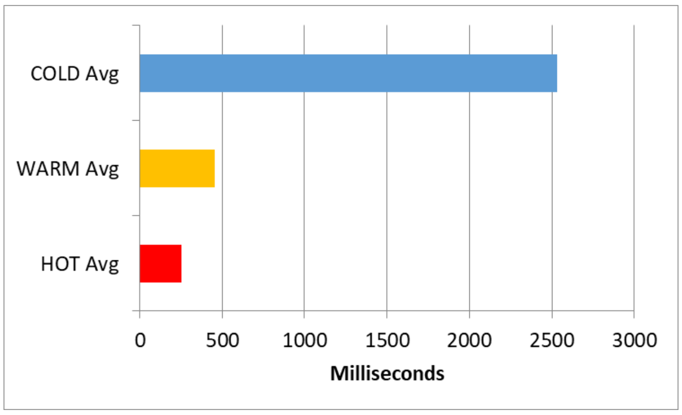


Figure 49: Cold/Warm/Hot Jmeter workloads response time of "Hello World" node.js function execution time

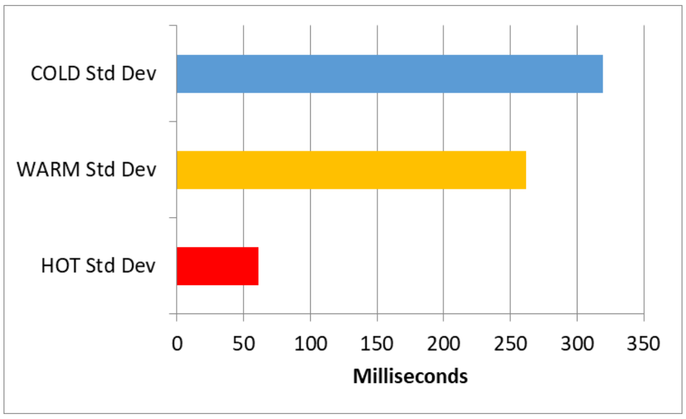


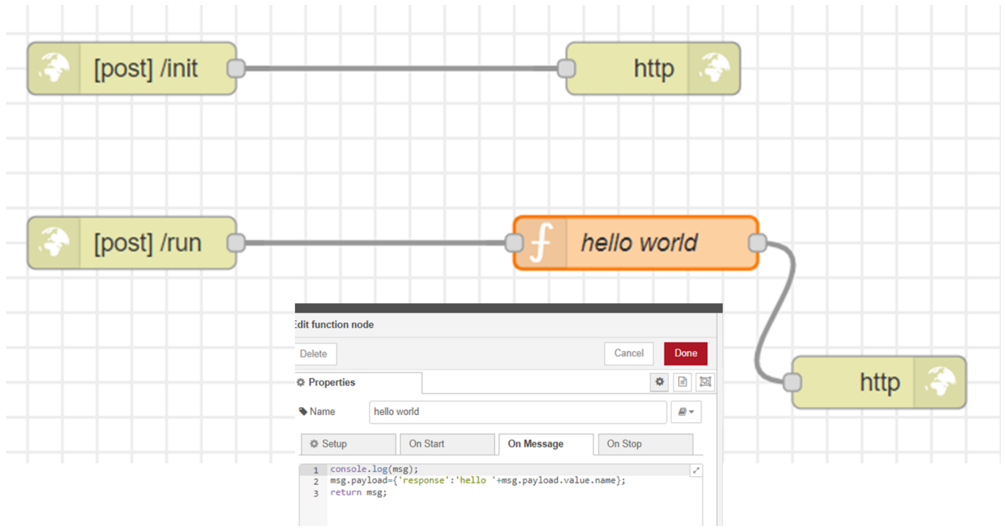
Figure 50: Cold/Warm/Hot Jmeter workloads standard deviation of "Hello World" node.js function execution time

From this it can be seen that a ratio of the three modes is the following:

- Cold/Warm= 5.5x slower
- Cold/Hot=10x slower
- Warm/Hot= 1.8x slower

3.3.3 Node-RED versus Node.js runtimes for Hello world single function application

One of the execution modes as mentioned in D3.1 includes the ability to define a Node-RED flow that will be wrapped and executed as a function in OpenWhisk, either in a native Node.js or a Node-RED runtime (Node-RED flow to function pattern). 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). 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. A relevant testing subflow (Figure 51) has been created with a "Hello world" simple function that includes the two methods if one wants to follow the Node-RED function execution mode. One question is how different in terms of performance the execution of a Node-RED function is, compared to the execution based on the typical, built-in nodeJS runtime. A similar hello world implementation was created and registered for the typical lightweight Node.js runtime of OpenWhisk. The OpenWhisk setup included a single node with 8 cores and 16 GB of RAM, given that we were primarily interested in the baseline times of one function execution (delay in terms of orchestration overheads will be investigated in the next section for larger workflows).



Name: hello world

```
console.log(msg);
msg.payload={'response':'hello '+msg.payload.value.name};
return msg;
```

Figure 51: Node-RED workflow of “Hello World” Node.js function (for Node-RED runtime)

Afterwards, 100 test runs were conducted on each target function, to measure the cold start times (need to spawn a new container) and the hot ones (reuse of an existing container). The results appear in Figure 52.

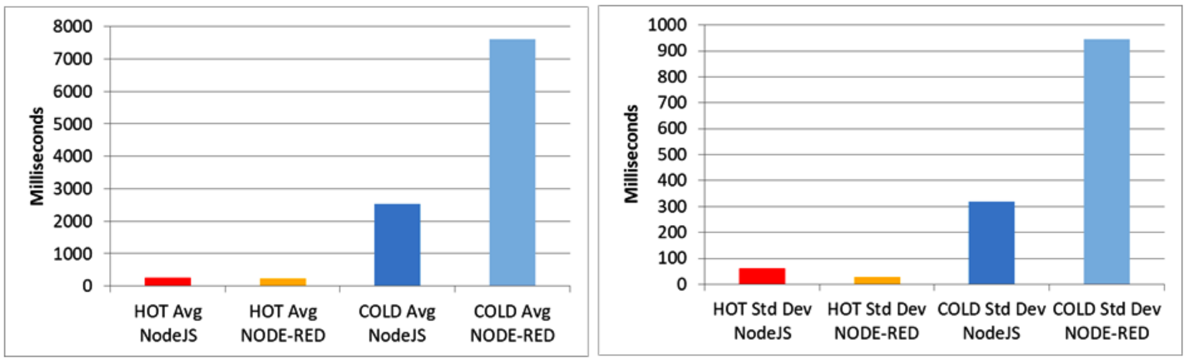


Figure 52: Node-RED execution times of “Hello World” Node.js function (with Node-RED runtime)

As expected, the cold start time for the Node-RED runtime is considerably higher (with an average of 7.6 seconds compared to 2.5 seconds), given that it is based on a larger container with more dependencies, as well as the fact that the Node-RED environment has a more heavyweight start-up time during function execution. In the case of the PHYSICS platform, this is expected to be alleviated also by the container scheduler used in WP5, that chooses nodes based on the existence of respective container layers. In the hot execution case, the two execution modes are very similar, with Node-RED having a slight advantage (227 against 252 milliseconds) as well as a smaller deviation. Addressing the larger cold start times could also be performed by utilizing the Function Warmer pattern [26] or by extending the number of prewarm containers mentioned in Section 3.2. It is necessary to stress that this primarily observes the performance difference between the two modes. A developer might opt for the Node-RED case for other reasons as well (e.g. easier development and function management). However, they need to be aware of the performance trade-off that needs to be made for cold cases. With relation to the usage of the Tabula Rasa context purge pattern defined in D3.1, this has been measured to have an average of 265 milliseconds, thus adding an approximate extra 38 milliseconds of delay compared to the opportunistic Node-RED runtime case.

3.3.4 Node-RED flow orchestration to OW function orchestration delays comparison

The aim of this experiment is to investigate the trade-off between two modes of function sequences (Figure 53). The first mode is Function sequence of nativeOpenWhisk runtime functions: this targets at workflows that are defined as function sequences directly in OpenWhisk (e.g., in the Node.js runtime), utilizing registered (to OpenWhisk) Node.js functions as their main building block. The second mode is Function sequence in Node-RED, executed as a docker action in OpenWhisk. The main difference in this case is that all the function workflow is created in Node-RED and deployed within a custom docker image (invoked as a function) that contains the Node-RED environment. Given that there is a significant delay in the start-up of the Node-RED environment (as measured in the previous section), the purpose is to investigate what (and if there) is the benefit of one mode versus the other compared to the number and duration of the functions that consist of the sequence. The motivation behind this approach is the fact that it is expected that inter-function communication (ifc) times in the sequence might differ between the two cases of execution, which might create a margin of exploitation of one mode over the other. This is reasonable to assume for two reasons:

- In a native OW sequence, the invocation from one function to another needs to pass through the respective OW mechanism and the new action invocation to be queued waiting for execution. On the other hand, in the Node-RED Action Image, the invocation is passed through the Node-RED runtime directly
- The Node-RED case will always be a hot execution (at least for the functions in the sequence after the first one), given that all functions execute in the same container. On the other hand, in the case of multiple and different function executions, the OW runtime may be forced to cold starts for the different function invocations

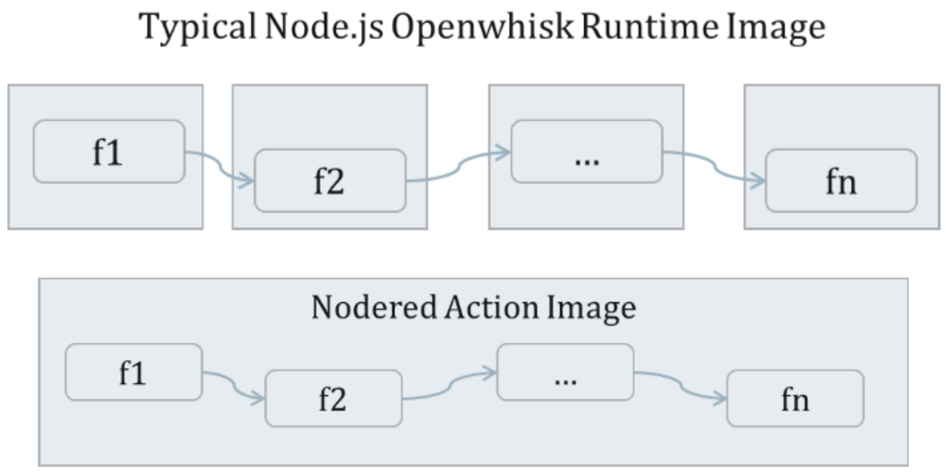
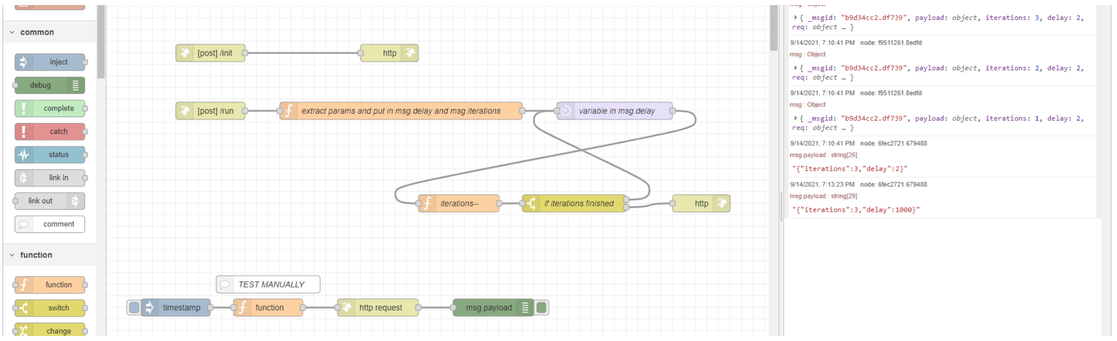


Figure 53: Node.js OpenWhisk function flow execution (many containers) vs. Node-RED (same container)

To implement the experiment, the following actions were performed:

- An artificial delay function has been created as a native Javascript function that can accept as parameter the milliseconds of delay to apply. This function is registered as a Node.js function type in OpenWhisk.
- A similar function flow is created for Node-RED (Figure 54). The flow is adapted in the Node-RED OW Skeleton described in D3.1. The core flow handles a control loop where delay and iteration numbers are dynamically passed on as input variables. The function node in the middle subflow extracts parameters from the incoming message and passes them through the message.delay and message.iterations fields. The follow-up node is a Node-RED built-in delay node that overrides the internal delay of the node with the assigned msg.delay we get from the call on the previous function node. Then there is a loop to check whether the iterations are finished, in order to simulate multiple functions in a sequence. If so then there is an http response, if not then the flow returns to the delay node and then to the iteration check. The third row is a manual timestamp test with a custom input to check if our main delay flow responds correctly and can be used inside the Node-RED environment for testing.
- Sequences of varying numbers of functions (e.g., 1, 5, 10, 15, 20, 25), set up as OpenWhisk function sequences for the Node.js delay function. The name of the sequence is parametric (in terms of the sequence name) so that all types of sequences can be invoked by a parametric Jmeter load client. Actions were exposed as web actions.
- A docker image with the Node-RED delay flow was created and registered as an OpenWhisk Docker Action (again as a web action)
- Jmeter clients that invoke both actions (OpenWhisk sequence with delay input parameter and Node-RED action with delay and delay loops parameters) and compare the results. Loops to implement multiple parameter values are applied in the Jmeter files to measure all the needed combinations.

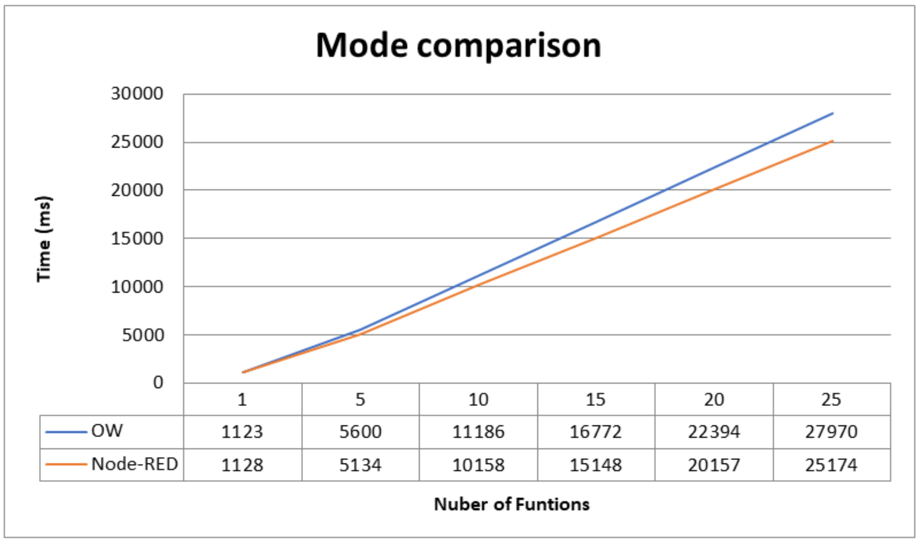


```
▸{ _msgid: "b9d34cc2.df739", payload: object, iterations: 3, delay: 2,
req: object … }
9/14/2021, 7:10:41 PM  node: f9511261.8edfd
msg : Object
▸{ _msgid: "b9d34cc2.df739", payload: object, iterations: 2, delay: 2,
req: object … }
9/14/2021, 7:10:41 PM  node: f9511261.8edfd
msg : Object
▸{ _msgid: "b9d34cc2.df739", payload: object, iterations: 1, delay: 2,
req: object … }
9/14/2021, 7:10:41 PM  node: 6fec2721.679488
msg.payload : string[26]
"{"iterations":3,"delay":2}"
9/14/2021, 7:13:23 PM  node: 6fec2721.679488
msg.payload : string[29]
"{"iterations":3,"delay":1000}"
```

Figure 54: Artificial Delay flow to mimic dynamic sequence numbers in Node-RED

Experiment Results

After the result gathering, we compared the execution times of the two modes of execution by designing Result Graphs for each mode (Mode comparison image in Figure 55). We clearly observed that for one warm function execution the execution time is approximately the same for the two modes, which is the same conclusion derived in the standalone hello world functions of the previous section. However, the more functions that are being included and executed (as a sequence) the more the mean average tends to be lower for the Node-RED case. Given that the inner delay of each function in the sequence is the same, the main source of differentiation is the inter-function communication delay (Figure 56). Consequently, the delay difference between the two modes gets progressively higher for larger sequences. We conducted the experiment with 1, 5, 10, 15, 20 and 25-function sequences with 40 samples for each of the two modes respectively on warm containers, for 1000 milliseconds of artificial delay per function. From the total times, the ifc measure can be extracted (Figure 57). So, for example in the Node-RED mode case, on 15 Functions we get 15148 average execution time. From this value we extract the static delay of each function (15*1000) and the remaining part is divided by the number of functions used in the specific case (15). This indicates the average time spent between function calls in each mode. The results appear in the IFC graph below. The fact that Node-RED ifc times per function appear higher in lower function numbers and then start to get lower can be attributed to the fact that any initialization times such as initial hot container reuse delays, are divided between the number of functions. Thus, when function numbers are low, the effect of this initial delay (happening once in the Node-RED mode but multiple times in the OW case) is higher and diminishes as the number of functions grows, since it is averaged on them.



	1	5	10	15	20	25
OW	1123	5600	11186	16772	22394	27970
Node-RED	1128	5134	10158	15148	20157	25174

Figure 55: Difference in Orchestration Times between OW and Node-RED compared to the function numbers

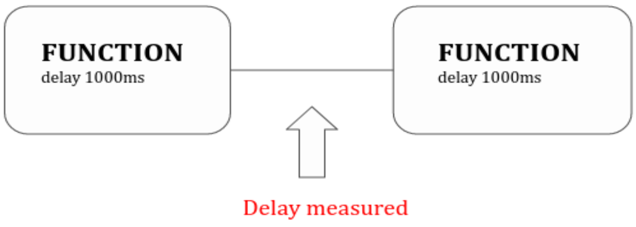
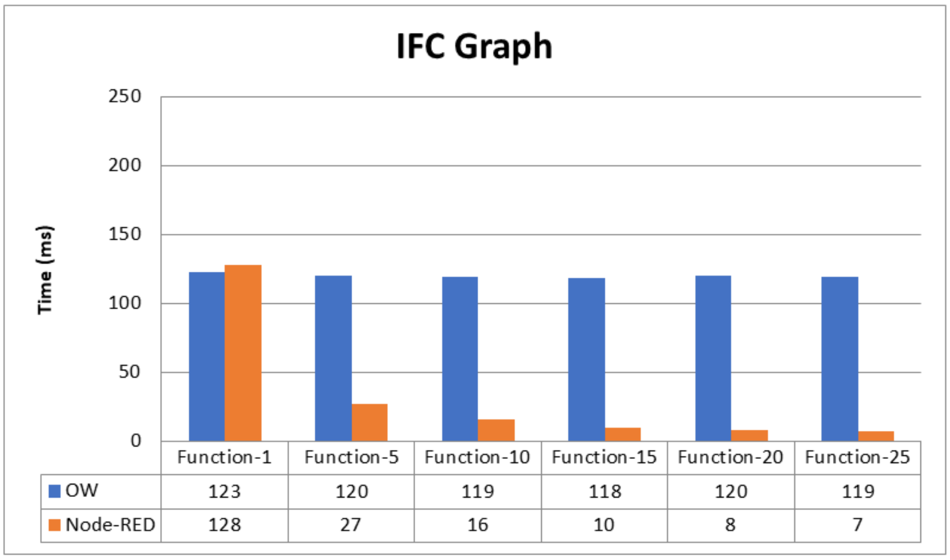


Figure 56: Interfunction Delay Measurement



	Function-1	Function-5	Function-10	Function-15	Function-20	Function-25
🟦 OW	123	120	119	118	120	119
🟧 Node-RED	128	27	16	10	8	7

Figure 57: IFC mode comparison

From the previous graph, it is evident that we can use the average time for the OW case, given that this is independent of the number of functions used. However, for the Node-RED case, as mentioned above, the initialization time, happening once, significantly affects the average produced. To have a more accurate approach, we need to estimate the ifc time from the acquired measurements.

Estimation of the parameters for ifc

As a next step we defined an analytical function to detect the dependence of total flow delay time on intermediate average delays and inter-function communication:

$$T = Tinit + n*delay + n*ifc$$

where:

- **Tinit** is any initial environment or other initialization time happening once
- **delay** is the preset delay of each function in the sequence,
- **n** is the number of functions (in sequence)
- **ifc** is the inter-function communication, the delay needed to go from one function to the next. It is kept as n and not (n-1) that are the number of links in an n-sequence since we consider that in this way the ifc can model any repetitive actions needed in each step, including the first step. For example, these could be container hot init times for the OW case that are needed in each function invocation. In the case of Node-RED it would include only the time needed in Node-RED to pass from one function to the next

From the experiments we have the total delays **T**, the delay of each function is set and known as well as the **n** used in each case. Given that we have measurements for different values of n we can apply a simple regression to estimate the parameters **Tinit** and **ifc**. The Pearson correlation coefficient for the dataset is 0.9105, indicating a strong linear relationship. Therefore, the previous equation can be transformed to:

T-n*delay=n*ifc+Tinit or y=a*x+b

where **x=n** and the coefficients are the **ifc** (a) and **Tinit** (b). The curve fit can be optimized via a typical method such as the ols [33] function of GNU Octave, giving the values of 128.36 for Tinit and 1.69 for the ifc (in the millisecond range) for the Node-RED case. When comparing the predicted outputs to the measured ones, the MAPE of the comparison gives a 0.07% value, indicating a good match.

Similarly, for the OW mode, we have a reverse effect, the **Tinit** is set at 2.11 and the ifc at 118.84. The latter was expected since the individual values are very similar and close to the average value of the ifc graph and indicates the fact that we need further initialization in each step.

The above times can be off-set by the estimated difference in the cold case, which can be subtracted from the measurements of the previous section, i.e., for the Node-RED case the difference between a cold and a hot start is 7.6-0.227= 7.373 seconds, and for the OW case 2.5-0.252=2.248 seconds. These times can be added as penalties in the final function (Table 9).

Table 9: Function sequence hot/cold execution times with the different runtime modes

Mode	Hot Function Sequence Execution (msec)	Cold Penalty (msec)
OW native case	T = 2.11 + n*delay + n*118.84	+2248
NR function case	T=128.36+n*delay+n*1.69	+7373

From these functions we can easily create parameterized plots to observe how the estimated total execution differs for different function numbers and delays (Figure 58), including or not the cold penalty. In the hot case the Node-RED mode is better in all cases, while in the cold case it starts being better than the OW native after approximately 40 functions. Indicatively, in some cases (e.g., OW 100 msec delay) the orchestration delay per function is higher than the actual computation needed inside the function (100 msec), leading to overheads of over 50%.

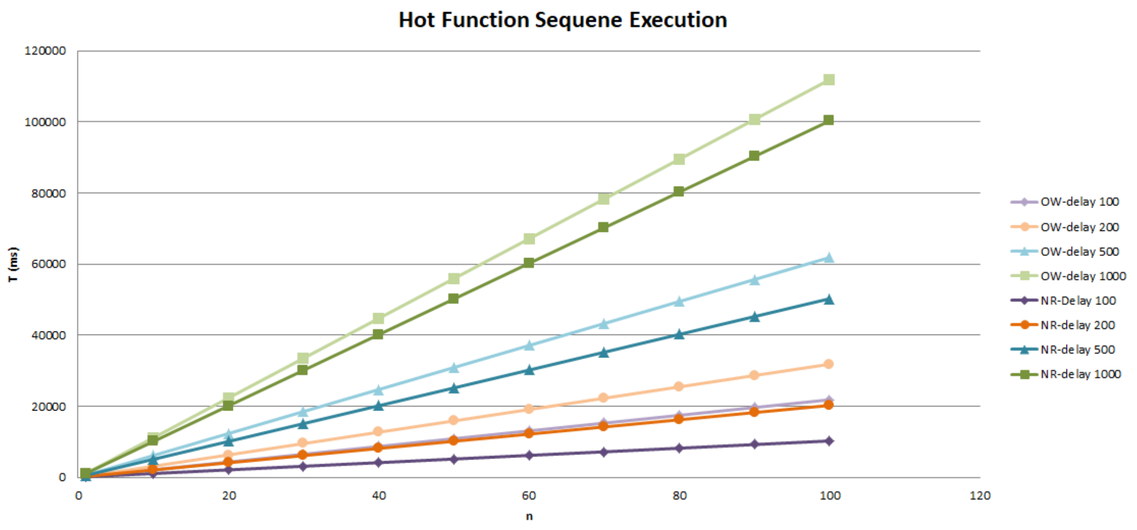


Figure 58: Hot Function sequence execution

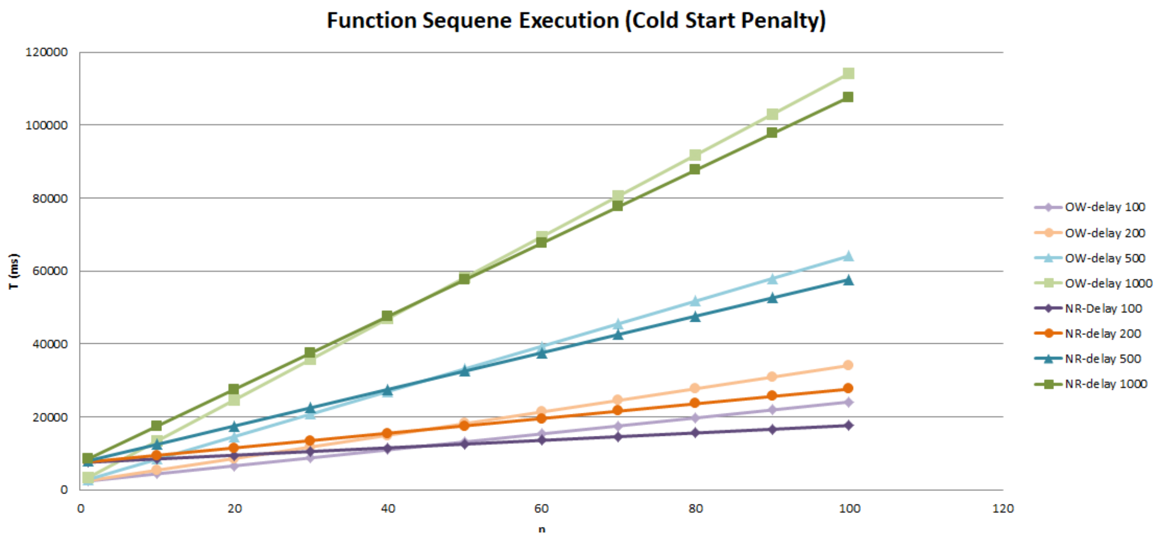


Figure 59: Function estimated total execution for different function numbers and delays

The orchestration overheads are extracted and presented in Figure 60, by extracting the n*delay factor from the function. In this case the overhead does not depend on the inner delay of each function, so we can better observe the pure total overhead in terms of time for the orchestration, as it evolves for a growing number of functions in the sequence. The borderline value of 40 functions in the cold start case is considerably high given the current landscape of FaaS applications, however this low function usage per application might also be limited by the capabilities of the available design and development environments of current FaaS platforms. In any case, the developers should make an informed decision on which mode to use, also considering the trade-offs of easier design and adaptation in Node-RED at the expense of cold start performance, the ability to create more complex workflows (as analyzed in D3.1) and based also on whether cold starts are expected to be frequently encountered. This can be also coupled or affect relevant decisions on the platform configuration layer, indicating the need for a higher number of pre-warm containers used for the Node-RED runtime based on the anticipated traffic.

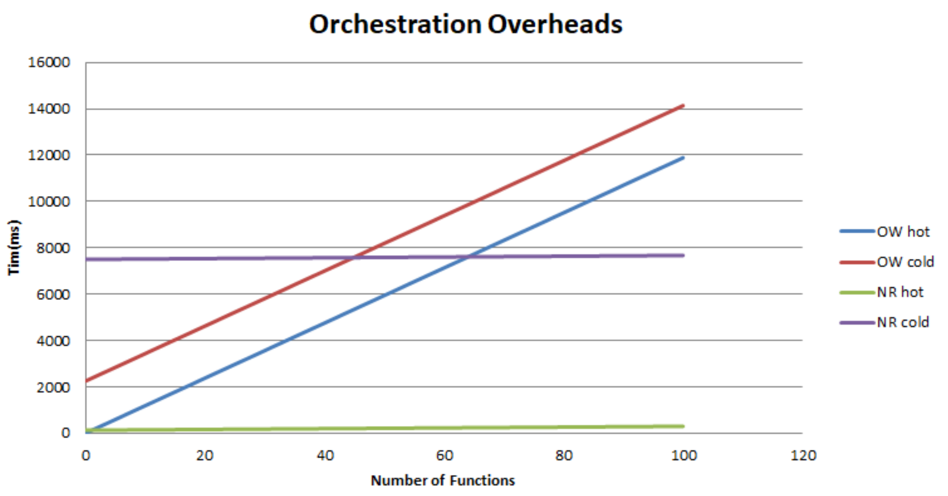


Figure 60: Orchestration Overheads

In the future, the artificial delay included in the delay flow can be replaced by the benchmark average delay values that are acquired through the FunctionBench execution on candidate types of resources. Furthermore, potential validation for cases that have not been included in the estimation process will be investigated. However it needs to be stressed that this approach is not used for predicting the execution time, since the latter may be affected by other parameters (e.g. multitenant executions on the same node, number of prewarm containers existing and concurrent invocations, queuing times in OW due to high traffic, etc.) but it can be used to investigate the limits between modes of orchestration to use (based on native OW sequences or Node-RED orchestrated ones), based on the number of functions in a workflow and an estimated average delay of them. A more detailed model could be pursued (example in Figure 61), created in a similar fashion as the RA model.

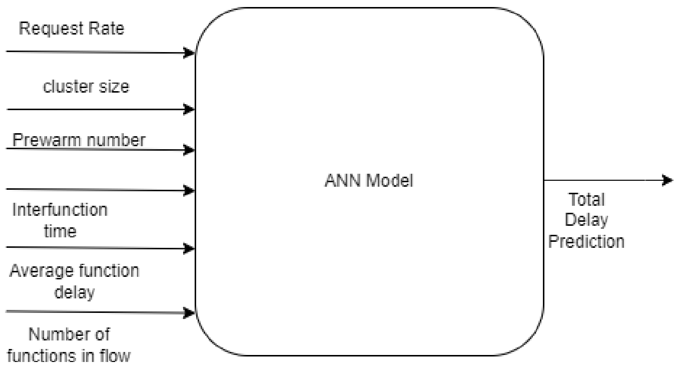


Figure 61: Proposal ANN model in a similar fashion as the RA model

3.3.5 Execution Model transformation for model querying results

As mentioned in Section 3.2, one of the differences of the current version of PEF is the transformation to a serverless model querying process. To measure the impact of this, a series of executions was performed for 150 runs at a frequency of 1 request per second to match the measurements of the original service implementation [16]. The results appear in Figure 62. It is indicative that due to the reuse of the already spawned containers, a typical feature of OpenWhisk, a 20x benefit in terms of execution time is gained for the response time of the FaaS service, when compared to the original service version that spawned a new container for each request.

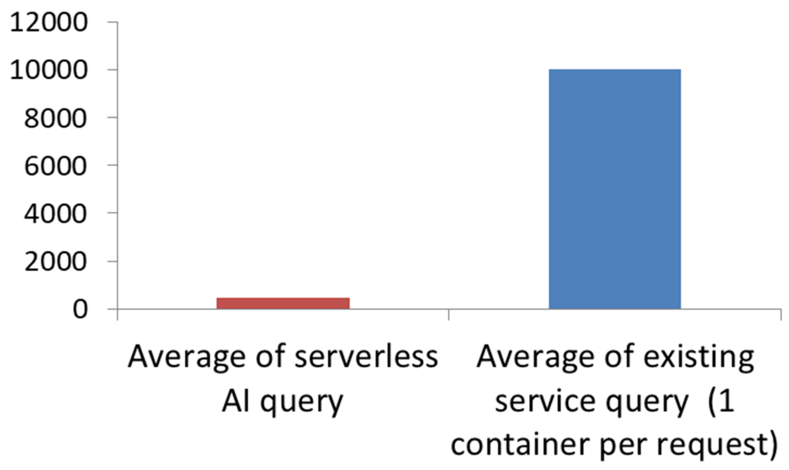


Figure 62: PEF transformation to serverless model measurement impact

3.3.6. Benchmarking of ehealth across different clusters

In this section, we present the outcomes of an extensive experimentation process conducted with three different FaaS systems, aiming to benchmark the performance of the eHealth application across various clusters with different configurations. A FaaS system (in this case OpenWhisk) typically relies on the queue based load leveling pattern with competing worker consumers [34], as depicted in Figure 63.

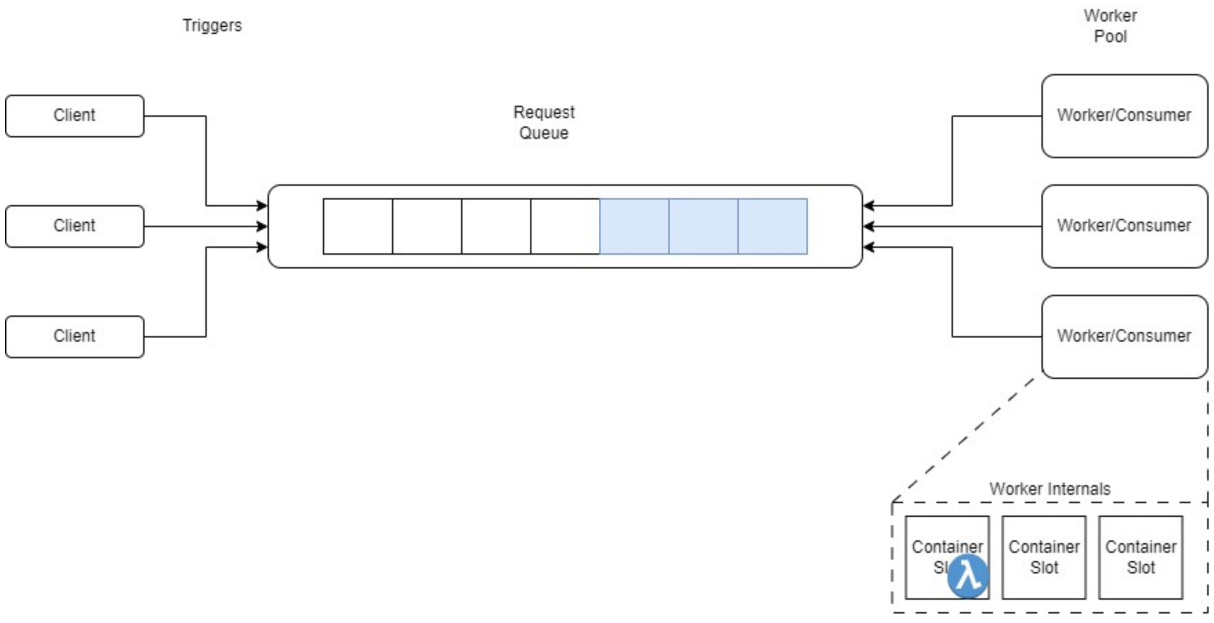


Figure 63: Overview of a FaaS System Architecture

Incoming requests are queued so that they do not create congestion on the back-end, while worker nodes consume them (at their own pace) on a First Come First Serve (FCFS) basis. Through this architecture, back-ends are relieved from bursty traffic and automatic load balancing is achieved, while scalability can be regulated by adding or removing worker instances based on auto scaling approaches (e.g. like the ones mentioned in D3.2). Wait time is the time spent in the queue, initialization time is the time it takes to raise a container and execution time is the actual processing duration inside the function container.

This generic FaaS architecture introduces a system uniformity since all registered functions follow the same execution model. Thus, generic modeling mechanisms, such as queuing models [35] can be used for predicting a system's performance in relation to factors, such as response time, wait time in the system, concurrent clients in the system, etc., while needing generic inputs that can be easily obtained (e.g. incoming request rate, function duration, etc.). Thus, they alleviate the burden of gathering sufficiently large and diverse datasets needed for training black box methods like ANNs, while they can easily produce simulated results to help drive decision making (e.g. on the size of the used cluster).

The primary goal of this analysis is to gain insights from the data initially acquired during the extensive experimentation process documented in [12], comparing performance across the three different FaaS systems. The aim of the process is to calculate example needed parameters for modeling approaches, validate potential assumptions required by different modeling methodologies as well as highlight some interesting phenomena that occur during the specific system operation. Additionally, understanding the performance of each location within a multi-cloud-edge setup can contribute to higher-level scheduling mechanisms, enabling the optimal distribution of incoming requests based on specific constraints. Effective resource allocation and scheduling play vital roles in ensuring overall performance and cost-effectiveness in such environments [36], [37], [38], [39], especially when balancing local resource contention in small clusters and latency-increasing execution in larger ones.

3.3.6.1. Experimental Setup and Data Presentation

Experiment Setup Description

In the experimental setup, data is collected from three diverse cloud/edge locations (Harokopio University, AWS Sweden and Azure Netherlands), each with unique characteristics. In each case, one VM node was used (with different characteristics in terms of CPU and memory) inside which OpenWhisk was deployed. The function invocation rates employed were 12, 30, and 60 messages per minute, while the test function memory was set to 256 MB, 512 MB, and 1024 MB. Through this feature, we are also able to regulate the maximum number of containers that can be launched to serve incoming function requests. This number is the ratio between the node memory available to the OpenWhisk process and the function memory used in each run. Hence, it needs to be viewed as the maximum number of available worker servers (or the c parameter in e.g. an M/M/c queuing model). The test function itself was the ehealth function presented in [12] and consists of a tensorflow-based AI model for inferring a patient's state.

Load generation is performed through the PHYSICS Load Generator, including the need to get the result through asynchronous calls, meaning that the client cannot block waiting for the request, but needs to poll afterward to get the result. This is the typical way through which OpenWhisk serves the results of an executed function. Furthermore, the client can log different timestamps in the process, like initial client sample time, and total response time, as well as process FaaS results in order to export provided statistics, such as wait time in the FaaS system, initialization time for the function container and pure function execution time (or service time).

The relevant information on the experimental setup appears in Table 10.

Table 10: eHealth Experiment Configurations

Testbed				**Test Function Memory (MB)**		
Testbed	Set Rate (msg/min)	Node Memory (GB)	Cores	256	512	1024
				Max Worker Containers		
HUA	12, 30, 60	8	4	32	16	8
AWS	12, 30, 60	8	4	32	16	8
AZURE	12, 30, 60	1.5	8	6	3	1

In order to eliminate any interference from initial cold starts on the results, a pre-warming run was conducted for each case. This step was necessary due to the inclusion of container initialization time in the function execution duration in OpenWhisk, despite being reported separately. Once the warm containers were available following the pre-warm run, data collection took place during the main run for a duration of 600 seconds for each case. A total of 4890 samples were collected throughout the process and published in[12].

The performance degradation from the concurrent containers in the node can have extreme effects. This could be reduced from one point of view (CPU sharing time) if strict scheduling strategies (like real time scheduling) and CPU quotas are used. However, even in this case, studies [21] have shown that still degradation can be extremely significant. For reducing the back-end stress and user costs, dynamic batching approaches ([40], [16]) have also proven to be very effective.

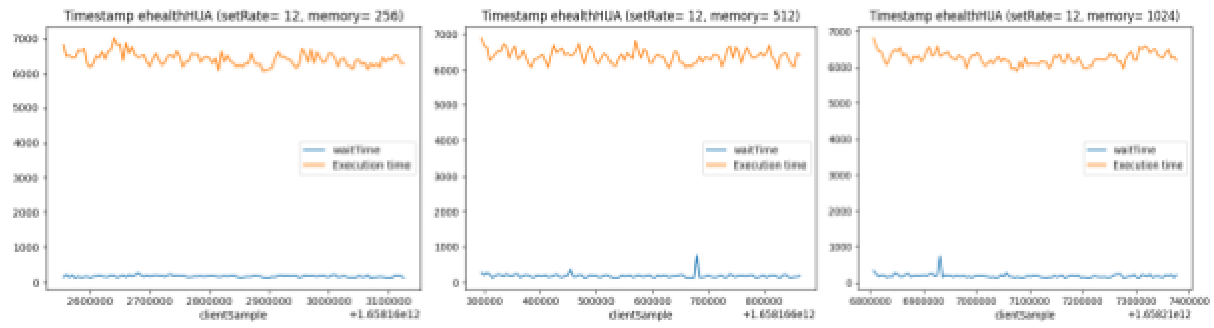
Raw Data Collection Presentation

The outcomes of the experimental analysis are presented, with a primary focus on the aspect of time from the start of each experiment. Through a series of various scenarios, the influence of different variables including function memory, set rates and cluster type, on the overall system performance. By examining these graphs, and visualizations, valuable insights into the behavior of the eHealth application are gained across diverse clusters and configurations while trends, patterns, and correlations can be identified, helping to understand how different variables affect system performance.

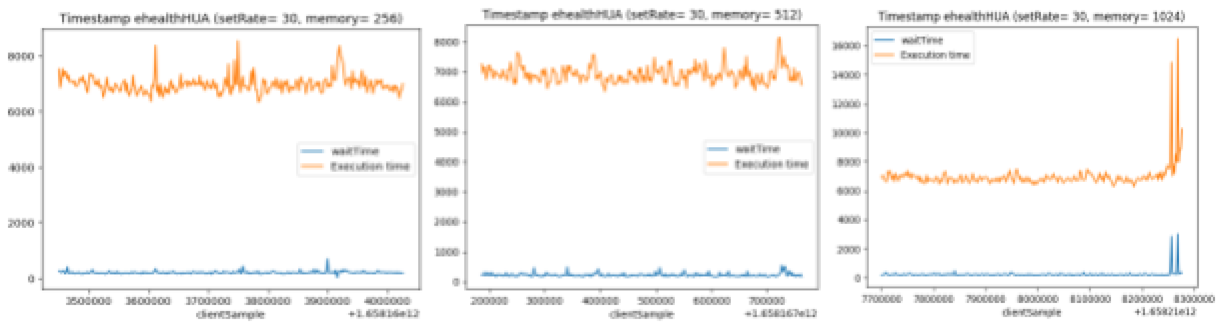
One of the key aspects of the analysis is observing how the system performance evolves over time during the experiment. This temporal perspective allows us to identify any potential trends, patterns or correlations that emerge as the experiment progresses.

HUA Testbed Results

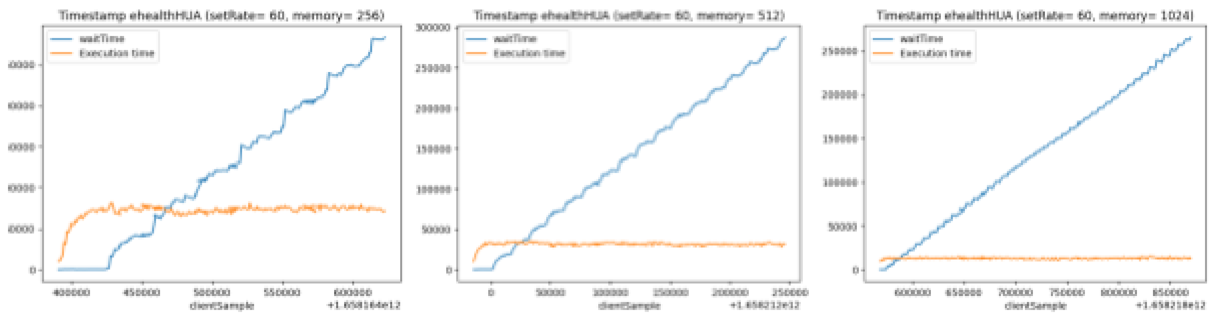
By examining the first series on the HUA testbed (Figure 64), it is evident that in the low rate case (i.e. 12 msg/min) the system is very stable, having a small wait time that is mainly functional (e.g. waiting for the incoming request to be forwarded across the various sub-components of the FaaS architecture).



(a) setRate12_memory256 (b) setRate12_memory512 (c) setRate12_memory1024



(a) setRate30_memory256 (b) setRate30_memory512 (c) setRate30_memory1024



(a) setRate60_memory256 (b) setRate60_memory512 (c) setRate60_memory1024

Figure 64: Performance plot of ehealthHUA

The amount of memory does not affect the function performance in this case, since in all 3 memory variations the execution time is very similar (6.5 seconds), indicating that the minimum memory amount applied is enough. If we apply Little's law in this case, with an incoming rate of 1 request per 5 seconds and a response time 6.5 seconds, we get an average of 1.3 customers in the system. This means that on average 2 max containers are needed for serving the functions in these scenarios and they run concurrently for about 30 percent of their time. A similar case applies for the medium rate, although in this case we have 1 request per 2 seconds and a slightly increased execution time of 7 seconds leading to about 3.5 customers in the system and, thus, 4 max concurrent containers. However, in the case of the high rate (subfigures g-i), the system becomes unstable as is evident by the constantly rising wait time, although there are in principle 32,16 and 8 workers (max container slots) and this should be enough to accommodate the increased traffic (e.g. based on typical M/M/C formulas).

However there is a significant difference in the case of FaaS compared to the application of queuing models in other domains (e.g. a supermarket or a calling center model).

As seen in Figure 63, a FaaS worker node internally divides its computational capacity into smaller slots (or container slots). Thus, a number of function containers can be squeezed into the same worker node depending on memory availability. This means that the overall resources of the node are now shared by the concurrent processes which leads to significant degradation in the function execution performance, due to direct (cpu time) or indirect (cache contamination) resource sharing.

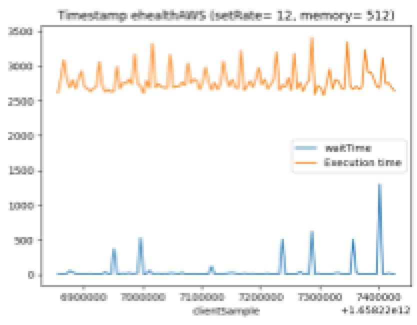
In most cases queuing model approaches assume that the service rate of the system is not dependent on the system state (in this case the number of concurrently served customers/containers in the system) but on a distribution that relates primarily to differences in the function execution itself. In a supermarket analogy, one customer may require more time to be served compared to another (because they have a larger cart) but the time it takes to be served does not depend on the number of tellers in the supermarket. In reality, what happens is that the serving time (function execution duration) will also depend on the amount of tellers in the system (because they waste time chatting to each other for example).

To quantify this increase, one can observe Figure 64(g-i), in which the system is operating using the max available containers. In the case of Fig. 63(g) the function execution duration increased from 6.5 seconds to ~ 70 seconds. In the case of Figure 64(h), we have 16 max containers (and a service time of ~35 seconds) while, in Figure 64(i), we have 8 max containers and a service time of ~12 seconds. So it seems to be following a rather linear reduction based on the percentage of cpu core time assigned to each container, once the container numbers exceed the amount of available cores (4 in the case of the HUA testbed). This is reasonable since after passing the ratio of 1 container per core the core time is split between the competing containers.

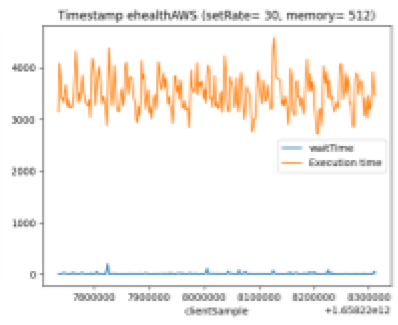
This leads us to a chicken and egg problem when trying to apply queuing models in such environments. How can we estimate how many customers in the system exist at any given time, given that this depends on the service rate, and on the other hand, how can we estimate the service rate when it depends on the customers in the system. An iterative approach could be potentially followed, based on split time windows in order also to adapt to incoming traffic variations, in which either the execution duration or the customers in the system are monitored and from this parameter we can estimate the remaining ones.

AWS Testbed Results

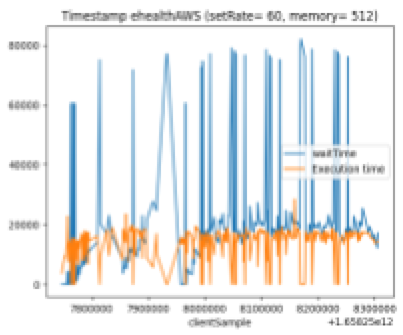
For the AWS case (Figures 65(a)-(f)), we can observe similar behavior, with two exceptions. Specifically, the 256 MB function memory consistently leads to failures in the AWS testbed due to function memory being too low. The reason, compared to the HUA testbed, is that in this case we are using Kubernetes container management system, which kills the container even if a slight memory violation occurs. On the HUA testbed, we use the Docker Engine, which is not that strict.



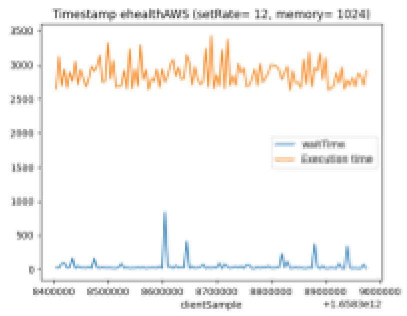
(a) setRate12_mem512



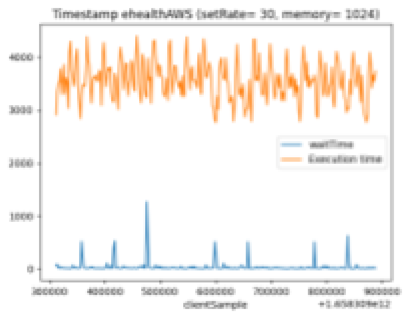
(b) setRate30_mem512



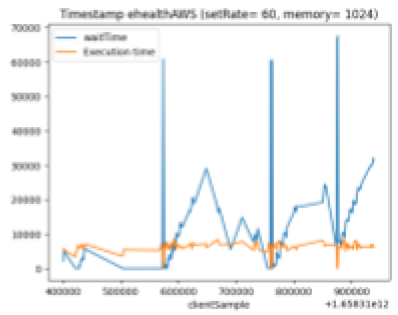
(c) setRate60_mem512



(a) setRate12_mem512



(b) setRate30_mem512

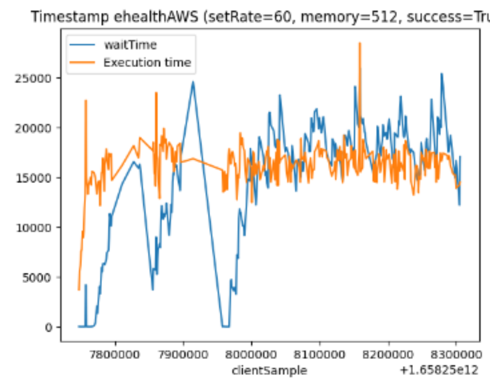


(c) setRate60_mem512

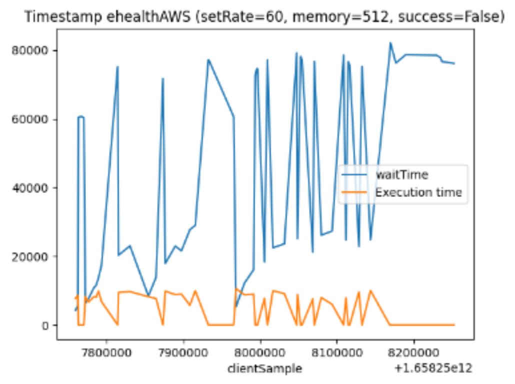
Figure 65: Performance plots of ehealthAWS

Probability of Failures Due to Image Pulling Bottlenecks

During the benchmarking process, anomalies were observed in AWS high request rate scenarios in Figures 65(f) -(g). While the system appears to be stable (in the sense that it has an increased but manageable service time), there are a number of spikes in waiting time, as well as an increased number of failures in the requests. In order to further investigate this phenomenon, we broke down the responses into successful and unsuccessful ones, appearing in Fig.66.



((a)) success true



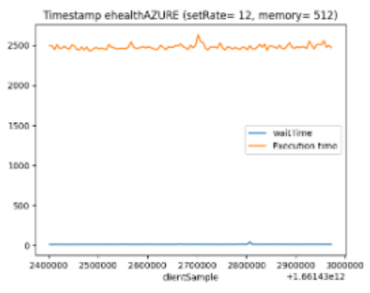
((b)) success False

Figure 66: Behavior of the ehealthAWS with setRate=60 and memory=512

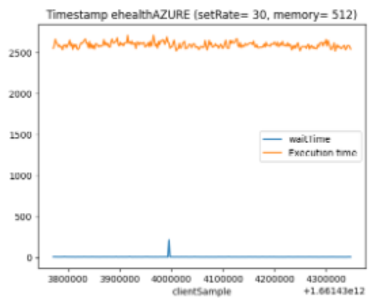
After examination, it was determined that these failures were due to numerous concurrent requests for pulling and launching the function container image and can be attributed to the image registry and the respective storage system. A relevant ErrImagePull event was generated due to this spike in image pulling, which was caused by the spike in cold starts. Cold starts are treated in literature significantly, however what this analysis shows is that they also need to be linked not only with extended initialization times but also with higher probabilities of failure. Thus, it should be added as a stage in a kind of Markov chain analysis.

Azure Testbed Results

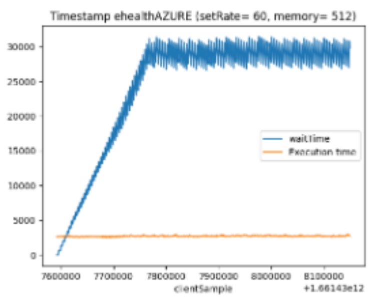
The benchmarking results from the Azure testbet are presented in Figure 67. In the low request rate case, the system demonstrates stability and performs slightly better compared to the AWS testbed, given that the VM is based on a slightly more advanced cpu. Several comparisons can be made between the two environments. Comparing between the rate 30-512 MB cases, it can be noted that the AWS execution time is raised to 3.5 seconds. On the other hand, the Azure case is more stable around 2.5 seconds (as the baseline value in the low rate scenario). In the 30-1024 MB case, AWS still maintains the 3.5 seconds execution and Azure the 2.5 one, however since Azure has the restriction of only 1 worker, requests are queued given that they arrive at a rate of 1 every 2 seconds. The key takeaway from these testbed results is that, if we want to maintain a very stable execution time, then we need to do it at the expense of higher waiting time.



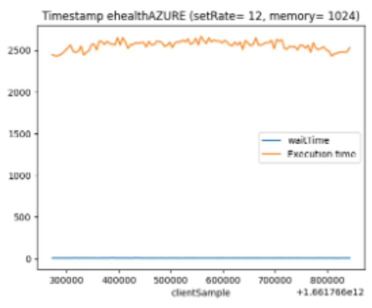
((a)) $setRate12\_mem512$



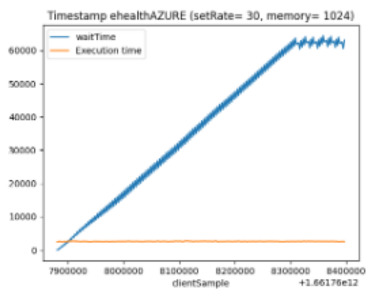
((b)) $setRate30\_mem512$



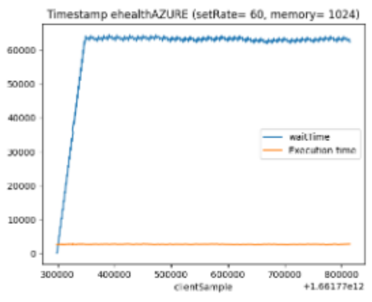
((c)) $setRate60\_mem512$



((d)) $setRate12\_mem1024$



((e)) $setRate30\_mem1024$



((f)) $setRate60\_mem1024$

Figure 67: Performance plots of ehealthAZURE

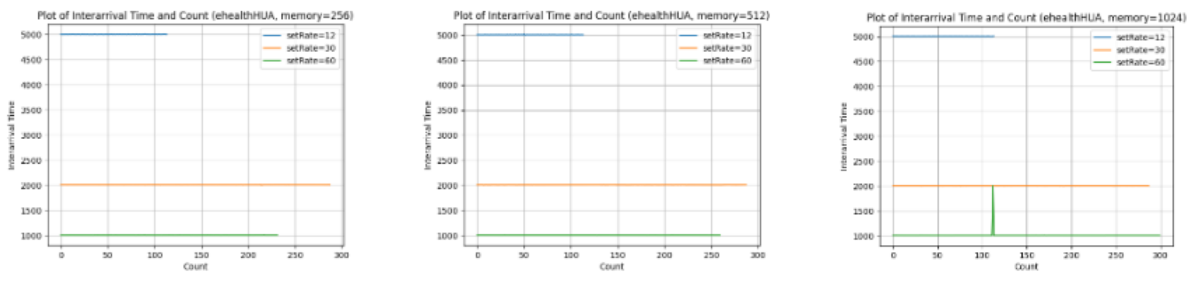
Comparing the performance of eHealth applications in the Azure testbed with the previous AWS and HUA testbeds reveals distinct characteristics in terms of execution time, waiting time, and queuing behavior. Each testbed demonstrates unique strengths and limitations based on its underlying infrastructure and resource allocation. Understanding these variations can lead to decisions about resource provisioning and allocation strategies based on the provider's goals.

Cut-off points and Load Generation Analysis

From the AWS and Azure cases, some peculiarities were observed in the exported graphs. For example, in the case of Azure (Fig. 67 (c)-(e)-(f)), the queue seems to be constantly rising, until a point in which it becomes stable. Given that the arrival rate is higher than the departure rate from all servers, this queuing time should be constantly rising. Initially it was considered that it might be a problem with the load generator. FaaS platforms typically do not allow blocking calls for a delay larger than a specific limit, thus async invocations should be performed and the result acquired through a polling process. Thus, a high number of actively monitored functions could put a large strain on the load generator, not enabling it to meet the desired rate. Furthermore, the load generator is built in such a way that it logs only the requests from which a result was obtained from the FaaS platform.

In order to evaluate the performance of the load generator, each request timestamp was logged on the client side and compared to the target rate. No deviation was found on that side, following the desired request rate. Then, we plotted the received requests in the three scenarios that appear in Figures 68-70.

The samples are sorted based on the timestamp of the initial request (x-axis) and the time difference between sample n and (n-1) (y-axis). Hence, a consistent request/response rate should be observed, similar to the cases of Figure 68, thus a straight line indicating the set rate in requests per minute.

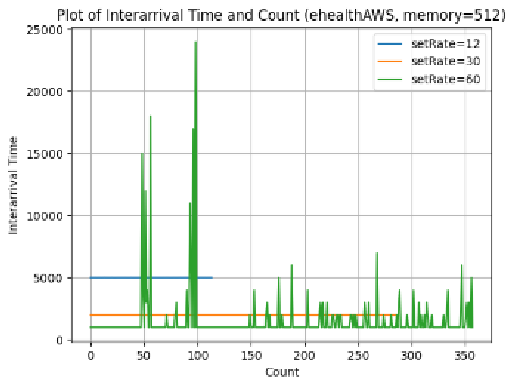


((a)) ehealthHUA_Memory256 ((b)) ehealthHUA_Memory512 ((c)) ehealthHUA_Memory1024

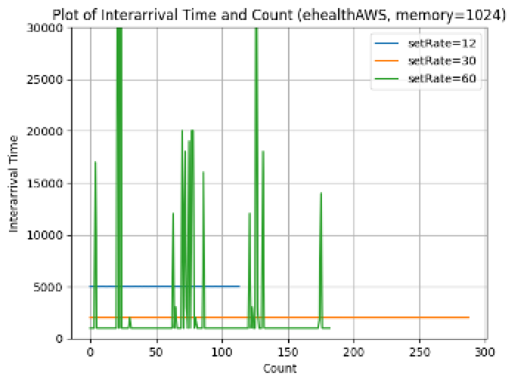
Figure 68: Inter-arrival time (seconds) between processed request samples in ehealthHUA

The aforementioned case in the HUA testbed appears to be working as expected. The fact that we have fewer samples in the HUA case in the 60 rate scenario can be explained due to the high execution time in this case. The load generator waits for the 600 seconds of each experiment duration and then proceeds to calculate the samples that arrived up to then. Thus, samples that arrive after that mark are neglected, hence the low sample count collection in the high rate/high delay case.

However, in the cases of Amazon (Figure 69) and especially Azure (Figure 70), the samples in the high rate case (and in some cases in the medium rate towards the end) do not follow the specified rate. Furthermore, this difference appears to be very specific, i.e. instead of having samples every 1 second we get a specific sequence like 1,4,1,3,1, etc.



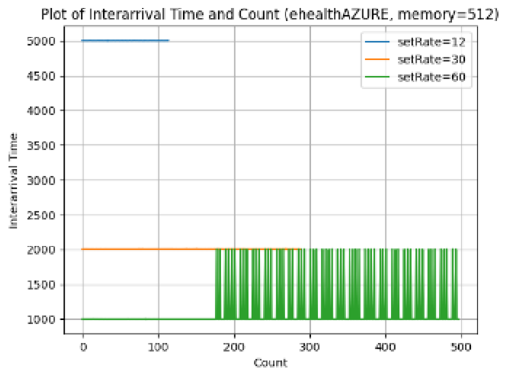
((a)) ehealthAWS_Memory 512



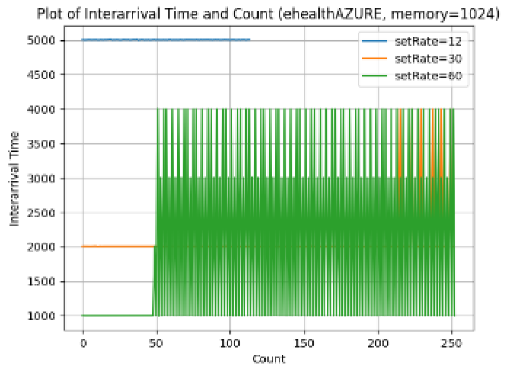
((b)) ehealthAWS_Memory 1024

Figure 69: Inter-arrival time (seconds) between processed request samples in ehealthAWS

This was due to the fact that in FaaS platforms there are two cut-off points that can be set. One cut-off point is at the overall level, indicating that the platform should reject invocations higher than e.g. 60 per minute. Another cut-off point is the fact that one can determine the maximum number of concurrent, active invocations. It is due to this limit that this queue leveling should be attributed. Once this limit is reached, the platform rejects the next request until one of the previous ones has been finished and that is why the interarrival times of processed requests are always at a multiple of 1. In the case of HUA this limit was set much higher (400 invocations per minute) than the AWS and Azure cases in which the specific cut-off point was set to 60 invocations per minute, hence the difference in the processed request sample plots.



((a)) ehealthAZURE_Memory 512



((b)) ehealthAZURE_Memory 1024

Figure 70: Inter-arrival time (seconds) between processed request samples in ehealthAzure

These details were gathered and calculated in Tables 11-13 for the three cases respectively.

Table 11: Effective Rate and Request Acceptance Probability for test name = ehealthHUA

Target Rate (request/sec)	Memory (MB)	Actual Samples	Expected Samples	Effective rate	Probability of accepted requests
0.2 (12 request/min)	**256**	**114**	**114**	**0.199**	**1**
0.2 (12 request/min)	**512**	**114**	**114**	**0.199**	**1**
0.2 (12 request/min)	**1024**	**114**	**114**	**0.199**	**1**
0.5 (30 request/min)	**256**	**288**	**288**	**0.499**	**1**
0.5 (30 request/min)	**512**	**288**	**288**	**0.499**	**1**
0.5 (30 request/min)	**1024**	**288**	**288**	**0.499**	**1**
1 (60 request/min)	**256**	**232**	**232**	**0.999**	**1**
1 (60 request/min)	**512**	**260**	**260**	**0.999**	**1**
1 (60 request/min)	**1024**	**300**	**301**	**0.995**	**0.995**

Table 12: Effective Rate and Request Acceptance Probability for test name = ehealthAWS

Target Rate (request/sec)	Memory (MB)	Actual Samples	Expected Samples	Effective rate	Probability of accepted requests
0.2 (12 request/min)	**512**	**114**	**114**	**0.199**	**0.999**
0.2 (12 request/min)	**1024**	**114**	**114**	**0.199**	**1**
0.5 (30 request/min)	**512**	**288**	**288**	**0.499**	**1**
0.5 (30 request/min)	**1024**	**288**	**288**	**0.499**	**1**
1 (60 request/min)	**512**	**358**	**558**	**0.641**	**0.641**
1 (60 request/min)	**1024**	**183**	**539**	**0.339**	**0.339**

Table 13: Effective Rate and Request Acceptance Probability for test name =ehealthAZURE

Target Rate (request/sec)	Memory (MB)	Actual Samples	Expected Samples	Effective rate	Probability of accepted requests
0.2 (12 request/min)	**512**	**114**	**114**	**0.199**	**1**
0.2 (12 request/min)	**1024**	**114**	**114**	**0.199**	**1**
0.5 (30 request/min)	**512**	**288**	**288**	**0.499**	**1**
0.5 (30 request/min)	**1024**	**252**	**288**	**0.499**	**0.976**
1 (60 request/min)	**512**	**497**	**558**	**0.8902**	**0.8902**
1 (60 request/min)	**1024**	**253**	**516**	**0.4901**	**0.4901**

What is most interesting in this case is to calculate from the total experiment duration which was the acquired samples compared to the formally foreseen ones. This can help us determine the effective rate of requests, i.e. the actual part of the arrival rate that indeed enters the system. Furthermore, the probability of a request for a given scenario to enter the system is important. For example, in cases where we need to foresee such a state (such as in Markov chains), as well as the probability of transitioning to that state.

The difference for higher rejection in the case of AWS compared to Azure may be attributed to the fact that AWS due to the concurrency overheads and the higher number of allowed containers has a higher execution time, thus the rate with which requests leave the system is lower. This leads to more requests being rejected due to the waiting queue being full.

Cost Model Consideration for Trade-off between Waiting and Execution Time

Waiting and Execution Time Trade-offs

The distribution of function execution time data for each testbed, as presented in Figures 71-73, provides valuable insights into the performance characteristics and potential container overheads or resource sharing penalties. This is influenced by various factors, including the management and execution of containers, as well as the system configuration and design choices.

This analysis allows us to observe the stability of the executions, as well as use them as potential future inputs into identifying the distribution types to be used in modeling approaches. It can also give insights into the split of the total response time between waiting and execution, based on the cluster setup.

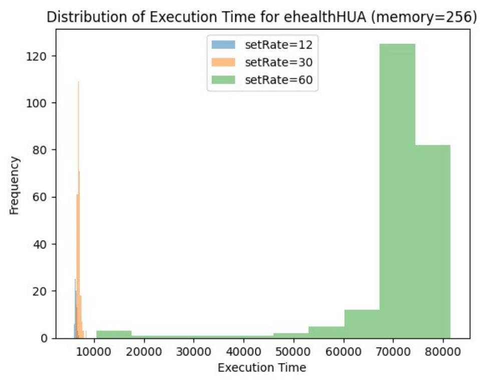
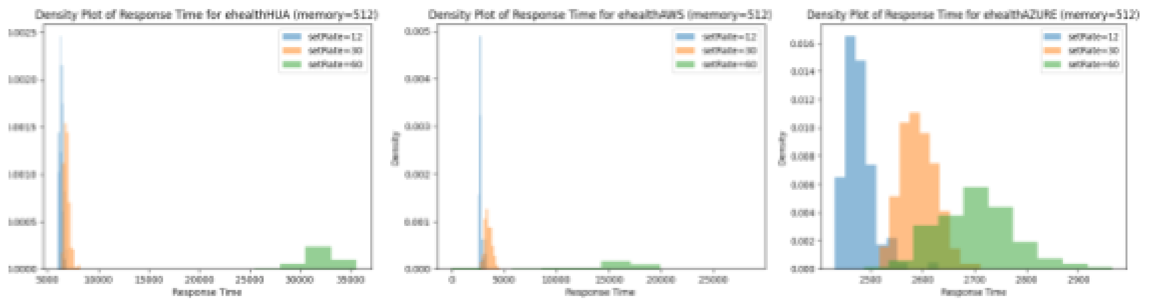
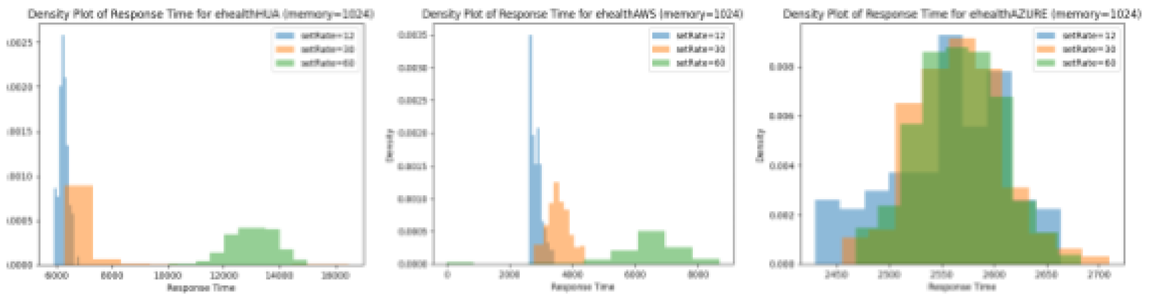


Figure 71: Distribution of Execution Time for Memory 256



(a) ehealthHUA (b) ehealthAWS (c) ehealthAZURE

Figure 72: Distribution of Execution for Memory 512 across Clusters



(a) ehealthHUA (b) ehealthAWS (c) ehealthAZURE

Figure 73: Distribution of Execution for Memory 1024 across Clusters

As an example, if we investigate the graphs of the rate 60-memory 512 case in the AWS testbed (Figure 72(b)), we can observe that it portrays a rather increased execution time, as well as deviation, given that the distribution spans across a large interval (2500-20000 milliseconds). This is reasonable since in this case we can expect that a significantly high number of containers is concurrently active. On the other hand, the Azure rate 60-memory 512 case (max 3 concurrent containers) (Figure 64(c)) is very much concentrated around a small interval (2500-2900 milliseconds). On the waiting time aspect, things are quite the opposite, with AWS portraying the majority of delays up to 20000 milliseconds while Azure over 20000 milliseconds. This is reasonable since with fewer container slots, requests need to be queued to find one available.

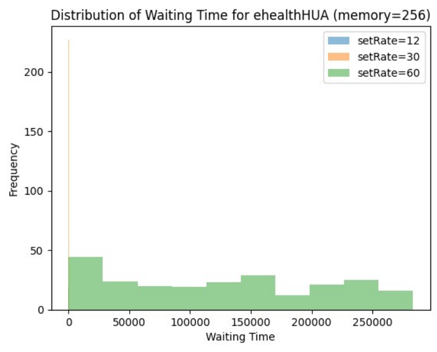
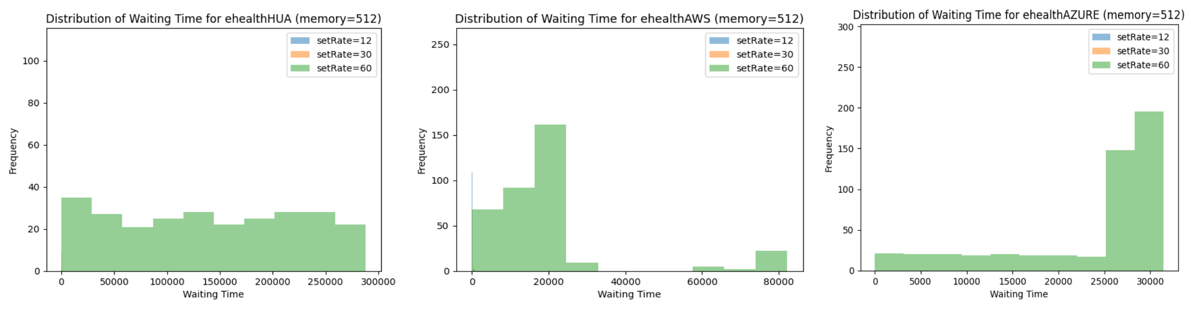
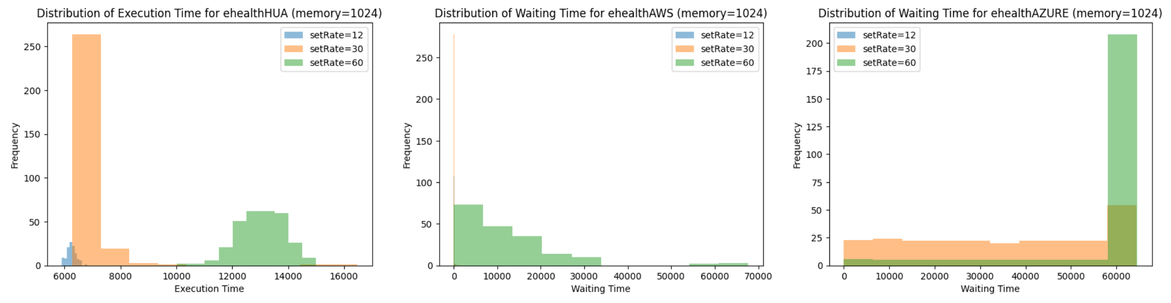


Figure 74: Distribution of Waiting Time for Memory 256



(a) ehealthHUA (b) ehealthAWS (c) ehealthAZURE

Figure 75: Distribution of Waiting Time for Memory 512



a) ehealthHUA (b) ehealthAWS (c) ehealthAZURE

Figure 76: Distribution of Waiting Time for Memory 1024

Cost Model Consideration for Waiting Time vs Execution Time

The cost model for FaaS is typically determined based on the execution time of a given function and the amount of memory it allocates for its container [7]. However, in the context of overloaded clusters, this model may be violated. Under increased load, customers may end up paying more for a degraded service quality due to the concurrency overheads. To prevent such violations, it is necessary to configure the system in a way that prioritizes maintaining a stable execution time, even at the expense of higher waiting time, as seen in the previous section. This ensures that customers are not penalized both with higher costs and a suboptimal user experience.

In order to compare the scenarios and to check how different configurations and loads affect the experienced QoS, Table 14 was created. In this, we consider as baseline the HUA low rate scenario. Then, we calculate the percent differences ((T-Tbaseline)/Tbaseline) of each according to the time of the other scenarios. Hence, several insights can be extracted that can aid in two directions. On one hand, to compare the different available clusters in case of routing requests between them. On the other hand, they can be used to compare configurations and baseline abilities of each cluster.

Table 14: Comparison of Testbed Data for Various Test Configurations with baseline HUA32, AWS16, AZURE4

Testbed		Set Rate (msg/min)								
Testbed	Max Worker	12			30			60		
	Containers	Execution Time	Waiting Time	Total Time	Execution Time	Waiting Time	Total Time	Execution Time	Waiting Time	Total Time
HUA	32	0.00%	0.00%	0.00%	8.87%	14.77%	9.03%	1008.89%	69946.56%	2868.85%
	16	-0.50%	8.63%	-0.26%	7.87%	25.16%	8.34%	394.67%	78205.41%	2494.02%
	8	-2.35%	8.12%	-2.07%	8.78%	32.83%	9.43%	103.48%	73593.77%	2086.26%
AWS	16	-56.45%	-74.02%	-56.93%	-45.45%	-92.54%	-46.72%	122.24%	10746.06%	408.88%
	8	-55.38%	-73.08%	-55.86%	-44.47%	-78.12%	-45.37%	-0.25%	6486.98%	174.77%
AZURE	4	-61.30%	-94.44%	-62.20%	-59.50%	-96.01%	-60.49%	-57.79%	13275.80%	301.95%
	2	-60.07%	-95.31%	-61.02%	-59.94%	19964.47%	480.32%	-59.93%	31940.89%	803.46%

For example, the main concurrency problems start from a ratio of containers/cores > 1. However, in the case of the AWS 60 rate for 8 max containers (double the number of the available cores on the node), although we get almost double the execution time compared to the AWS low rate case, the achieved execution time is almost identical to the baseline HUA scenario that is used as the reference. This gives us an overall good trade-off between execution and wait time, indicated by the fact that this setup has the lowest degradation in the total response time (174%) of the high rate case.

On the other hand, if we want to keep a very consistent execution time, then configurations, such as the Azure testbed, can be used. In these setups, the ratio of 1 container per core is not violated, thus indicating a stable improvement of around 60% compared to the baseline HUA execution. However, the user should also be willing to trade this stability with a higher wait and total time.

Dynamic Resource Allocation and Auto-Scaling Strategies for Performance Optimization in FaaS Platforms

Auto-scaling techniques have been widely studied to manage dynamic container allocation in Kubernetes. Threshold-based auto-scaling techniques have been used to adjust container numbers based on thresholds of CPU or Memory usage rates [41]. Feedback control methods, such as linear-performance-model-based fixed gain [42] [43], adaptive [44], or multi-model switching feedback control methods [45], have been widely used in application resource auto-scaling [46]. Such methods should be applied in order to regulate the maximum container slots dynamically so that the function execution time (or its distribution) is relatively similar over time, given the availability of relevant monitoring data [12].

However, in the case of FaaS platforms, and given the availability of monitoring information across system stages, e.g. wait time, initialization time, execution time, etc., more fine grained elasticity may be applied. For example, if monitoring indicates a rise in execution time, this might mean reducing the memory setting in order to reduce the number of concurrent containers, so that more requests are queued rather than executed directly to avoid concurrency overheads. If, on the other hand, we have such a configuration and we observe high waiting times, it is an indication that further compute nodes should be added to the Kubernetes cluster in a horizontal scaling manner or the used nodes can be increased mainly from a number of available processors point of view in a vertical scaling manner (an according strategy is defined in D3.2 regarding the autoscalers approach).

Transient Phenomena Stabilization

In many cases, knowing the time it takes for the system to stabilize can be beneficial. For example, in some modeling cases such as PID (Proportional– Integral– Derivative) controller-based regulation of resources [47], a temporarily increased derivative of the execution times may lead to corrective actions such as cluster resizing. In some cases however, we may first need to see the steady state before actually deciding on the corrective action since the steady state may still be within the set goals of the system. In that case we could avoid oscillation of the system metrics caused by premature corrective actions. Thus such information that can be extracted for example from Figure 64 (g) (the time it takes to reach a more steady condition in the execution time) can aid in adding such safeguards by adding a relevant lag in the autoscaling mechanisms.

Average Times Investigation

Average Total User Side Delay Investigation

The total user side delay (Figure 77) is measured as the response times of the function invocations from the client generator. As such it includes all intermediate times. From that it can be observed that for the middle memory scenario, the 3 available container slots (1.5 GB Container Memory/512MB function memory=3 slots) in the Azure testbed are sufficient in order to keep wait time relatively short, even in the high rate case. Thus the benefit of the other factors (reduced latency and execution time) prevails. In the high function memory scenario, the slots are half due to the doubling of the function memory, and now the wait time is considerable in the medium and high rate scenarios. Thus in this case the execution on AWS is better.

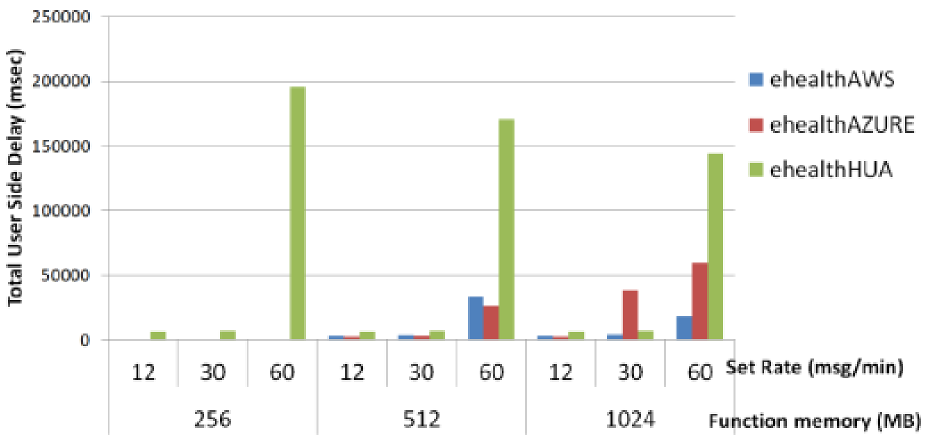


Figure 77: Average Total User Side Response Time

Average Waiting time investigation

The waiting time is reported by OpenWhisk and retrieved by the Load Generator. The average for each scenario is plotted in Figure 78, which shows the effect of the parameter of the Container Pool Memory can be observed. Although the Azure VM is larger (32 GB RAM and 8 cores), using a low value (2 GB) for this parameter limits the available container slots. As a result, queuing times are higher at the high rate (60msg/min) compared to the AWS case, even though the latter is smaller (4 cores and 16GB of RAM). The AWS testbed can handle more concurrent requests by using 8 GB of Container Pool Memory. The Azure cluster starts having wait delays from the medium rate.

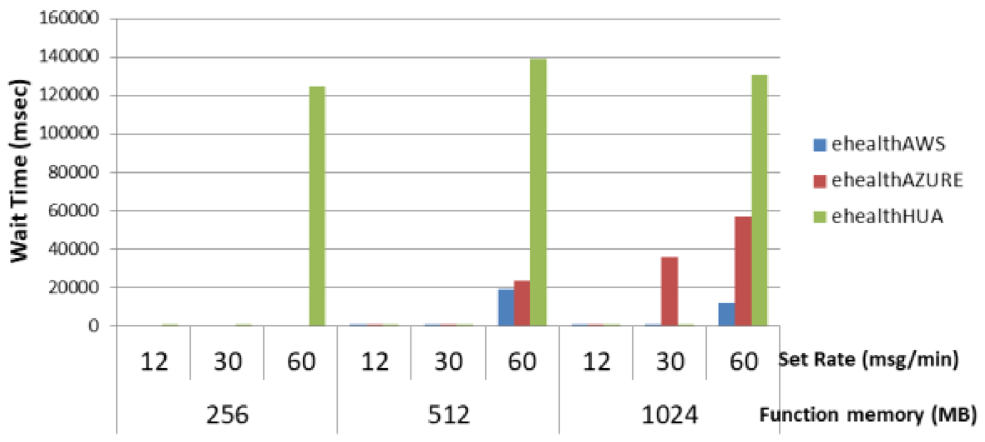


Figure 78: Average Waiting Time for Function Invocation

Note that the number of setup parameters that can affect performance is significant and can have a large effect on the cluster. In this case, when combined with the concurrent container analysis, interesting trade-offs can be explored. Higher numbers of available container slots reduce wait time but increase execution duration due to concurrency.

This is reminiscent of the context switching problem in typical web servers where thread limits need to be set. For cluster sizing, more nodes with less memory per node could help reduce wait time while not skyrocketing interference effects. From an energy perspective, this would help the specific execution to consume less energy if the overall response time is the same, but in one case it consists more of waiting time than execution.

3.3.7. Split Join Pattern Performance Experimentation

3.3.7.1 Scope of Split-Join Performance Experimentation

The current study aims to explore the various performance trade-offs when using the Split-Join pattern. While splitting a computational job into smaller parts is mostly beneficial, this also depends on the size of that job as well as the abilities of the available resources. Too much splitting may result in overheads (during startup, execution, etc.) that will negate the benefits of parallelization. Thus a relevant balance needs to be applied, that will determine the amount of parallelism that can be used. Through acquiring a relevant dataset, an informed decision can be made on what should be the split size or in the future it can be used to train relevant predictive models.

3.3.7.2. Input Parameters

The experiment under investigation involves a variety of inputs due to its singularity and specificity. Initially, the primary input factor, the target function, is an AI model inference one (the same that was presented in Section 3.2.3 and is the backbone of the PEF modeling mechanism). The study aims to examine the execution times of this when a Split and Join is intervened in order to parallelize the number of predictions needed (one row equals one prediction but the function may get an array of inputs upon which an equal number of predictions need to be made). Thus different function inputs are tried out, together with different split sizes, based on which the function input will be split. So for example, if the incoming request needs 100 predictions, and the split size is set to 10, this will result in 100/10 parallel function calls being made. One specific aspect of configuration is also how many maximum containers are allowed by the specific OpenWhisk setup. As mentioned in Section 3.2.5, the container pool memory option dictates that maximum amount, which when reached, will cause any subsequent requests to be queued and thus accumulate wait time in the system. The overall ranges for the experimentation appear in Table 15, resulting in approximately 600 different setups. Each setup is repeated 11 times in order to get an average for each case. The first execution is always deleted in order to remove any cold start phenomena.

Table 15: Input Ranges for the Split-Join Experiment

Parameter	**Range**	**Step**
Input Size	100-10100	500
Split Size	20-100	20
#Max Containers	2-32	6

3.3.7.2. Output parameters

For the outputs we monitored a wide variety of outputs, including timestamps from before and after the Split-Join operations, Start time and End time for each function execution, initialization and wait Time for each fragment to acquire a container. Aside from time related aspects, we additionally monitored the repetition number for each case (ID) and the Status of each activation as well as the flag associated with that.

3.3.7.3. Data Acquisition Process Automation

Given the abundance of setups that need to be measured, an automation flow was created in Node-RED (Figure 79). The flow starts by preparing the function input and the split size dictated by this iteration, feeding it to the Split Join node. The latter breaks it down according to HTTP requests towards the target OpenWhisk function. The input arrays are divided based on their predetermined Split Size, and then they are reconciled (once all individual functions have finished) to provide the output of this custom node. The final part of the flow (post-custom Split-Join node) is dedicated to data acquisition, where metrics are stored in a CSV-formatted file. One critical consideration from the asynchronous nature of Node-RED is that we need to check the progress of each iteration, without proceeding to the next before the previous is complete. For this reason a mutex-like operation is implemented in order to ensure this isolation between iterations.

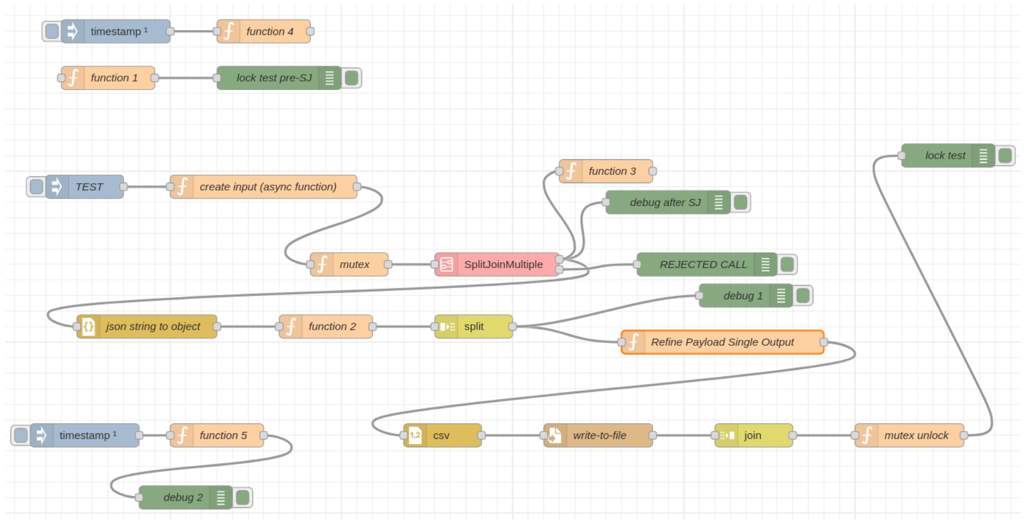


Figure 79: Split Join Experiment Automation Flow

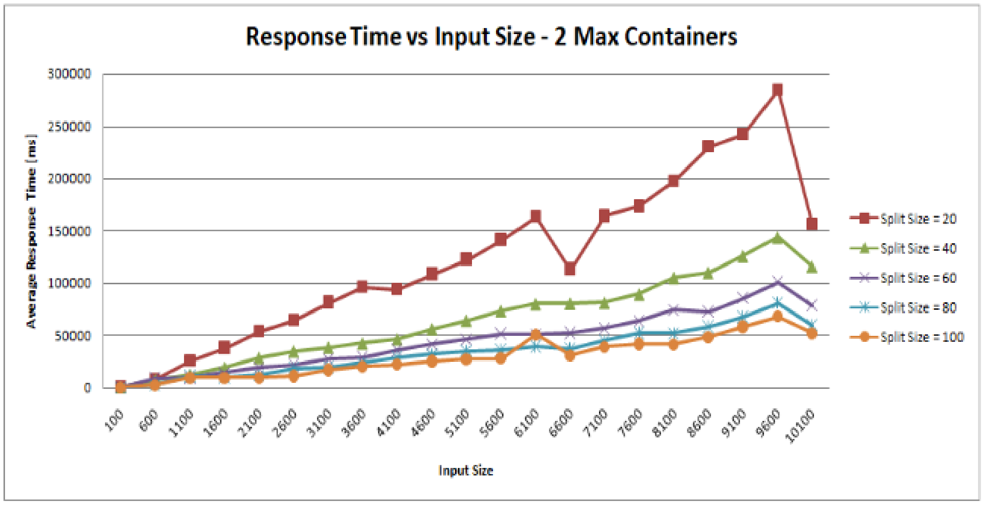
3.3.7.4. Experimental Results

In our study, we utilized the dataset that was extracted to conduct a graph analysis of the experiment, with a specific focus on Performance and Cost for the main cases (low, medium and max numbers of containers). To establish a Cost-Performance model, we considered the crucial factors of Cost calculation, Response Time, and Input Size, particularly for varying Split Sizes. The Cost equation used in our analysis is derived from the AWS Lambda Pricing model, specifically for the East Ohio Pricing region and is defined as follows:

Cost = 0.0000000021*2*(average duration in milliseconds)*(container number) + ((0.20 $ per million requests) * container number)

where Container Number = Input Size / Split Size

The results appear in the following figures, for 2, 8 and 32 containers from a response time and cost perspective.



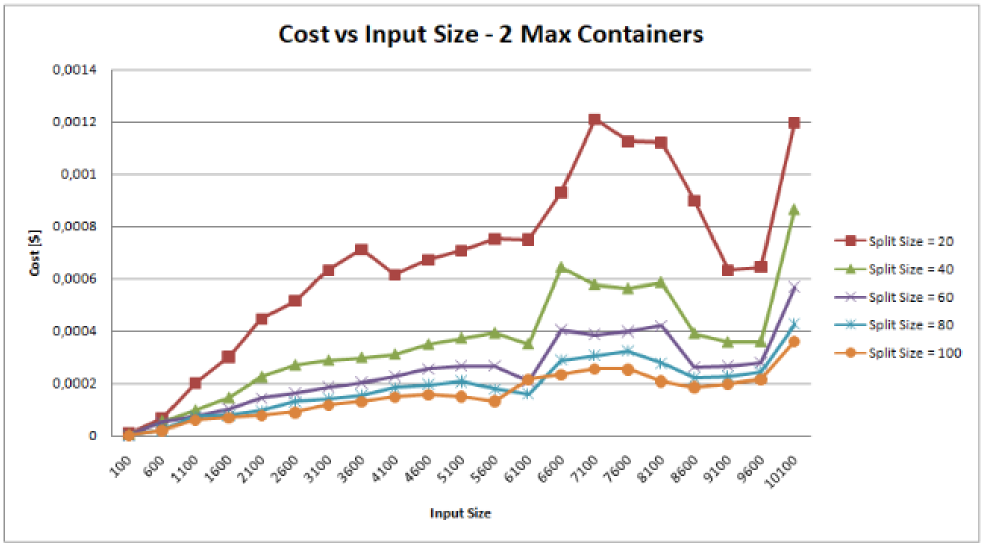
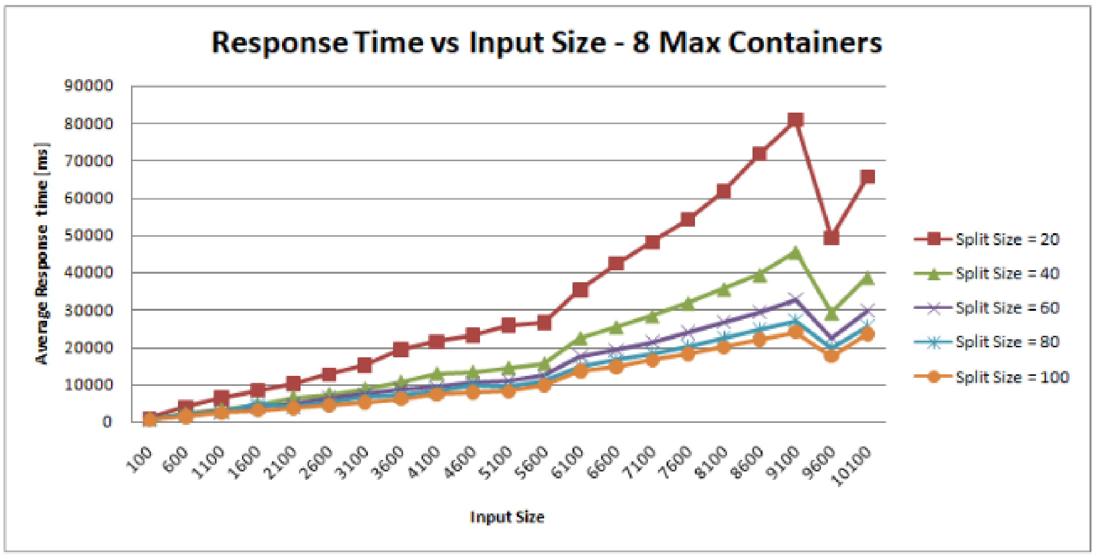


Figure 80: Average Response Time and Cost compared to Input Size for 2 Max Containers



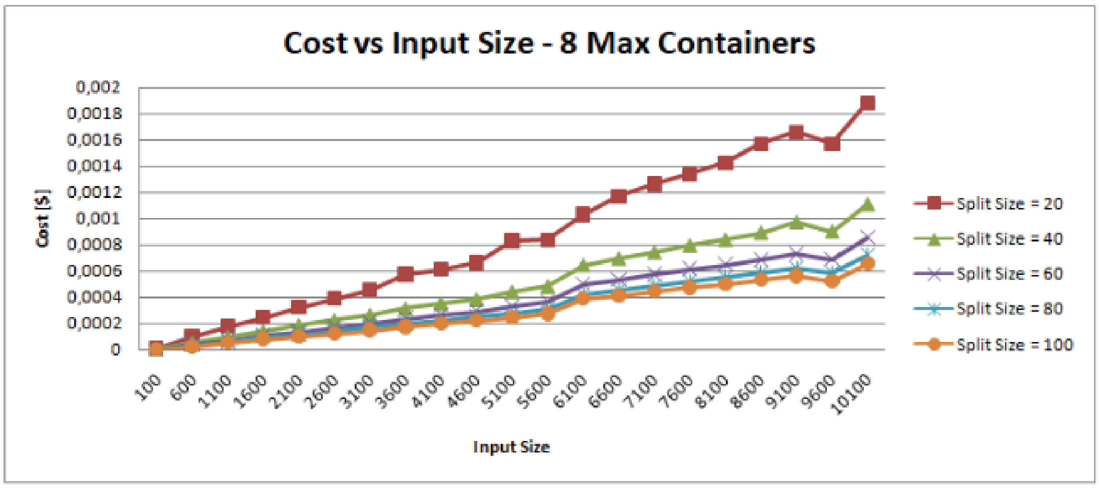
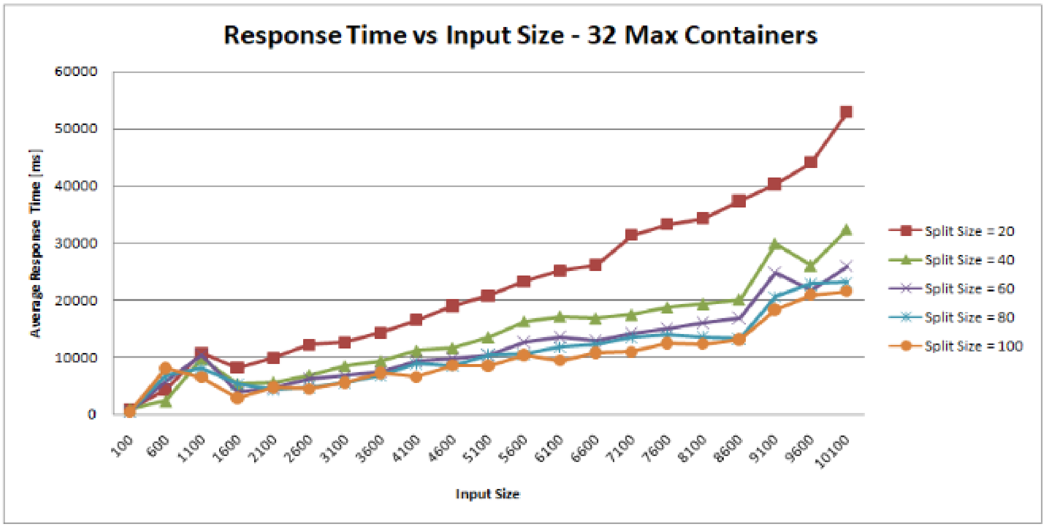


Figure 81: Average Response Time and Cost compared to Input Size for 8 Max Containers



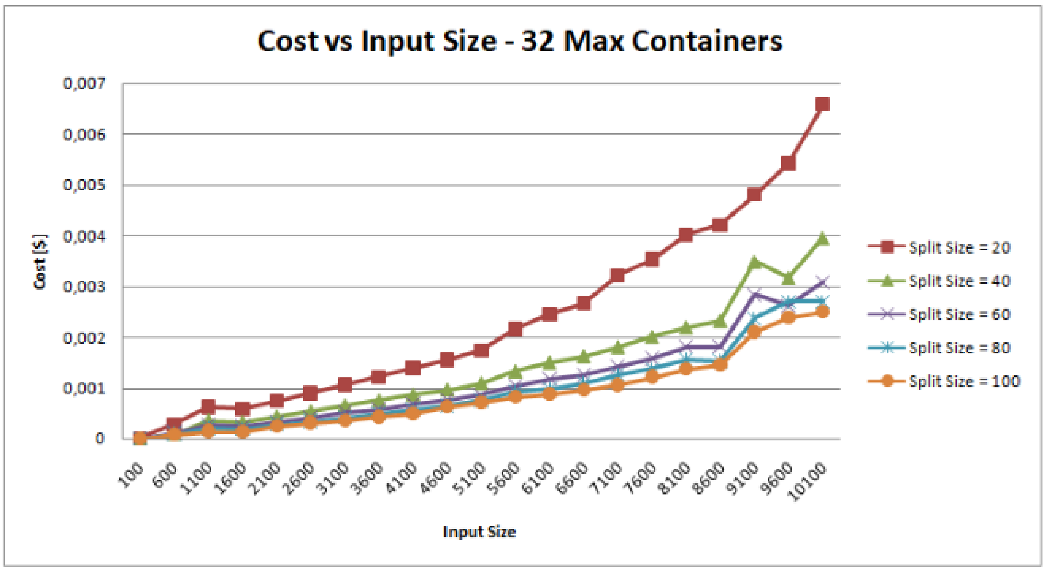


Figure 82: Average Response Time and Cost compared to Input Size for 32 Max Containers

3.3.7.5 Discussion on SJ experimentation results

From the analysis of the above graphs we can extract several useful conclusions that can help the developer to dictate the split size of the SJ pattern. There is a clear pattern that a high split size benefits performance (in the form of the total response time needed), especially as the number of inputs grows. By changing the max containers allowed as well as the ratio of input/split size, we can also observe trade-offs between the wait time and the execution time, the main elements from which the total response time comprises. In terms of the max containers allowed, we can see the benefit from trading wait time (in the case of 2 max containers with a total response from 50 to 150 seconds in the highest input case) with execution time with concurrent requests (in the case of 8 and 32 max containers with ranges from 20-70 and 20-55 seconds respectively).

However this comes also at an increased cost, since the concurrency overheads mean that a higher duration of execution is needed (although the total response time is smaller due to the lower wait time), raising the cost from 0.0012 in the 2 max containers (for the higher split size) to 0.0019 and 0.0068 $ for 8 and 32 max containers respectively. Specifically in the cases for 8 and 32 max containers, the latter seems to have a greater performance benefit, in contrast with the 8 max container case where the cost is strictly less but performance significantly worse.

By comparing the different split sizes for the same max containers, we can conclude that in all cases of the highest inputs the effect seems similar, i.e. the total response time is approximately 3 times lower when selecting a high split size. In the context of cost analysis, it is crucial to consider the relation between the maximum number of containers and the associated cost, while selecting a high split size can lead to higher cost outputs but with a lower reduction rate at each max container case. This can be attributed to the fact that the cost calculation equation relies on the “number of active containers “ value, which in high split cases is greater, meaning we have less waiting time.

4. Global Continuum Patterns Placement

This section focuses on the description of the global continuum placement component and related tools that provide the intelligent scheduling of applications upon a highly distributed computing infrastructure. The specific component proposes adapted task placement mechanisms to perform the resource allocation and deployment schema for each application selecting resources across a distributed environment composed of different types of platforms in an optimal and timely manner. It plays the role of a meta-scheduler of application workflows across the global continuum, which can be composed of independent, self-managed clusters of different types such as public clouds, on-premises and edge systems. Each application is considered to be structured as a workflow of independent tasks which have their own resources and constraints requirements while each of them may have the need and ability to be deployed upon a different platform from the other.

The remainder of this section describes the design specification of the component and its related tools which is followed by the implementation and integration highlights which go deeper into the architecture and the implementation of the particular component. Then we provide various experimentation and evaluation outcomes related to the usage and execution of the component and finally we discuss the next steps providing details about the current and future work.

4.1. Functional description

The Global Continuum Placement component makes the scheduling decision for the different workflows to be deployed upon the PHYSICS platform. To finalize the schedule of a workflow all its tasks must be matched to their destined resources to be executed upon. This is done by considering their needs in resources as described during the application design in conjunction with the resource availability and possible optimization insights. In particular, as shown in Figure 83 the component gets the following inputs:

- The application graph where the workflow's needs for resources and constraints are given per task while various information such as objectives are given per workflow. These details are passed from the Design Environment.
- The resource characteristics and availability for each cluster participating in the global continuum. The Reasoning Framework gives these details.
- The possible execution optimizations insights to take into account as given by the Performance Evaluation Framework.

Using the above inputs, the component needs to perform the decision making of which resources to use for each task of the workflow in a timely and optimal manner. So, as we can see in Figure 83 the output of the component is:

- The placement schema to be used as initial deployment or adaptation for an already existing execution. This schema is given to the platform orchestrator which then forwards it using the adequate form to the Open Cluster Management which eventually passes the information to the local cluster's schedulers managed by the coordinated work of OpenWhisk-Kubernetes. This is described in detail in the deliverable 5.2.

Internally the Global Continuum Placement has to provide the adapted heuristics in order to perform optimal and rapid scheduling decisions. An important parameter of the application graph to be used to optimize the execution of a workflow is the list of objectives. Through this parameter, the user can define which metrics are important for the particular workflow such as performance, cost, availability or energy improvements. Hence, a workflow can select one or multiple objectives and the component will try to give a solution to the particular problem. For that, we are designing the component in such a way as to enable capabilities for multi-objective scheduling by exploring different types of algorithms.

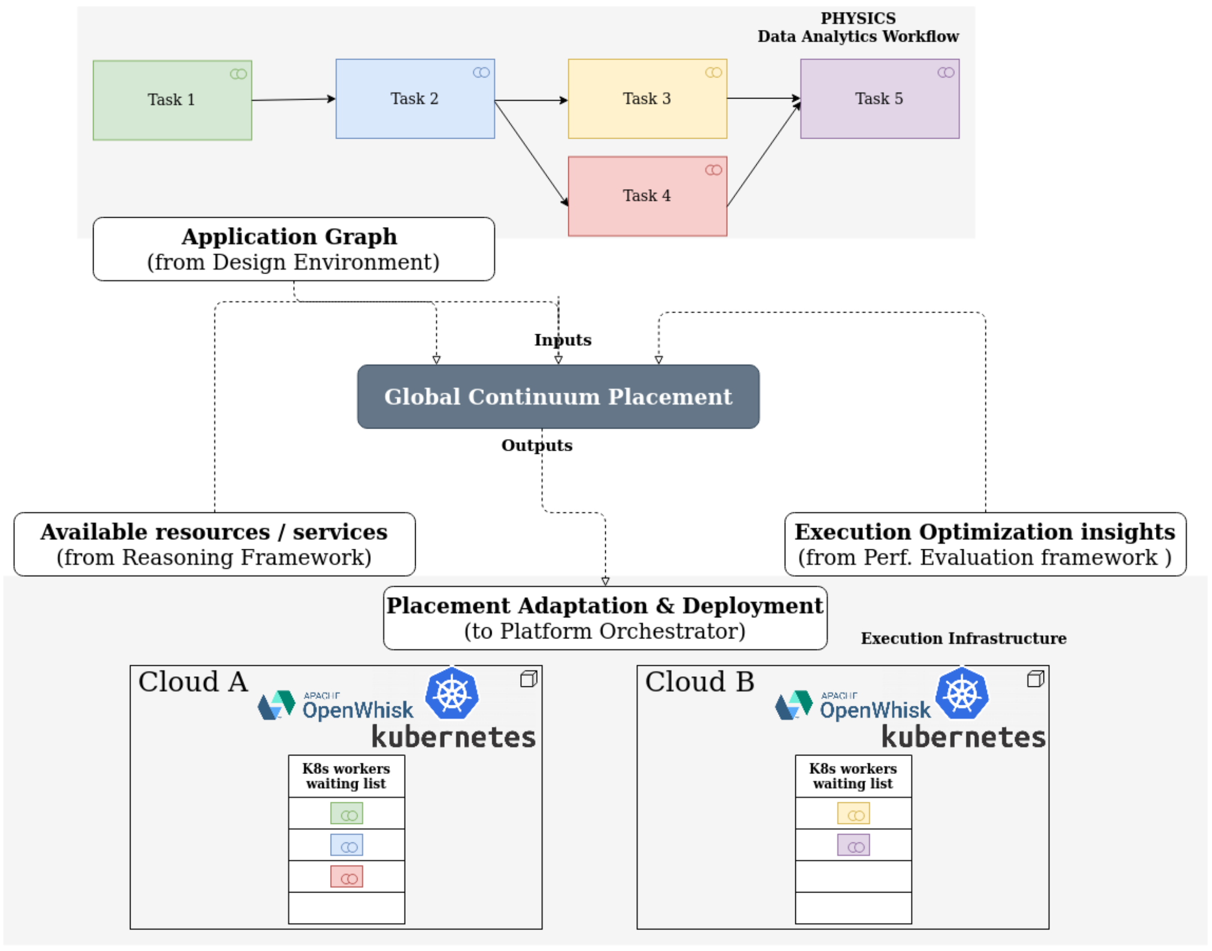

PHYSICS
Data Analytics Workflow
Task 1
Task 2
Task 3
Task 4
Task 5
Application Graph
(from Design Environment)
Inputs
Global Continuum Placement
Outputs
Available resources / services
(from Reasoning Framework)
Execution Optimization insights
(from Perf. Evaluation framework)
Placement Adaptation & Deployment
(to Platform Orchestrator)
Execution Infrastructure
Cloud A
OpenWhisk
kubernetes
K8s workers waiting list
Cloud B
OpenWhisk
kubernetes
K8s workers waiting list

Figure 83: Global Continuum Placement high-level view

The study of complex scheduling algorithms in such a distributed environment needed specialized tools for fast experimentation and evaluation of new algorithms under various conditions. For that, an important part of the Global Continuum Placement list of side tools is the Scheduling Simulator. This allowed us to rapidly evaluate different variations of new scheduling algorithms and try to adequately adapt them to fit the needs of each particular context. A slight modification of some parameters of an algorithm may play a significant role in the performance of the scheduler for particular workloads and specific global continuums. The simulator helps us trace the impact of various parameters on different metrics. Furthermore, the combination of an experimental methodology based on simulation and real-scale evaluation, whenever possible, allowed us to better understand the behavior of the placement algorithms under various contexts while validating the effectiveness of the simulator. In particular, large-scale scalability evaluation is not possible without a simulator but having confirmed that the simulator shows correct results when comparing it with the real-scale experiments in smaller scales allows us to trust the simulator results for the large-scale scalability experiments.

4.2. Technical description

4.2.1. Generic component and first best fit algorithm with multi-objective capabilities

The Global Continuum Placement Component architecture is shown in Figure 84 and is composed of the following principal parts:

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

In parallel, two additional external subcomponents are provided:

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

The communication with other software such as to pull and push the inputs and outputs related to the component is performed through a REST-API.

The implementation of the Global Continuum Placement component is done using Python programming language. Furthermore, we make use of NIX functional package manager to prepare the packaged containerized environment to perform the necessary CI/CD pipelines. The whole software can be packaged and installed as a Docker container.

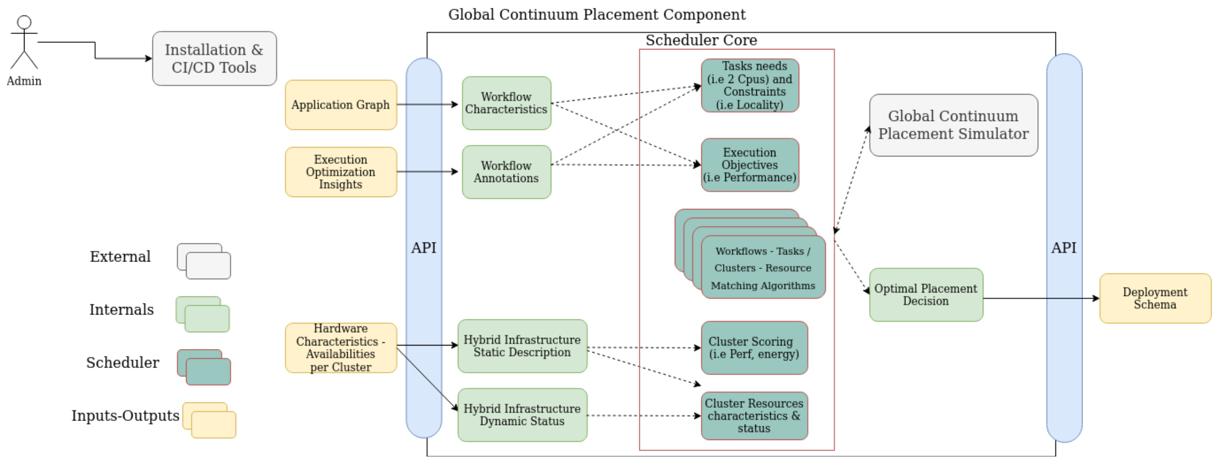

Global Continuum Placement Component
Scheduler Core
Admin
Installation & CI/CD Tools
Application Graph
Execution Optimization Insights
API
Workflow Characteristics
Workflow Annotations
Tasks needs (i.e 2 Cpus) and Constraints (i.e Locality)
Execution Objectives (i.e Performance)
Workflows - Tasks / Clusters - Resource Matching Algorithms
Global Continuum Placement Simulator
Optimal Placement Decision
API
Deployment Schema
External
Internals
Scheduler
Inputs-Outputs
Hardware Characteristics - Availabilities per Cluster
Hybrid Infrastructure Static Description
Hybrid Infrastructure Dynamic Status
Cluster Scoring (i.e Perf, energy)
Cluster Resources characteristics & status

Figure 84: Global Continuum Placement component internal architecture

Based on the current architecture of Physics the Global Continuum Placement (GCP) component does not have to make the final decision of the exact resources where the execution of a task needs to take place. This is done by the local cluster Scheduling Algorithms which lies within the combination of OpenWhisk-Kubernetes schedulers. Furthermore, it does not communicate directly with the local schedulers to propose the deployment schema. It forwards the deployment schema to the Orchestrator which will then communicate with each local-cluster scheduler. It is also the Orchestrator that will manage any possible adaptations that may be needed. For example, if a reconfiguration of the schedule is needed the Orchestrator will demand to the Global Continuum Placement to perform a new schedule taking into account the modifications. This is an important aspect of the internal architecture of GCP because we are not obliged to keep either an internal state of the status of the resources availability, nor how the final deployment has been finally made. GCP can be a stateless service that can continuously get different inputs and based on them perform an adapted scheduling decision. Of course, this may change in the upcoming versions of this component, but the initial version is implemented as a stateless service without an internal database.

The principal part of the intelligence of the Global Continuum Placement Component lies within the Scheduler Core. The Scheduler Core decomposes the collected inputs into meaningful parameters that will take part in the scheduling decision.

In particular, in the first implementation of GCP we take into account the following parameters. From the application perspective, we get the following:

- Type of clusters to be used such as Edge, Cloud, HPC, on premise. This is a task related parameter and is handled as a constraint. Multiple values of sites are resolved with a logical OR.
- Type of architecture such as x86_64 or ARM64. This is a task related parameter and is handled as a constraint.
- Computing resources needs such as the number of CPUs, amount of memory, and number of GPUs. This is a task related parameter.
- Type of execution objective such as Energy, Performance, and Availability along with a particular value among “high”, “medium”, “low” to define the importance for each one of them. This is a workflow related parameter.

From the cluster site perspective, we create a platform map of our continuum where each cluster provides the following parameters:

- Type of clusters such as Edge, Cloud, HPC, on premise. Only one is valid per cluster.
- Type of architecture such as x86_64 or ARM64. Only one is valid per cluster. This is currently implemented per cluster site but will be modified in an upcoming version to reflect different groups of the same cluster.
- Total Number of computing resources, amount of memory per node and total amount of GPUs. This is the currently implemented list of resources, but it will be updated in future versions to allow the declaration of additional resources such as Network, IO, etc.
- Objective scoring for all the available objectives. Currently, the following objectives are implemented: Energy, Performance and Availability. The score is done by providing a value from 0 to 100 for each objective. The availability here is directly related to the uptime of each cluster. This is currently implemented per cluster, but it can be updated to reflect groups of clusters in future versions. The scores are calculated by the Reasoning Framework and may be combined with data coming from the Performance Evaluation component. As an example of objective scores: A score of 90 in Performance, 30 in Energy and 60 in Availability means that the cluster has a particular focus on Performance, provides a decent amount of Availability and is not very energy efficient.

Based on the above details the scheduler core will offer different policies that will try to provide solutions to the workflow-tasks VS. cluster-resources matching problem.

The scheduler resolves the problem for each task of the workflow in the following order:

1. Resolve the site constraints of the task by filtering out the sites that do not fit

2. Resolve the architecture constraints of the task by filtering out the clusters that do not fit

3. Calculate the scores per objective for the workflow

4. Sort out the (filtered-out) sites with the highest score first
5. Apply a particular policy to match the tasks to the available resources based on their needs. Initially, the first policy that was implemented at this level was a FirstFit policy which places the task on the first available resource that fulfills the constraints, has the highest score in objectives and covers the resource needs.

The objective scoring heuristic is an interesting aspect of the scheduler which can seamlessly allow us to explore single and multi-objective scheduling policies. As mentioned, we have currently implemented only a simple FirstFit policy which is the fastest way to deal with multiple objectives in a scheduling problem. Nevertheless, the FirstFit policy, besides the advantage of being fast does not provide an optimal or near to optimal solution. However, this particular objective scoring heuristic will allow us to study in a pragmatic way more complex policies based on MINLP or genetic algorithms that consider multiple objectives while trying to return optimal or sub-optimal results in a less timely manner.

The integration with the different components that affect the decision making of the Global Continuum Placement along with those that receive the final scheduling proposition is done through a REST-API and specific JSON files that get augmented when passing from the different components. For example, the Design Environment component compiles a particular JSON file which provides the workflow-tasks needs in terms of objectives, resources and constraints. The Performance Evaluation framework can provide some further annotations related to the constraints or objectives of the workflows. In parallel, the Reasoning Framework will bring the sites' static resources characteristics, dynamic availabilities and scoring per objective in a separate JSON file. Finally, exchanges between the Global Continuum Placement and the Orchestrator will be done on the one side to allow the scheduling decision to be forwarded to the local clusters schedulers but also to enable readaptations based on possible reconfigurations as triggered by the Orchestrator.

This integration with the different interconnecting components has been performed and the necessary data are exchanged successfully to perform end-to-end executions.

The implementation of the installation and CI/CD tools have been also performed based on the characteristics we mentioned initially but further changes may be done to adapt to the particularities of PHYSICS installation procedures.

4.2.2. Optimization algorithms

Inspired by the dual-approximation algorithm of Shmoys and Tardos [48], we have implemented FOA (Function Orchestration Algorithm) a scheduling algorithm that enables the allocation of batches of serverless functions on serverless-based heterogeneous edge-cloud platforms. FOA is a multi-objective algorithm, specifically conceived for serverless based systems on the edge-cloud continuum, minimizing both the makespan and the data movement (the sum of the amount of data transferred to deploy containers and the amount of data transferred by functions I/O). This section details this scheduling policy, its algorithm, and its two main components: the linear program and the integral matching process. In addition, we detail the container layer download optimization model used.

It works under the following assumptions:

a) functions depend on environments and all dependencies are known;
b) functions and environments are independent between themselves and known in advance;

c) their cost and processing time on each machine are known.

And it expects the following inputs:

1. environment execution time: the time to download and deploy functions environments (containers), in seconds (s);
2. environment cost: the amount of downloaded data per environment, in megabytes (MB);
3. function execution time: the execution time of serverless functions, in seconds;
4. function cost: the amount of data transferred as functions' I/O, in megabytes.

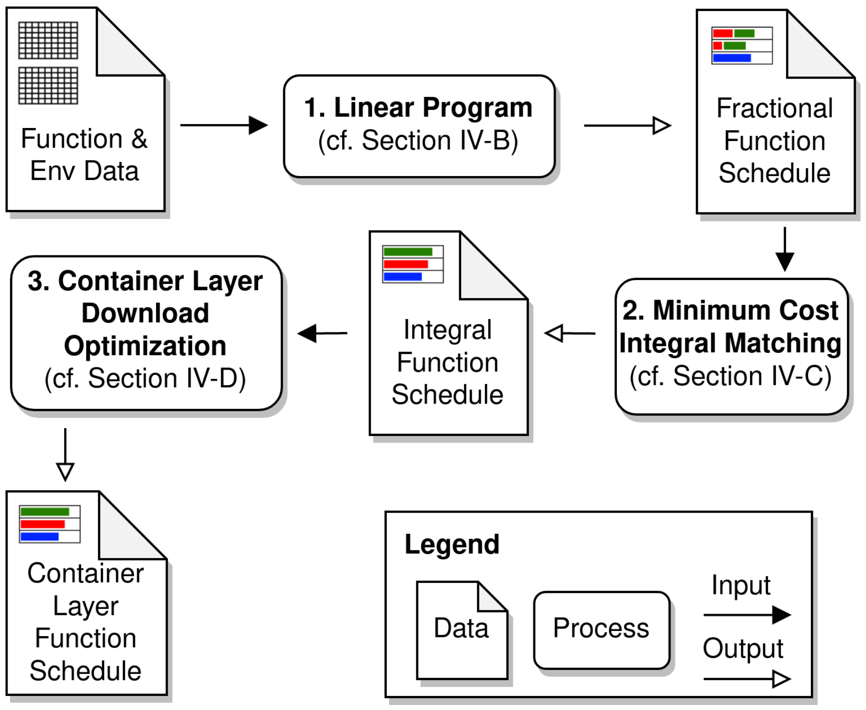

Function & Env Data
1. Linear Program (cf. Section IV-B)
Fractional Function Schedule
2. Minimum Cost Integral Matching (cf. Section IV-C)
Integral Function Schedule
3. Container Layer Download Optimization (cf. Section IV-D)
Container Layer Function Schedule
Legend
Data
Process
Input
Output

Figure 85: FOA's algorithm step by step

Figure 85 illustrates all steps performed by FOA. Step 1. Linear Program (detailed in Section IV-B) uses function and environment data to produce a fractional function schedule (to get the results fast enough). However, we need an integral solution for allocating functions in our context. Then, step 2. Minimum Cost Integral Matching converts the fractional function schedule into an integral function schedule. Finally, step 3. Container Layer Download Optimization (in Section IV-C) models the sharing of container layers done by Docker in Kubernetes, for each function that arrives in the machines. It reduces the amount of data downloaded and execution time based on the cache state of the machines. Our algorithm optimizes the cost of the entire workload, under a constraint on the makespan, of an arbitrary value T. To optimize the makespan, we re-execute steps 1 and 2 of FOA up a given number of times until we reach the expected precision, before starting step 3. These repetitions update the makespan constraint T at each iteration by a binary search process. The first iteration of the algorithm provides a solution with the best cost because the makespan constraint is fully relaxed. As far as the makespan decreases through the binary search process, the cost increases. Empirically, eight iterations are sufficient.

The details of the algorithm have been extensively explained along with the methodology and experimental evaluations in an article [49] that has been submitted and published in

CCGrid2023. Hence we omit these details here, but the reader may refer to this article for all the information.

FOA has been implemented and validated under simulated experiments and it is currently being integrated as a new algorithm within the main Global Continuum Placement component in the PHYSICS testbed to be used by FaaS applications and the PHYSICS use cases.

Following the same path with the above scheduling algorithm we have also added one more objective to be taken into account and this is the one of energy consumption named FOA-E. In this context in the global continuum, we use a multi-objective scheduling policy to get the best from the heterogeneous local clusters connected to the continuum to minimize the energy consumption along with the makespan of the batches of serverless functions that arrive on the platform. The algorithm has been developed and we are currently performing experimentations to validate it, preparing a new publication to be submitted and integrating it in the PHYSICS testbed.

4.3. Demonstration and experimentation

4.3.1 Demonstration of the implemented algorithm

The first version of the global continuum placement component allows us to experiment with different workflows deployed upon various combinations of platforms. A simple example is shown below. The following JSON file shows the way a hybrid distributed infrastructure composed of 3 different sites with different hardware characteristics, possible architectures and objectives scores, can be described:

Code 8: Code snippet showing platform information.

```
{
  "platform": {
    "cluster1": {
      "type": "Edge",
      "resources": {"nb_cpu": 4, "nb_gpu": 0, "memory_in_MB": 1024},
      "architecture": "x86_64",
      "objective_scores": {"Energy": 60, "Availability": 5, "Performance": 25}
    },
    "cluster2": {
      "type": "Edge",
      "resources": {"nb_cpu": 2, "nb_gpu": 1, "memory_in_MB": 4096},
      "architecture": "arm64",
      "objective_scores": {"Energy": 100, "Availability": 30, "Performance": 50}
    },
    "cluster3": {
      "type": "HPC",
      "resources": {"nb_cpu": 1000, "nb_gpu": 50, "memory_in_MB": 16e6},
      "objective_scores": {"Energy": 10, "Availability": 80, "Performance": 100}
    }
}}
```

Once we have described our infrastructure with the above json file we can register this platform description using a REST API:

Code 9: Code snippet to register the platform information using an API call

```
curl -X POST -H "Content-Type: application/json" -d @test-platform.json
http://127.0.0.1:8080/init
```

Then we can create a workflow to be launched for execution using the following json file test-workload.json which describes a workflow composed of 4 tasks based on particular dependencies among them, with different demands in resources and constraints and with specific workflow objectives.

Code 10: Code snippet showing demands and constraints per task

```
{
  "id": "19fe4293742e0b2c",
  "displayName": "Full example",
  "type": "Flow",
  "executorMode": "NativeSequence",
  "native": true,
  "objectives": {
        "Energy": "high",
        "Availability": "low"
  },
  "flows": [
        {
        "flowID": "flow1",
        "functions": [
        {
        "id": "function1",
        "sequence": 1,
        "allocations": [
        "cluster3"
        ]
        },
        {
        "id": "function2",
        "sequence": 2,
        "annotations": {
        "cores": "2"
        }
        },
        {
        "id": "function3",
        "sequence": 3,
        "annotations": {
        "cores": "2"
        }
        },
        {
        "id": "function4",
        "sequence": 4,
        "annotations": {
        "cores": "2",
        "architecture": "arm64"
        }
        },
        {
        "id": "function5",
        "sequence": 5,
        "annotations": {
        "cores": 2,
        "memory": 1000
        }
        }
        ]
        },
        {
        "flowID": "flow2",
        "executorMode": "NoderedFunction",
        "annotations": {"core": 1, "memory": 1000},
        "functions": [
        {
        "id": "excluded-func",
        "annotations": { }
        }
      ]
    }
  ]
}
```

Then, we can ask the scheduler to perform the placement of the workflow upon the defined global continuum registered previously. For that, we can use the following REST API:

Code 11: Code snippet to realize the placement by the global continuum engine.

```
curl -H "Content-Type: application/json" -d @test-workload.json
http://127.0.0.1:8080/schedule
```

and we will get the following result using the default FirstFit scheduler:

Code 12: Code snippet with output of the result from the locations.

```
[
  {
    "flowID": "1234",
    "allocations": [
      {"cluster": "cluster3", "resource_id": "function1"},
      {"cluster": "cluster1", "resource_id": "function2"},
      {"cluster": "cluster1", "resource_id": "function3"},
      {"cluster": "cluster2", "resource_id": "function4"},
      {"cluster": "cluster3", "resource_id": "function5"}
    ]
  },
  {
    "flowID": "flow2",
    "allocations": [
      {"cluster": "cluster1", "resource_id": "flow2"}]
  }
]
```

Here is the explanation of these decisions:

- `function1` has an explicit cluster constraint for the `cluster3` so it is allocated there.
- `function2` only requires 2 CPU and all clusters have at least 2 CPU. The architecture constraint is not defined but by default it is x86_64, so only the cluster 1 and 3 can fit the constraint. The scheduler then takes into account the objectives and favors the Energy and the Availability so it chooses the `cluster1`.
- `function3` has the same constraints as `function2` so it goes on the same cluster, the `cluster1` because it still has 2 CPU available.
- `function4` goes on `cluster2` because it requires an `arm64` architecture and only `cluster2` is providing it.
- `function5` has only resources constraint and should go to the cluster1 regarding the objectives but it does not have enough resources. It is finally allocated to `cluster3` which is the only one that fits the constraints and has available resources.
- `flow1`: because it is a NoderedFunction, this flow is scheduled at the flow level. It is scheduled on the `cluster1` because it has enough resources.

This also works at the flow level with the same annotations.

As described in the previous section the scheduling is done task by task in the dependency order of the workflow. For each task we apply filters to remove sites that do not fit the site and architecture constraints. Then, we apply a scoring function based on the objective scores of the sites and the objective levels of the workflow. Finally, we use a first fit policy on the sorted by highest score and allocate to the first site in the list that has enough resources.

4.3.2. Simulation-based experimentation and evaluation of the optimized multi-objective algorithms

We evaluate FOA by comparing it with a baseline policy inspired by the Kubernetes scheduling policy. More specifically, we compare it with the image locality plugin. To conduct such experiments, we performed simulations on top of Batsim/Simgrid simulation tools.

We present in this section how we model our workloads, platforms, and the baseline policy. We also detail the simulated environment that makes possible the study of these scheduling policies, and the grid of experiments conducted.

We perform bare-metal executions of functions that we adapted from the FunctionBench benchmark, such as matrix multiplication, linpack, chameleon, modeling training, and image and video processing.

We evaluate 9 functions with different inputs. In total, we have 19 combinations of functions and inputs. Each function requires a different container, being 9 in total. The container sizes vary from 170 to 2560 MB. All of them are available on a public repository on DockerHub. We deployed the serverless platform OpenWhisk on top of GRID5000 and execute our functions there.

For each function executed, we obtain its execution time and resource usage measurements (CPU, memory, bandwidth, etc.). We also instrument the functions to extract the time consumed by different phases of serverless functions such as download and deployment of containers, functions execution, I/O transferring, container deletion, etc. At last, we perform a calibration phase to estimate the number of floating-point operations (flops) necessary to execute each function, which allows Batsim and Simgrid to accurately perform the simulations.

Our workload model is based on such executions of serverless functions on serverless-based heterogeneous edge-cloud platforms. For each function, we translate the data retrieved from the bare-metal executions to the Batsim/Simgrid requirements and format. We use random seeds to randomly select different combinations of functions and inputs for each workload created. In addition, for each function and its required container, we retrieved the container layers' composition. Such a description was also attached to the workloads to reproduce the layers' sharing behavior during our simulations.

FOAs linear program resulted in fractional solutions optimizing cost and makespan. Minimizing the cost is simple by allocating all tasks of the same environment to a single fast machine. But, this solution does not minimize the makespan. Hence, optimizing cost and makespan is a compromise.

Figure 86 illustrates this compromise through FOA's binary search process for all combinations of workload size and platform size. The heterogeneity level is not distinguished because the three levels show similar behaviors. The x-axis is the makespan (in minutes), and the y-axis is the cost (the amount of downloaded data in GB). The colors represent the iterations of FOA's binary search process.

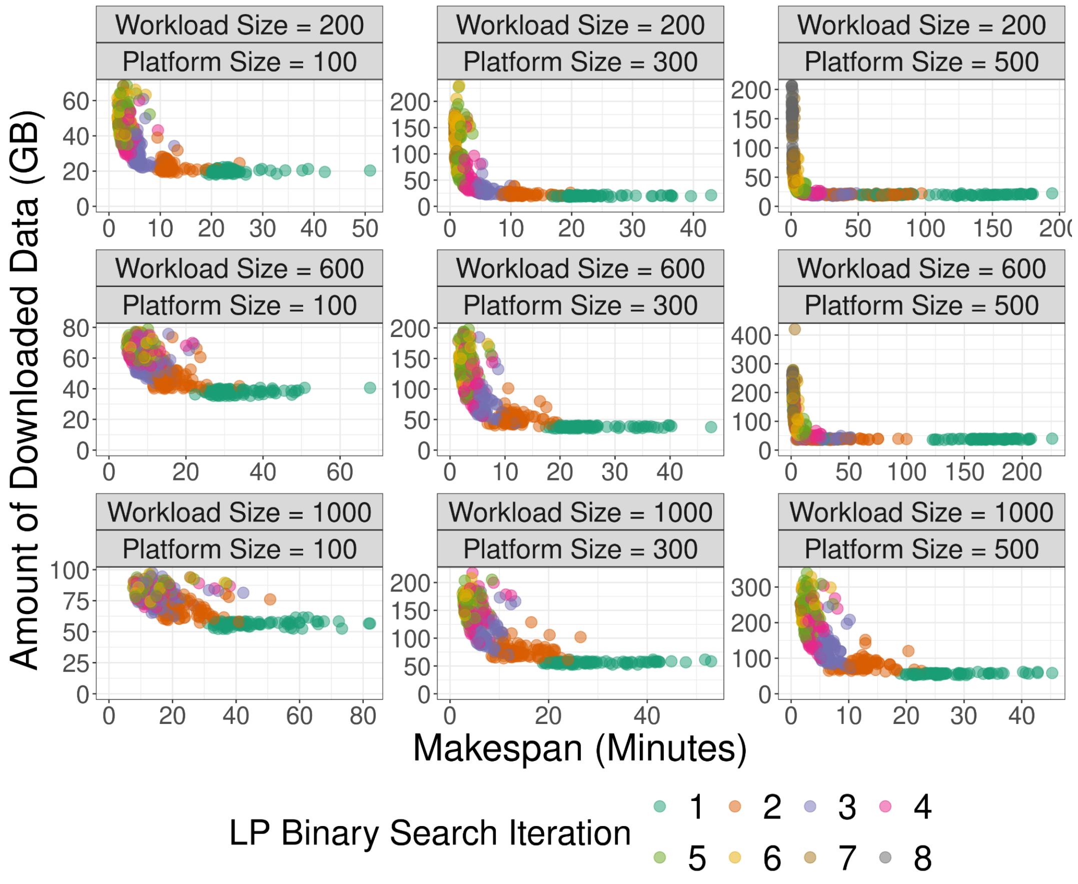


Figure 86: FOA's linear program trade-off between makespan (x-axis) and amount of downloaded data (y-axis). The color of the points represents the iteration-id of the linear program binary search. Each one is one solution of the linear program.

The first iteration computes the smallest cost and hence the highest relevant makespan. As far as we constrain the makespan over the iterations, the cost increases. We highlight that

I. the 3rd iteration looks to be a good trade-off between both objectives,
II. the Pareto's shape is quite smooth in relevant scenarios for us.

In addition, we remark that (a) not all instances found a solution of cost with small values for makespan, and (b) since we need to re-execute the linear program at each iteration, the processing time to produce a final solution is cumulative. % and as we increase the constraint of makespan at each iteration, the processing time of the linear program increases as well; so increasing the number of iterations, increases the overall computation of the solutions. We call the time that a scheduling policy takes to decide the allocation of the functions as processing time. FOA is based on a linear program with several constraints, which is expensive in terms of processing time. Unlike our approach, the Imagelocality baseline has a greedy algorithm and does not have a global view of the platform during the scheduling phase. Hence, Iimagelocality produces a non-optimal allocation decision within a second while FOA reaches a better solution within minutes. Our scheduling policy takes a median of 2.69 minutes to produce a decision, while our baseline performed in the median of 0.6 seconds. However, FOA's decision considered the eight repetitions of its binary search process. Each repetition performs with a median of 0.34 minutes. Considering that, and as mentioned above, our results show that three repetitions are enough for good results, then FOA may produce them with a median of 1.02 minutes. It is still not as fast asImagelocality, but it is reasonable to evaluate the gap between both scheduling policies.

We also remark that in our experiments, we use an open-source solver, CBC, through a library to compute our linear program. Using a commercial solver would reduce the processing time but also would hinder the system administration simplicity of our computations. Various other experiments have been performed and can be found in the related publication [49].

We conclude that even with light levels of heterogeneity, the gains of FOA are important in serverless computing at the edge-cloud continuum, and we believe that with a more accurate model of heterogeneity, these gains may increase. By reducing the amount of data transferred to download containers, we minimize cold start delays through faster container deployments. In addition, it also speeds up the functions' execution time. We emphasize that the results are even better in the scenarios important for us, where there are not many functions per machine. Thus, efficient scheduling policies for serverless computing at the edge-cloud continuum require better management of the heterogeneity to drastically improve the amount of data downloaded and the system utilization.

4.4. Discussion

The Global Continuum Placement component has been implemented as a PHYSICS component and this section provides a thorough description of its principal functions, its design specification, its implementation and integration highlights along with various validation and experimentation results. We have performed a tight integration with the different components that GCP exchanges information with and we are still considering adaptations to better fit the needs of these tight integrations.

Furthermore, we have studied, implemented and evaluated the FOA multi-objective algorithm based on Linear Programming which has been the center of one publication in the international peer-reviewed conference CCGrid2023. The algorithm developed in this article is currently being enhanced to enable the usage of one more objective, that of energy consumption (FOA-E) and we are planning to finalize one more publication describing our achievements regarding these aspects as well, very shortly, before the end of the PHYSICS project.

Some other ongoing work is related to the integration and deployment of the FOA multi-objective algorithm in the GCP component which is currently being finalized, along with some specific integration improvements which are currently taking place. In particular, the consideration of tasks' profiles based on previous executions as forwarded by the Performance Evaluation Framework is an interesting and important optimization to include that will help select more adapted resources for the optimal placement of each particular task.

5. Distributed In-memory State Services for Data Interplay

This section presents the Distributed In-Memory State Service (DMS) component developed under the task T4.4, Distributed In-memory State Services for Data Interplay, its internal architecture, the status at month 30 of the PHYSICS project. Finally, a performance evaluation of the final version of the DMS is described.

5.1. Functional Description

Functions in the FaaS paradigm are stateless. Functions can receive parameters but the size of the parameters is limited. For instance, the maximum size of the parameters per function in OpenWhisk is limited to 5MB. Many functions produce large results that are the input of other functions. This is the case of analytics tasks that produce intermediate results that are consumed by other tasks (for instance, in map-reduce processes). Mapping this type of application to the FaaS model incurs a high overhead since any state that needs to be kept or shared with another function must be stored in an external remote storage unit (Amazon S3, or database like Redis). This incurs high overhead in terms of latency. The DMS component goal is to implement an efficient in-memory data service for sharing large objects between functions.

5.1.1 Main Objective and Functionalities

The main objective of this DMS component is to provide an in-memory distributed data store for sharing data between functions. The DMS should be able to scale up and down automatically taking into account the amount of data to be shared among functions, saving platform usage costs.

5.1.2 Relationship with Other Components

The DMS is an independent component that can be used in any FaaS framework. The DMS is used by functions (defined in WP3) and deployed in a cluster (managed in WP5) where the functions are executed. The DMS is used as a library in a function with both Python, Java and C++ interfaces.

5.1.3 Requirements Matrix

The DMS component is involved in meeting the following requirements defined in D2.3 – State of the Art Analysis and Requirements Definition V2 delivered by month 21 of the project.

- Req-4.4-state: "The distributed management system must maintain the state between function invocations."
- Req-4.4-interplay: "Enable numerous functionalities of interplay between the in-memory state and the persistent storage layer."
- Req-4.4-tradeOffs: "Investigate trade-offs between consistency and performance."
- Req-4.4-perf: "Dynamic caching mechanisms and persistent storage functionalities and activation performance."
- Req-4.4-access: "Different access patterns must be investigated and included such as single client-multiple access, multiple client-single access, multiple client-multiple access."
- Req-4.4-elasticity: "Implement the distributed data management service elasticity for both reducing and expanding the exploitation resource plane available by the application."
- Req-4.5-persStorage: "FaaS platform should provide persistent storage(Like AWS S3 or

Minio) for applications that also supports notifications based on the content changes."

5.2. Technical Description

The DMS component has been reimplemented since September 2022. The first version of this component was based on Pocket[8]. This research prototype was based on Apache Crail[9] project. However, this project has not been maintained since July 2022. The new DMS service implementation is now based on KeyDB[10]. KeyDB is a multithreaded fork of the in-memory Redis[11] database. KeyDB can run on several cores taking advantage of multi-core computers. KeyDB provides persistence, replication and security.

The DMS provides a set of OpenWhisk actions (functions) for accessing the in-memory datastore (KeyDB). The DMS is able to automatically scale up and down based on the load and access patterns.

5.2.1 Interfaces and Integration

Application developers are provided with two functions for reading and writing data in the DMS. The next example shows a sequence with two functions (actions in OpenWhisk) that use the DMS.

The OpenWhisk function that writes data (Code 13) receives as parameter the information to be stored as a string and a key to identify the item. Internally this function imports the KeyDB library (line 1), establishes the connection with the DMS (line 8), and writes the key and the value provided (line 9). Finally, the connection is closed and the host, port, db and key parameters are returned so they can be used by other functions (line 11).

Code 13: functionWrite.py

```
1 from keydb import KeyDB

2 def main(event):
3    host = event['host']
4    port = event['port']
5    db = event['db']
6    key = event['key']
7    value = event['value']
8    db = KeyDB(host=host, port=port, password="", db=db)
9    db.set(key, value)
10   db.close()
11 return {"host": host, "port": port, "db": db, "key": key}
```

Another OpenWhisk function, functionRead.py (Code 14), is used for reading data from the DMS. It receives as parameters the host, the port, the database id and the key to read the data from. This function connects to the DMS (line 7), and reads the data (line 8), closes the connection to

[8] https://pocketbase.io/
[9] https://incubator.apache.org/projects/crail.html
[10] https://docs.keydb.dev/
[11] https://redis.io/

the DMS and returns a JSON with the key and value (line 10).

Code 14: functionRead.py

```
1 from keydb import KeyDB

2 def main(event):
3     host = event['host']
4     port = event['port']
5     db = event['db']
6     key = event['key']
7     db = KeyDB(host=host, port=port, password="", db=db)
8     value = db.set(key, value)
9     db.close()
10 return {"key": key, "value": value}
```

The DMS can also be accessed directly using KeyDB client as well as Python, Java and C++ interfaces. Even if developers use the OpenWhisk functions for reading and writing from the DMS they need to be aware of the limitations and parameters of the KeyDB interface. The main methods of the Python interface are shown in Table 16. By default, users connect to database 0 unless a parameter is added(0-16). The *close* method closes the connection with the KeyDB service.

The *set(key, value)* method stores the *value* in the database with the given *key*. The maximum value size is 512MB. In order to store larger data, the recommended method is to split the data into batches of 512MB and use consecutive keys; i.e.: f1-1, f1-2, f1-3,... The *get(key)* method is used for retrieving the data associated with a given key. The same restriction applies to the *set* method which writes data in KeyDB.

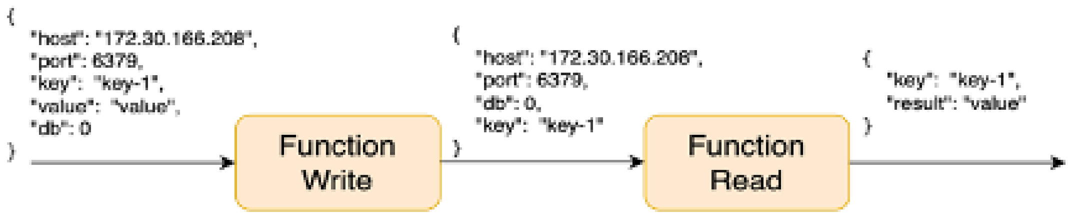
Table 16: DMS interface

Client API Function	**Description**
KeyDB(host, port, password, DB)	Open connection to KeyDB service. Password by default. Db by default 0 (0-16)
close()	Closes a connection
set(key, value)	Writes data (value) at key
get(key)	Reads the data stored at key
getset(key, value)	Sets the value at key and returns the old value stored at key
append(key, value)	If the key exists and the value is a string, the command appends the value at the end of the string. Otherwise, it is like a set.

del(key)	Delete the data and the key
keys(regex)	Show all available keys that satisfy the regular expression (regex)

5.2.2 Usage

The next example shows an OpenWhisk sequence with the two functions for reading and writing data in the DMS. This example can be found in the distribution of the DMS in the DMS-Sequence-Example folder. The first function in the sequence writes data using the DMS function, and the second one reads the data using the corresponding function.



The sequence is created in OpenWhisk using the DMS Docker image. This process can be done from the command line or using a manifest file. To simplify the presentation only the command line is included here.

Command line:

Code 15: Functions and Sequence creation

```
$ wsk -i action create dms_action_write --docker
$DOCKER_REPOSITORY/dms-python-runtimeimage:latest --main write functionWrite.py
$ wsk -i action create dms_action_read --docker
$DOCKER_REPOSITORY/dms-python-runtimeimage:latest --main read functionRead.py
$ wsk -i action create dms_sequence_write_read --sequence dms_action_write,
dms_action_read
```

The result of executing the sequence shows the key and value written in the DMS:

Code 16: Sequence invocation results

```
DMS-Sequence-Example$ wsk -i action invoke dms_sequence_write_read -p host
'172.30.166.208' -p port 6379 -p key p4 -p db 0 -p value 'This is an example of
the usage of the DMS component' --result
{
       "key": "p4",
       "result": "This is an example of the usage of the DMS component"
}
```

5.3. Experimentation Outcomes

Different DMS experimentations have been performed both deploying the DMS (KeyDB) as a service and in a Kubernetes cluster in order to benchmark the performance of KeyDB in different scenarios.

5.3.1 KeyDB as a Service

The main objective of this experiment is to validate the results of the evaluation carried out by the KeyDB team [50] using different hardware. This evaluation compares the performance of KeyDB with Redis using the Memtier[12] benchmark provided by RedisLabs. Memtier is a command line utility created for load generation and benchmarking key-value databases. They used one m5.4xlarge ec2 instance (16 virtual CPUs and 64 GiBs RAM). Memtier benchmark was configured with data sized from 8 bytes up to 16 KB. Redis was deployed using the default configuration and KeyDB was configured with 7 server-threads and enabling server-thread-affinity. The benchmark runs in the same node.

Their performance evaluation results [50] show that KeyDB is 5X faster than Redis, it performs 5 times more operations per second (ops/sec) than a single instance of Redis with nearly 5 times lower latency, as shown in Figure 87 and Figure 88.

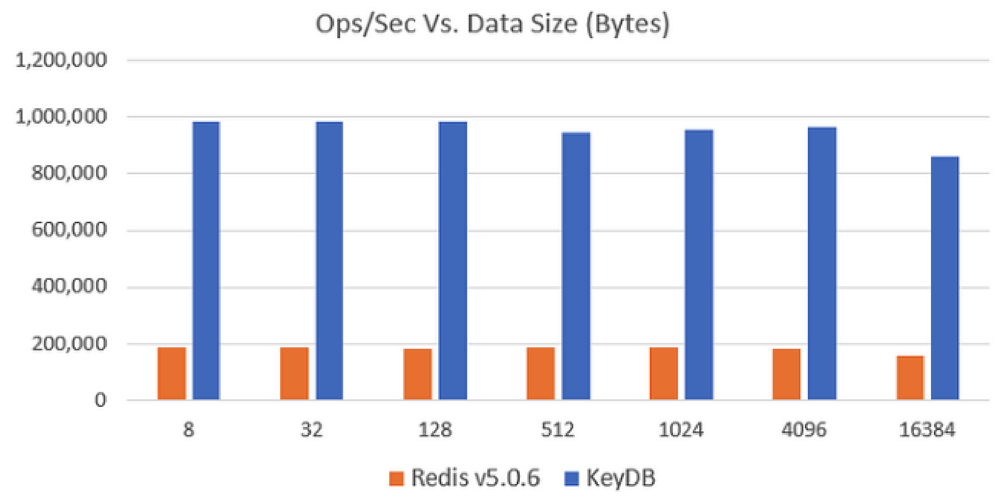


Figure 87: KeyDB vs Redis Ops/Sec performance

[12] https://github.com/RedisLabs/memtier_benchmark

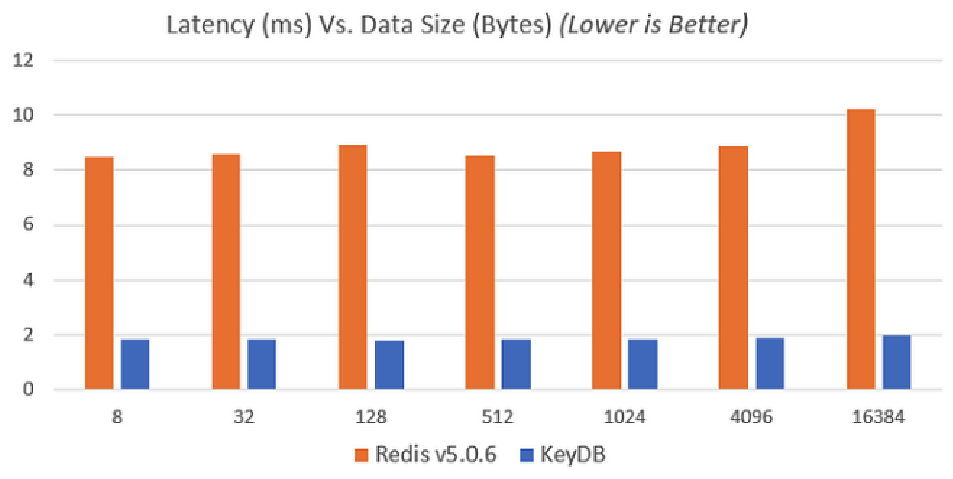


Figure 88: KeyDB vs Redis Latency performance

In our performance evaluation the data size varies from 8 bytes up to 500MB (32000 times larger data size) and is deployed KeyDB using 1, 7 and 14 server threads enabling thread affinity, to find out the best configuration regarding the server parallelism. The evaluation was carried out on an AMD Ryzen 9 3950X 16-Core processor (32 virtual CPUs and 132 GB RAM) hosted at UPM. As in the original performance evaluation Redis was deployed using the default configuration.

The results are shown in Figure 89 and Figure90. The performance of Redis and KeyDB configured with one server-thread is similar. The performance of KeyDB improves when using multiple threads. The performance of KeyDB with 7 server threads is similar to the one already published. The throughput increases up to 5 times with KeyDB and the latency is almost 5 times lower.

The throughput of KeyDB with 14 server threads is not twice the one with 7 server threads. The throughput increment is 23%. Analyzing the node resource consumption it was found that the CPU usage was close to 78% in the evaluation with 14 threads and close to 43% with 7 threads. This was due to the fact that the Memtier benchmark was running on the same node using 35% of the CPU with 14 threads and 19% with 7 threads.

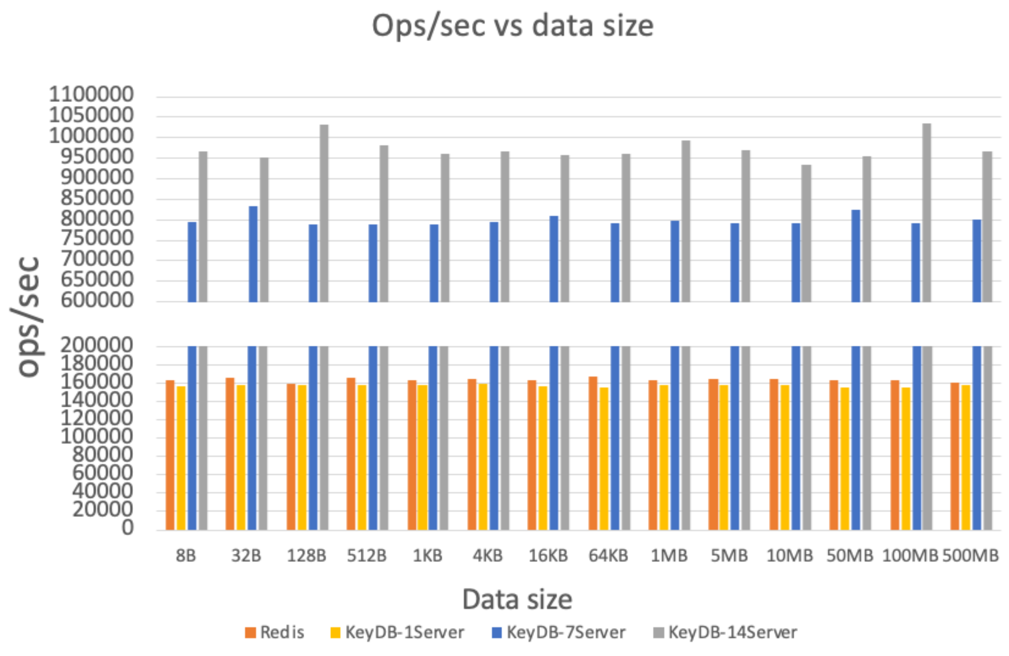


Figure 89: KeyDB and Redis performance evaluation (Ops/sec)

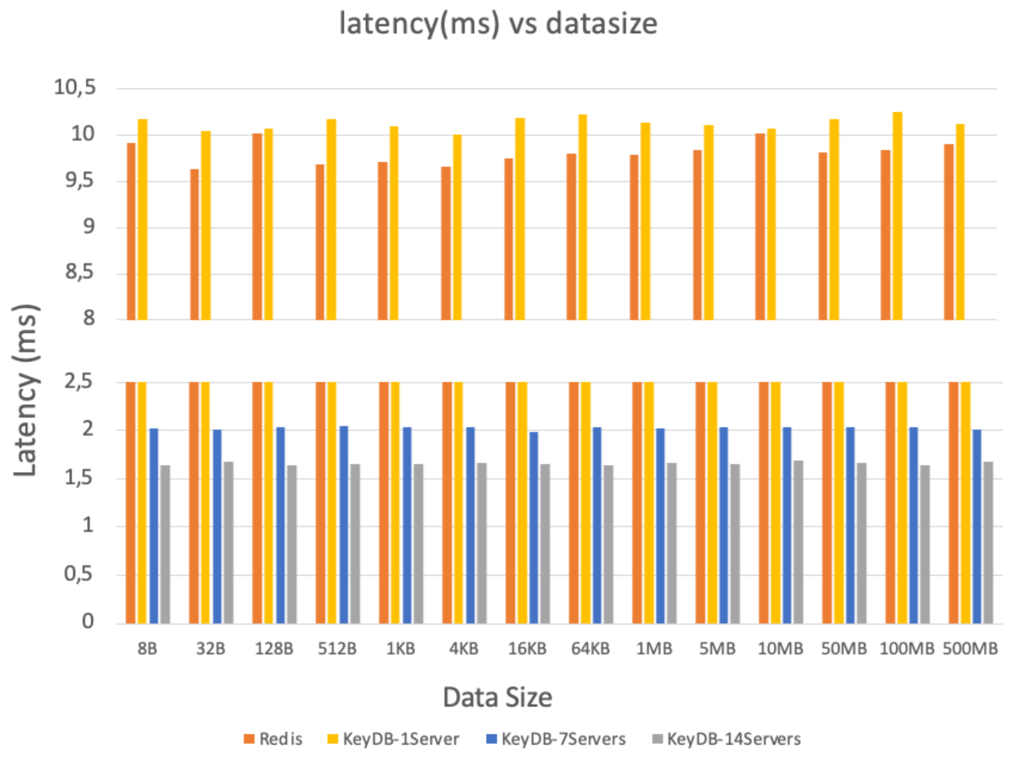


Figure 90: KeyDB and Redis performance evaluation (Latency)

Given a number of server-threads both the throughput and the latency remain quite stable independently of the data size. In this case there is no network traffic since the benchmark and the datastore run in the same node. Analyzing the results obtained with the different data sizes, both the operations per second and the latency results are similar with the different data sizes due to the client, the Memtier benchmark was hosted in the same machine and there was no network latency.

The same evaluation has been performed running the Memtier client from a different machine. Both nodes have the same hardware configuration and are connected through a 1gbps ethernet network. In this case, the data sizes are from 8 bytes up to 64 KB, with larger data sizes the network is saturated, and the results are not valid. The KeyDB database has been deployed with 7 threads, which is the same configuration as the 7 threads evaluation from Figure 89 and Figure 90. Figure 91 shows the results obtained, the performance has been reduced from the 80 kops/sec obtained with the database and the client co-located to 6 kops/sec when both are in different machines. As expected, latencies are higher when the client and the database are in different machines, 2.5ms, and 2ms when they are together.

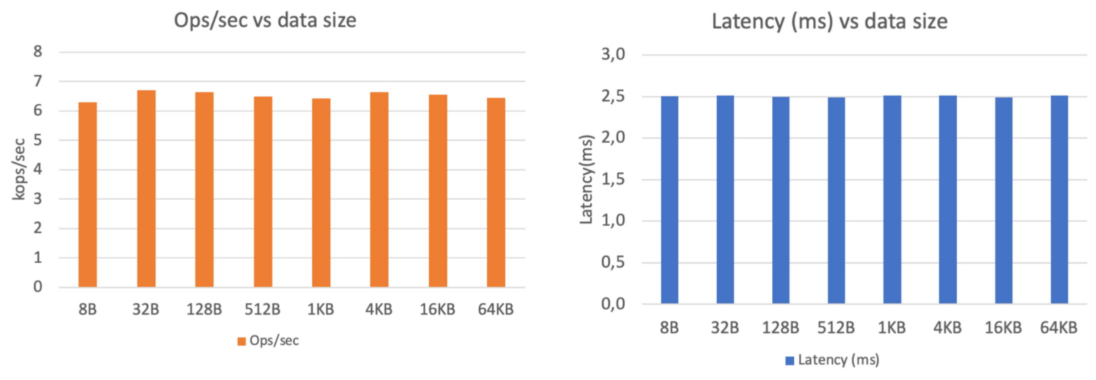


Figure 91: KeyDB evaluation with the client and the database in different nodes

5.3.2 DMS deployed in a Kubernetes cluster

The objective of this evaluation is to show the performance when the DMS is executed in a Kubernetes cluster as functions do in PHYSICS using different configurations. We tested different properties of KeyDB: persistence and replication. First of all, we compare the performance of KeyDB with and without persistence. Then, we tested active replication with persistence and persistence and no replication in three nodes.

5.3.2.1 Evaluation Setup

The evaluation was performed on a Kubernetes cluster with 6 Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz Processor instances (24 virtual CPUs and 128GB memory), all of them connected through a 56 gbps Infiniband network. The Kubernetes cluster was configured with a TCP/IP emulation limiting the bandwidth to 13 gbps.

The client is a Python function deployed in OpenWhisk running on the same Kubernetes cluster but, not co-located with the DMS. The data size varies from 1MB up to 100MB. The Python function gets the value associated with a key and writes that value with a different key (Code 17). The function returns the time needed for opening a connection, reading (get), writing (set) and closing the connection.

Code 17: Python function

```
from keydb import KeyDB
from time import time

def main(event):
latencies = {}
timestamps = {}
timestamps["starting_time"] = time()

host = event['host']
port = event['port']
key = event['key']
metadata = event['metadata']

try:
start = time()
db = KeyDB(host=host, port=port, password="")
connection_time = time() - start

start = time()
value=db.get(key)
get_time=time() - start

start = time()
db.set(key+"-w", value)
set_time=time() - start

start = time()
db.close
connection_close_time = time() - start

latencies['set_connection'] = connection_time
latencies['set'] = set_time
latencies['get'] = get_time
latencies['close_connection'] = connection_close_time
timestamps["finishing_time"] = time()

return {"latencies": latencies, "timestamps": timestamps, "metadata": metadata}
except Exception as e:
return {"error": e.with_traceback}
```

5.3.2.2 Outcomes

First of all, the evaluation compares the latency of the read operation (get) with and without persistence write operation (set) (Figures 92 and 93). There were 1,2 and 5 clients running concurrently, invoking the function. The DMS offers the same persistence guarantees KeydDB does. RDB persistence writes in a background process every 60 seconds if at least 1000 keys have changed. AOF persistence writes to a file every time a set operation is executed.

The DMS has been configured without persistence (green bars), with RDB persistence (orange bars), AOF persistence (grey bars) and RDB+AOF persistence (blue bars). The figures show that the impact on performance of using RDB persistence is around 20%, using AOF persistence is around 55% and using RDB+AOF persistence is 58%.

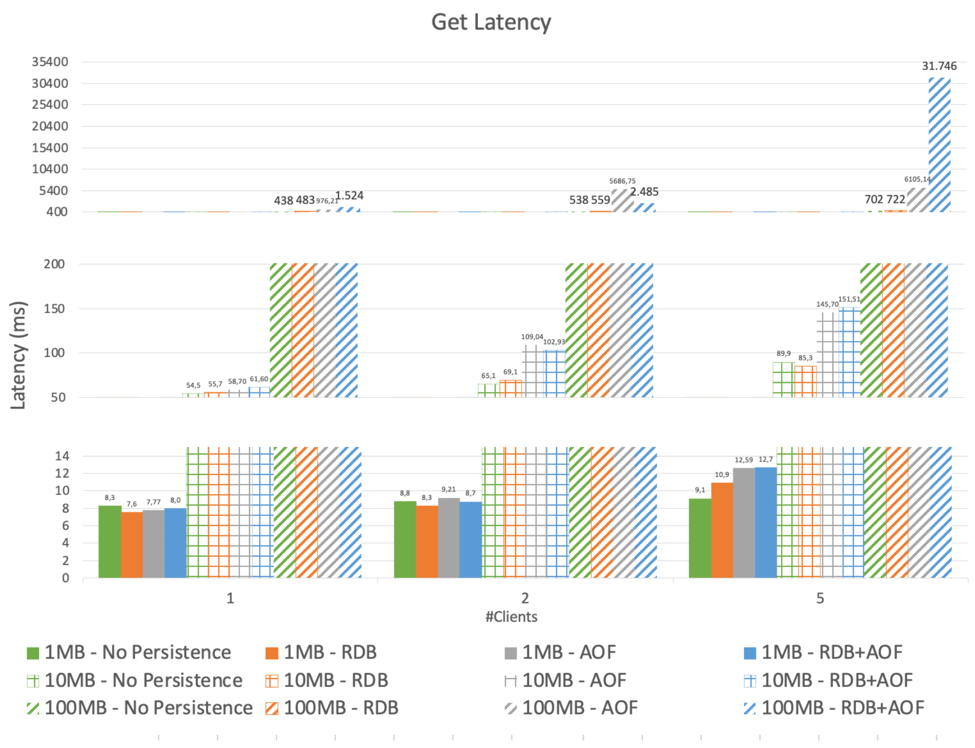


Figure 92: Get latency with and without persistence in 1 node

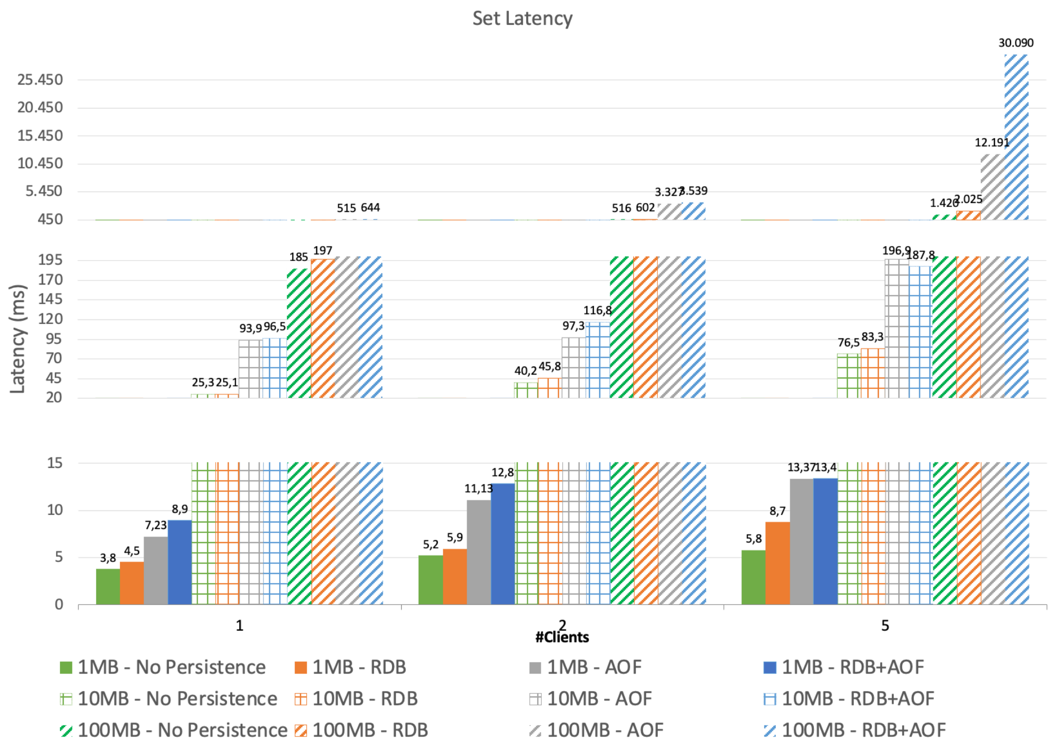


Figure 93: Set latency with and without persistence in 1 node

Figure 92 and Figure 93 show the latency of the get and set operations respectively with three DMS servers. This evaluation compares the latency without replication, with master-replica replication and with active replication, without persistence. Without replication, the data is partitioned among the three nodes. With master-replica configuration, the data is partitioned among the three nodes, each node is the master of its partitioned data and the other node is the replica. For instance, in the cluster the data managed by the node #1 is replicated in node #2, the data managed by node #2 is replicated in node #3 and the data served by node #3 is replicated in node #1. And with active replication the three nodes process read and write operations to any data item.

Figure 94 and Figure 95 show the latency of the get/set operation, respectively, when the data size varies from 1MB up to 10 MB with different numbers of concurrent clients. There is an overhead in both cases when there is replication.. This overhead can be high when the number of concurrent clients and data size increase. This happens because write operations need to be propagated to the replicas in the case of replication. When there is multi-master replication if the master does not store the requested key an exception is thrown to the client with the information about the master storing that key. The client then resubmits the request.

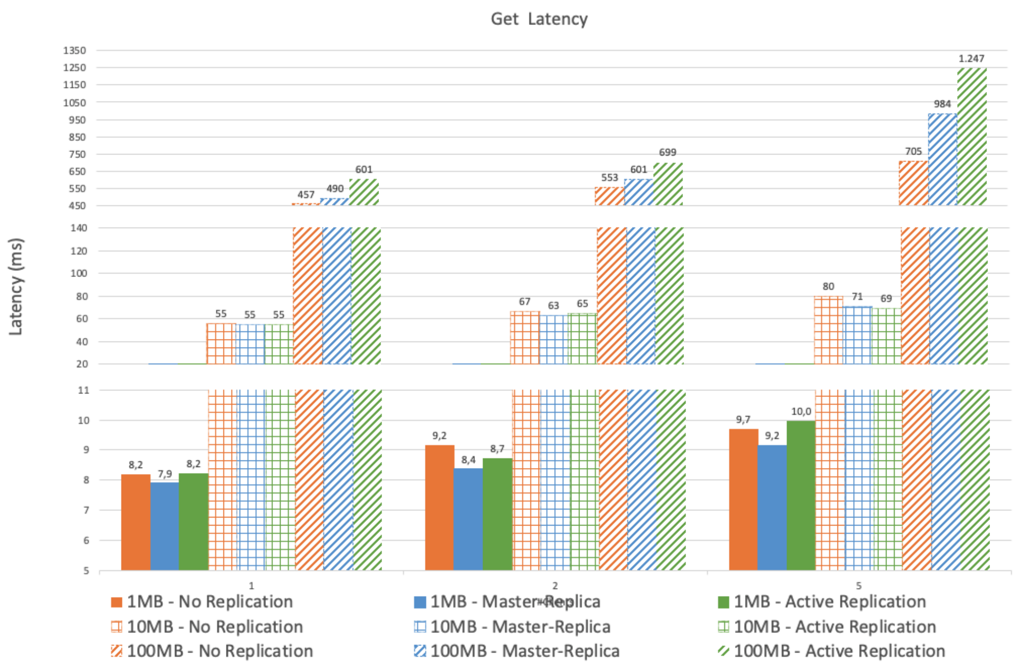


Figure 94: Get latency without persistence, without replication, master-replica and active replication in 3 nodes

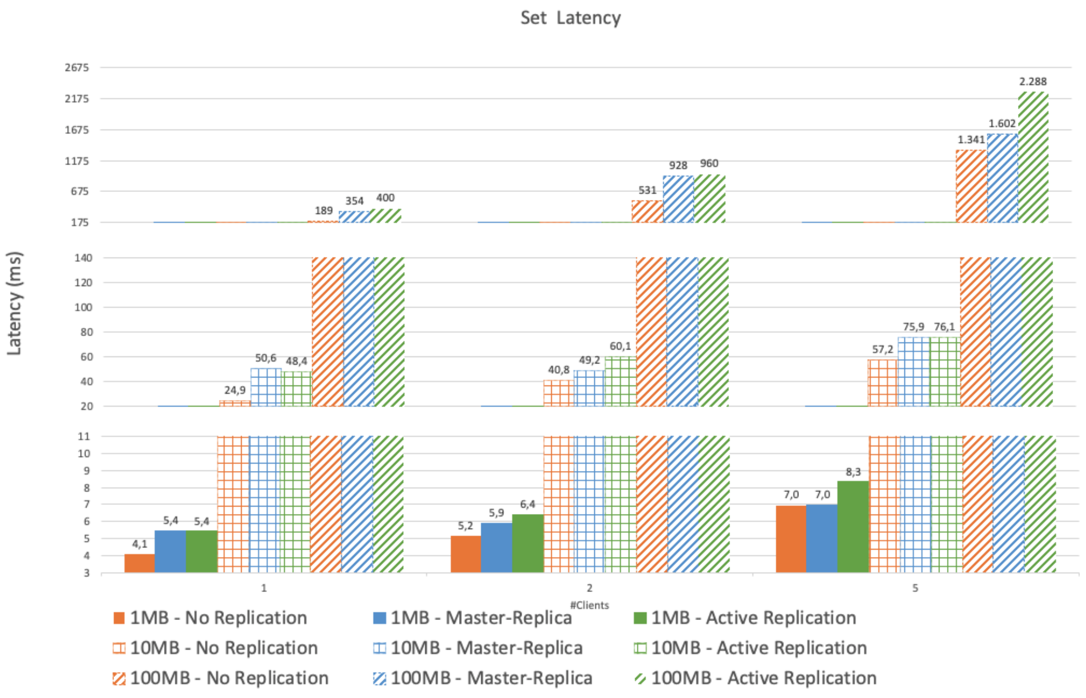


Figure 95: Set latency no persistence, no replication, master-replica and active replication in 3 nodes

6. Adaptive Platform Deployment, Operation & Orchestration

6.1. Functional description

The main objective of this component (Orchestrator) is to allow deployment, reconfiguration and adaptation of the applications defined in the deployment graph that was created with the toolkit of WP3, enhanced by the performance evaluator (Chapter 3) and transformed by the reasoning framework (Chapter 2) and finally decorated with location information by the Global Continuum using WP5 infrastructure information (Chapter 4). The Orchestrator will be in charge of the lifecycle management of applications divided into workflows for functions. The lifecycle management will implement initial deployment, update and reallocation of functions according to requirements.

The functionalities to be implemented by this component are:

- Parsing and translation of the Deployment Graph into Kubernetes YAML.
    - Use of ManifestWork and Workflow CRD definition to parse the application information.
    - Subdivide the app information into independent functions according to the location proposed by T4.3 optimization.
- Update workflow with status and forward this information to the central cluster using OCM feedback status custom definition.
    - Collect OpenWhisk communication information from the initial deployment to update the application in two steps.
    - Update interconnections between workflows across clusters using OpenWhisk default parameters that are included in the Workflow CRD definition.
- QoS enforcement by reading OpenWhisk Prometheus metrics and communicating issues via RabbitMQ.
    - Collection of messages in relation to QoS issues and update workflow information to use the fallback cluster location provided by T4.3.
- Communicate with the user concerning deployment status using RabbitMQ.

6.1.1. Background

6.1.1.1. Relationship with other components

As aforementioned, this component will get information from the Reasoning Framework and Global Continuum Placement to deploy applications and functions in some target infrastructure (cloud or edge). These two last components are part of the WP4 toolkit as the Orchestrator. The Orchestrator will also need to interact with resource management controllers from WP5 toolkit to be able to execute operations over the applications and functions deployed or to be deployed in the infrastructures controlled by this WP5 toolkit.

6.1.1.2. Requirements matrix

With relation to the expressed requirements in D2.2 and the functionalities described in the introduction of this chapter; this component is involved in the following requirements as it forwards and translates information from other components into the workflow CRD that can be leveraged by components in WP5:

- Req-4.5-FaaSRuntimeAdaptation: "The selected FaaS platform needs to have means of dynamically setting several parameters with relation to the operation of the platform".
- Req-4.5-placementDecision: "Design and implement an automated placement decision maker of virtual resources at the physical node level and reducing the noisy neighbor effect."
- Req-4.5-PersStorage: "FaaS platform should provide persistent storage (Like AWS S3 or Minio) for applications that also supports notifications based on the content changes."
- Req-4.5-CustomDockerImages: "FaaS platform could allow the execution of custom docker images from developers, with a specific structure, as a function."
- Req-4.5-FaaSandIaaSMonitoring: "The FaaS platform as well as the Container Orchestration platform need to expose collected monitoring metrics from benchmark functions execution."

6.1.2. Multicluster platform

To allocate application functions in a multi cluster environment, we leverage the Open cluster management (OCM) toolset that allows to set namespaces in a central cluster (OCM Hub cluster) as placeholders of cluster definitions that will be collected by the satellite clusters of the Physics platform (OCM Spoke clusters). OCM uses a CRD named ManifestWork[13] to define those resources that are to be deployed in a Spoke cluster. Therefore, the role of the Orchestrator is to translate a JSON application definition which is received as Deployment Graph into the ManifestWork format and place those in the correct namespace defined by the output of T4.3.

To make allocation and reconfiguration more flexible. The Applications, which may incorporate many workflows/functions, are split into as many ManifestWorks as functions. This allows the placement of different functions in a different cluster according to the requirements and optimizations of an application.

6.1.3. Connecting applications across clusters

Due to the application deployment distribution across clusters, the connectivity between functions of a specific application can be compromised. Using a mixed approach that leverages the WP3 at the application level (D3.2) and the WP5 Workflow CRD (D5.2), the WP4 provides the connectivity details required for the application to work by filling the OpenWhisk default parameters field per function in a two-step deployment process. A set of IP, namespaces and credentials are provided per action for the application running in OpenWhisk to make API calls to the remote OpenWhisk cluster and connect the different processing functions. This provides the flexibility to incorporate the information in complex applications and overcome the limitations of OpenWhisk for sequential execution of functions as the internal NodeRed application can incorporate the execution of functions into a custom and more complex logic.

6.1.4. Alarm component when cluster shows limitations

The idea of having a multicluster FaaS solution is to have the capability to migrate function execution to a different cluster when the preferred cluster is struggling to provide the required

[13] https://open-cluster-management.io/concepts/manifestwork/

QoS. As part of this prototype, the idea is to respond to OpenWhisk metrics per function to provide an example of execution redirection. With this in mind, the Orchestrator uses the output of T4.3 with optimal cluster location and fallback per function to deploy them into the two preferred clusters. The two-step deployment allows setting the default parameters for interconnected functions using the preferred cluster. While this should be the default behavior, an additional Alarm component deployed in each of the clusters is routinely monitoring the Openwhisk metrics and providing an alarm when a specific QoS metric is compromised. When this happens, the Orchestrator updates the default parameters of interconnected functions to point to the secondary cluster and thus, minimizes the time in which the QoS is compromised transparently for the final user of the platform. The use of Submariner (WP5) as a common multicluster networking solution allows this to work without the need for public IPs or additional networking infrastructure.

6.2. Technical description

6.2.1 Baseline technologies and dependencies

The following Table 17 shows the final baseline technologies used in this component. Some of the previous tools have been discarded, we included Kubernetes as a broader compatibility for Openshift and we eliminated the SLALite component. New components have also been added:

Table 17: Orchestrator component dependencies

Name	Description	License
Kubernetes	Kubernetes is an open-source container orchestrator platform. Kubernetes is maintained by the CNCF (Cloud Native Computing Foundation). https://kubernetes.io/	*Apache License 2.0*
OCM (WP5)	Open Cluster Management is a community-driven project focused on multicluster and multicloud scenarios for Kubernetes apps. https://open-cluster-management.io/	*Apache License 2.0*
OpenWhisk	Apache OpenWhisk is an open source serverless platform to execute functions triggered by events. https://openwhisk.apache.org	*Apache License 2.0*
Docker / CRI-O / Containerd	Runtime engine to execute containerized applications.	*Apache License, 2.0*
Prometheus	Prometheus is an open-source monitoring system. https://prometheus.io/	*Apache License 2.0*
Submariner	Submariner is a CNCF project that enables direct networking between Pods and Services in different Kubernetes based clusters, either on-premises or in the cloud. https://submariner.io/	*Apache License 2.0*

RabbitMQ	RabbitMQ is an open-source message broker that helps in scaling applications by deploying a message queuing mechanism in between them. https://www.rabbitmq.com/	*Mozilla Public License 2.0*
Golang	Golang is an open-source, compiled, statically typed, high-performing, readable, and efficient programming language designed by Google, that allows programs to be containerized into very small images, easy to manage and run in the Edge and Cloud. https://go.dev/	*3-clause BSD + patent grant*

6.2.2 Component internal architecture

The internal conceptual architecture of this component is shown in Figure 96 and shows the functionality described previously in section 6.1:

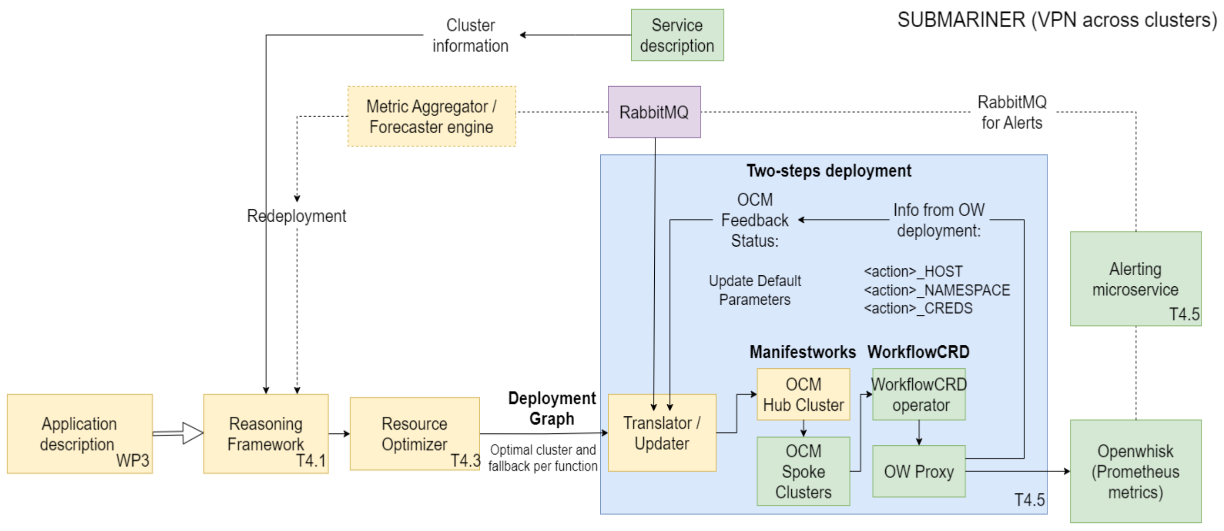


Figure 96: Diagram of WP4 components and Orchestration pipeline with runtime adaptation based on OpenWhisk metrics collected in Prometheus.

Apart from the baseline technologies used by the Orchestrator, this component uses the following applications (Table 18) developed during the project:

Table 18: Components required for the Orchestrator once the Deployment graph has been received.

Name	Description	License	Dependen-cies	Baseline tools
Translator	This component exposes an API to receive JSON description of the application (Deployment graph)	*Apache License 2.0*	Open Cluster Management	Golang, Docker.
OpenWhisk-Proxy (OW Proxy)	This component is a functions orchestrator in the continuum edge to cloud, which connects to tools like OpenWhisk and other FaaS.	*Apache License 2.0*	Operator, OpenWhisk	Golang, Docker, OpenWhisk
Workflow CRD Operator (developed as part of WP5)	The Operator is the component that connects to Kubernetes to be aware of changes in the applications manifests.	*Apache License 2.0*	Operator, Kubernetes, OCM, OW Proxy	Golang, Docker, Kubernetes
Monitoring Alerts (Alarming microservice)	The Monitoring Alerts system is the component responsible for checking that QoS definitions are met by the applications/ infrastructures.	*Apache License 2.0*	RabbitMQ, Translator	Golang, Prometheus, RabbitMQ, Docker

6.2.2.1 From JSON deployment graph to ManifestWork to workflow

JSON application graph to YAML workflow CRD

The following code snippets (Code 18) show how the JSON description of the application in the Deployment Graph is transformed into a YAML ManifestWork that will incorporate all the information from WP3 and WP4 into WP5 Workflow CRD.

Code 18: Code snippet with JSON Deployment Graph describing the application.

```
{
    "application": {
        "appName": "apptest1",
        "app_id": "app-test",
        "SoftwareImage": "",
        "flows": [
            {
                "flowID": "cd8e6011e15212bc",
                "flowName": "testing-artifact-image",
                "native": false,
                "type": "Flow",
                "executorMode": "NoderedFunction",
                "artifact": "registry.atosresearch.eu:18465/physicsgeorge185",
                "allocations": [
                    "cluster0"
                ],
                "functions": [
                    {
                        "displayName": "Hello?",
                        "code": "function main(msg){\nmsg.payload=msg.payload+' hello';\nreturn
msg;}",
                        "type": "Function",
                        "id": "6234e8d5c6b86c8f",
                        "sequence": 1,
                        "allocations": [],
                        "annotations": {}
                    }
                ],
                "annotations": {
                    "memory": "512",
                    "locality": "Cloud",
                    "timeout": "300000",
                    "goal": "performance"
                },
                "hasAction": []
            }
        ],
        "owner": "Carlos"
    },
    "platform": {
        "cluster0": {
            "id": "cluster0",
            "name": "hub-as-spoke",
            "type": "Cloud",
            "total_resources": {
                "nb_cpu": 28,
                "nb_gpu": 0,
                "memory_in_MB": 103981
            },
            "architecture": "x86_64",
            "objective_scores": {
                "Energy": 60,
                "Performance": 70,
                "Availability": 75
            }
        }
    },
    "allocations": [
        {
            "flowID": "cd8e6011e15212bc",
            "allocations": [
                {
                    "cluster": "cluster0",
                    "resource_id": "cd8e6011e15212bc"
                }
            ]
}]}
```

The Translator corrects possible formatting issues that are not accepted by Kubernetes and incorporates the required information for the Workflow CRD into a YAML formatted Manifeswork (Code 19), ignoring redundant data.

Code 19: Code Snippet with YAML ManifestWork that incorporates the Workflow CRD from the previous Deployment Graph.

```yaml
---
apiVersion: work.open-cluster-management.io/v1
kind: ManifestWork
metadata:
  namespace: hub-as-spoke #azure2 #okdhub-cluster
  name: apptest1-testing-artifact-image
spec:
  workload:
    manifests:
      - apiVersion: wp5.physics-faas.eu/v1alpha1
        kind: Workflow
        id: cd8e6011e15212bc
        metadata:
          name: testing-artifact-image
          namespace: default
          annotations:
            goal: "performance"
            locality: "Cloud"
            memory: "512"
            timeout: "300000"
        spec:
          execution: NoderedFunction
          listOfActions:
            - id: cd8e6011e15212bc
          native: false
          platform: openwhisk # DEFAULT
          type: Flow
          actions:
            - name: testing-artifact-image
              defaultParams:
                  actionname-host: ip
                  actionname-namespace: guest
                  actionname-credentials: sjavbisebces
              id: cd8e6011e15212bc
              image: registry/image:label
              runtime: blackbox
              semantics:
                  memory: "io-intensive"
              annotations:
                goal: "performance"
                locality: "Cloud"
                memory: "512"
                timeout: "300000"
  manifestConfigs:
    - resourceIdentifier:
        group: wp5.physics-faas.eu
        name: testing-artifact-image
        namespace: default
        resource: workflows
      feedbackRules:
        # - type: WellKnownStatus
        - type: JSONPaths
          jsonPaths:
            - name: namespace
              path: '.status.actionStatus[0].actionNamespace'
            - name: host
              path: '.status.actionStatus[0].actionHost'
            - name: credentials
              path: '.status.actionStatus[0].actionCredentials'
```

Operator and OpenWhisk Proxy microservice

The Workflow CRD Operator component is a Kubernetes/OCM controller that is running on each cluster of the PHYSICS platform, which is responsible for getting and applying the workflow information contained in the ManifestWorks generated by the Translator. To do that, this component listens first for the creation of new ManifestWorks or changes in them to process then the actions derived from this information by connecting to the OW Proxy application.

The component responsible for applying these actions is the OpenWhisk Proxy (OW Proxy) microservice. This component connects to the OpenWhisk API to deploy and manage workflow functions in the corresponding nodes. It also exposes a REST API with all the operations needed by the Operator to implement and execute the ManifestWorks workflows, including annotations (e.g., memory, limits, etc.) defined in these ManifestWorks.

Although the OpenWhisk Proxy component is used in this project to connect to OpenWhisk, it also uses a modular approach that makes possible an easy incorporation of other FaaS technologies like Knative and Kubeless.

Feedback status and default parameters

Once the OpenWhisk receives the function and this gets deployed, the OpenWhisk Proxy populates the Workflow CRD Status with the information required for the action invocation. The action invocation information and credentials are then communicated to the Hub Cluster using OCM feedback Status. Once in the Hub cluster, the Translator can collect this information and feed the one required to those workflows in an application that may require it to connect the overall application using OpenWhisk default parameters. This field has been provided in the Workflow CRD in collaboration with WP5 to pass the information.

6.2.2.2 Monitoring-Alerts and ManifestWork updates

The Monitoring-Alerts component is a Golang application responsible for managing the QoS objects used to monitor and check that applications and infrastructure metrics values accomplish the QoS requirements defined by users. First, this tool is responsible for creating and managing these QoS definitions. This also includes a REST API that can be accessed by other components to do these operations. Then, this component periodically connects to Prometheus to get the metrics defined in the QoS definitions in order to check that these metrics values meet the QoS requirements. Finally, if this tool detects that some QoS does not meet the user requirements, then it sends an alert message to the message broker (i.e., RabbitMQ).

The following snippet shows a QoS definition example in a JSON format (Code 20). This JSON defines the Prometheus metrics that are going to be monitored and checked, and the values or constraints for these metrics. If the values don't match the defined constraint, then the component sends a notification message to the message broker.

Code 20: Code snippet showing QoS definition example

```
{
    "id": "qosId1234",
    "name": "openwhisk_action_waitTime_seconds_count_definition",
    "state": "started",
    "details": {
                "name": "qos-definition-name",
                "creation": "2023-06-06T00:00:00Z",
                "guarantees": [{
                                "name": "openwhisk_action_waitTime_seconds_count",
                                "constraint": "[metric1/metric2] < XXX"
                }]
    }
}
```

This monitoring component manages and evaluates OpenWhisk metrics that are sent to Prometheus. These metrics are used to measure the status of the actions deployed and running by the orchestrator in the nodes.

OpenWhisk metrics in Prometheus metric collection

The QoS constraint used in the Monitoring Alarms component is the following:

Code 21: Code Snippet for Prometheus metric processing

```
openwhisk_action_waitTime_seconds_sum / openwhisk_action_waitTime_seconds_count < XXX
```

Prometheus gathers these metrics from OpenWhisk and makes them accessible from the API used by tools like the Monitoring Alerts system.

Alarm queue in RabbitMQ

All the alarm notifications that are sent to the message broker have the following information: the QoS definition identifiers, the metric value that generated the alarm, the datetime, and the action identifier.

Code 22: Code snippet of the Alarm notification example

```
{
    "QoS": "openwhisk_action_waitTime_seconds_count",
    "QoSId": "d03",
    "Result": [{
        "key": "10.42.1.140:9095",
        "action": "testing-artifact-image/testing-artifact-image",
        "value": 11.639000000000001,
        "datetime": "2023-08-23T12:04:49.713Z"
        }
    ]
}
```

Update default parameters after alarm trigger

The trigger signals the Translator to update the Default parameters with the feedback status values for the secondary location and allow this way a reconfiguration based on infrastructure load (Figure 97). Using the ID of the application and the function/workflow name, it can collect this information from the secondary cluster and update all the default parameters that mention that function name within the same application similar to how it updates the default parameters in two-step deployment explained in the previous section 6.2.2.1 (Table 19).

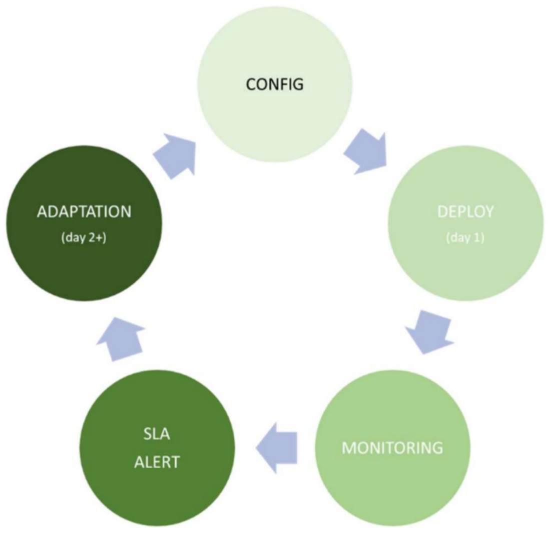


Figure 97: Plan, Do, Check, Act cycle in the Orchestrator

Table 19: Update of fields for default parameters in the ManifestWork due to the reallocation of interconnected functions.

YAML MANIFESTWORK BEFORE ALARM	YAML MANIFESTWORK AFTER ALARM (Updated default params)

```
actions:
  - name: testing-artifact-image
    defaultParams:
      actionname-host: ip-first
      actionname-namespace: guest
      actionname-credentials: creds
```
	```
	actions:
	  - name: testing-artifact-image
	    defaultParams:
	      actionname-host: ip-fallback
	      actionname-namespace: guest
	      actionname-credentials: fallback-creds
	```

6.2.3 Interfaces and integrations

The information from the Reasoning Framework for Semantic Matching and Runtime Adaptation about resource management controllers' endpoints and credentials, to be defined in WP5, to be able to connect to the target controller to deploy the workload. The Global Continuum Patterns Placement component starts the deployment of an application or a standalone function in the selected target FaaS platform. The Orchestrator parses and translates the Deployment Graph from the Continuum Placement to a deployment schema valid for the underlying FaaS platform (OpenWhisk) and this is used by the WP5 operator and OpenWhisk proxy to interact with the API of the target FaaS platform to deploy the workload. The Orchestrator has to extract and translate not only the functions contained in the application but also keep connection between the set of functions that form the application.

There are API interfaces available to expose the basic functionalities of the Orchestrator to other components. The operations in this API are REST API around the main entities of the component (like functions and sequences). In a FaaS environment we have to separate the execution of a function in two steps: initialization phase (deployment) and run phase (invocation). This is necessary because the FaaS engine has to preconfigure the runtime environment to allow fast execution of functions (function loading and preparation). The run phase receives the parameters provided at the runtime and starts the function. This execution could be a warm start (reusing runtime engines like docker containers) or a cold start for the first time. The following table shows the API operations of the orchestrator components.

6.2.3.1. Translator single endpoint API

The Translator component offers one single operation which is responsible for creating the ManifestWorks (Table 20).

Table 20: Translator unique POST endpoint that received the JSON deployment graph.

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

6.2.3.2. Monitoring-Alerts component API

This component offers a REST API that can be accessed by other PHYSICS components. These are the methods exposed by this API (Table 21).

Table 21: Endpoints for the Alarm microservice API.

Method	**Path/URI**	**Description**
GET	*/api/v1/qos*	Returns the list of all QoS definitions
POST	*/api/v1/qos*	Creates a new QoS definition
GET	*/api/v1/qos/{id}*	Returns the information about the QoS definition identified by *{id}*
DELETE	*/api/v1/qos/{id}*	Deletes the QoS definition identified by *{id}*

6.2.3.3. OpenWhisk proxy API

This component is called by the Operator (WP5) component, and it exposes a REST API offering the operations shown in Table 22.

Table 22: Endpoints of the OpenWhisk proxy API.

Method	**Path/URI**	**Description**
GET	*/api/v1/function/{id}*	Returns the information about a function (packages+actions)
POST	*/api/v1/function*	Creates and deploys a function (JSON definition)
DELETE	*/api/v1/function/{id}*	Deletes a function
PUT	*/api/v1/function/{id}*	Updates a function
GET	*/api/v1/functions*	Returns the list of functions (packages+actions)
POST	*/api/v1/functions/deploy*	Creates and deploys a function (Manifest/YAML definition)
POST	*/api/v1/functions/run*	Runs the actions of the function
POST	*/api/v1/functions/undeploy*	Undeploy the function

6.3. Demonstration and experimentation

6.3.1. Scenario or experimentation description

The following demonstration scenario shows a complete deployment and adaptation flow (Figure 98), which includes the deployment of a "deployment graph" in the Translator (1), and the resulting interactions between this component, the Operator, the OW-Proxy and the Monitoring Alertes needed to deploy, run (2,3) and update (4,5) functions in the selected testbed node.

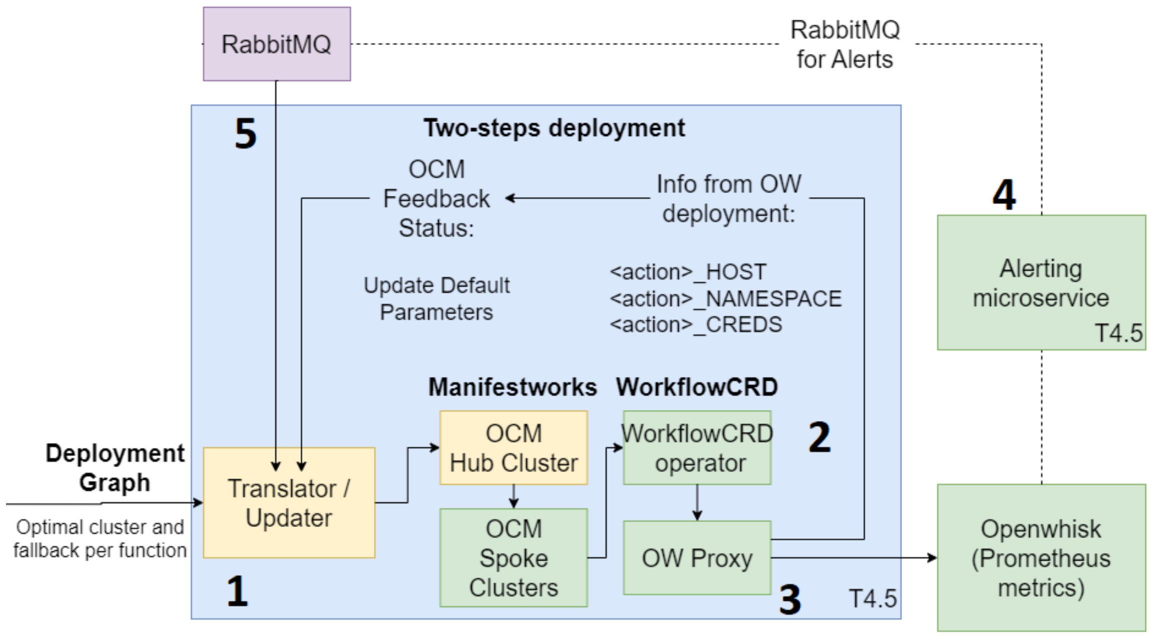


Figure 98: Deployment flow

6.3.1.1- Translator - Deployment Graph

The process starts when the Translator receives a Deployment Graph (Code 23).

1. Send Deployment Graph JSON to Translator (POST /api/v1/deploy)

Code 23: Code snippet showing an example of a Deployment Graph

```
{ "application": {
        "appName": "",
        "app_id": "app-test-infra-atos",
        "SoftwareImage": "",
        "flows": [ {
                "flowID": "f4a0fb9518d4df8d",
                "flowName": "testing-artifact-image",
                "native": false,
                "type": "Flow",
                "executorMode": "NoderedFunction",
                "artifact": "registry.atosresearch.eu:18465/physicsgeorge185",
                "allocations": [
                    "cluster0"],
                "functions": [{
                        "displayName": "Hello?",
                        "code": "function main(msg){\nmsg.payload=msg.payload+'
hello';\nreturn msg;}",
                        "type": "Function",
                        "id": "6234e8d5c6b86c8f",
                        "sequence": 1,
                        "allocations": [],
                        "annotations": {}}],
                "annotations": {
                    "memory": "512",
                    "locality": "Cloud",
                    "timeout": "300000",
                    "goal": "performance"
                },
                "hasAction": []
            }
        ],
        "owner": "Carlos"
    },
    "platform": {
        "cluster0": {
            "id": "cluster0",
            "name": "hub-as-spoke",
            "type": "Cloud",
            "total_resources": {
                "nb_cpu": 28,
                "nb_gpu": 0,
                "memory_in_MB": 103981
            },
            "architecture": "x86_64",
            "objective_scores": {
                "Energy": 60,
                "Performance": 70,
                "Availability": 75
            }
        }
    },
    "allocations": [{
            "flowID": "f4a0fb9518d4df8d",
            "allocations": [{
                    "cluster": "cluster0",
                    "resource_id": "f4a0fb9518d4df8d"}]}]
}
```

2. When the Translator receives the Deployment Graph, it generates the corresponding ManifestWorks with the Workflow CR object inside..
3. These ManifestWorks are sent to OCM.
4. OCM sends the ManifestWorks to the correspondent managed cluster (i.e., Kubernetes).
    a. Create/Delete/Update CR entity (Workflow CRD)
    b. Manifest is applied (Spec)
5. After ManifestWorks are sent to Kubernetes nodes via OCM, we can see them in the managed cluster (Code 24).

Code 24: Code snippet of the output of the ManifestWorks in the cluster

```
$ kubectl get manifestworks -A

NAMESPACE     NAME                                          AGE
hub-as-spoke  chj2avbf753pdpdla6ug--testing-native-sequence  21s
hub-as-spoke  chj2avbf753pdpdla6ug--testing-artifact-image   21s
```

6.3.1.2- Workflow CRD Operator

This component (from WP5, see D5.2) is continuously listening for changes in the Kubernetes nodes. When the Operator detects a new Workflow CR object or an update in an existing one, then it connects to OpenWhisk Proxy application to apply the needed changes:

1. The Operator checks for status changes and reconciles Spec and Status for Workflow CRD
2. The Operator connects to OpenWhisk Proxy to (un)register the corresponding functions

6.3.1.3- OpenWhisk Proxy

This component exposes a REST API to allow other components to interact with FaaS tools like OpenWhisk. In this case, after receiving instructions from the Operator it does the following:

1. OpenWhisk-Proxy connects to OpenWhisk API to deploy the functions
2. This component uses the OpenWhisk API to deploy, update and delete the actions when required.

6.3.1.4- Monitoring Alerts

The Monitoring Alerts component is continuously checking the QoS constraints metrics defined in the QoS definitions (see section 6.2.2.2 Monitoring-Alerts and ManifestWork updates). Thus, the first thing that needs to be done is the definition of the QoS object with its metrics (Code 25).

1. Add a QoS object / definition (POST /api/v1/qos)

Code 25: Code snippet of the QoS object/definition

```
{
    "id": "d03",
    "name": "openwhisk_action_waitTime_seconds_count_definition",
    "state": "started",
    "details": {
      "id": "d03",
      "type": "qos",
      "name": "qos-definition-name",
      "creation": "2023-06-06T00:00:00Z",
      "guarantees": [{
          "name": "openwhisk_action_waitTime_seconds_count",
          "constraint":
"[openwhisk_action_waitTime_seconds_sum%2Fopenwhisk_action_waitTime_seconds_count] < 4"
        } ] }
}
```

2. The Monitoring-Alerts retrieves the metrics values from Prometheus and evaluates the QoS constraints defined in the QoS definitions.
3. In case the Monitoring-Alerts detects a violation, it sends a notification message to RabbitMQ. These notifications include the identifier of the action (Code 26).

Code 26: Code snippet of the QoS violation notification

```
{
    "QoS": "openwhisk_action_waitTime_seconds_count",
    "QoSId": "d03",
    "Result": [{
            "key": "10.42.1.140:9095",
            "action": "testing-artifact-image/testing-artifact-image",
            "value": 11.639000000000001,
            "datetime": "2023-08-23T12:04:49.713Z"
        }
    ]
}
```

6.3.1.5- RabbitMQ violations - Translator

The last step includes the update of the function after an alert notification has been received.

1. RabbitMQ receives the alert notification.
2. The translator reads the alert notification
3. The translator checks for the application/workflow to update the dependent function parameters for remote invocations to point to the fallback cluster of the failing workflow.
4. Translator notifies through rabbitmq of the changes to be processed in the user interface.

7. Integration with Physics platform

Physics platform uses Jenkins pipelines to build images, test for features, push those into the Harbour image repository as well as incorporate those images into the platform as functions or as services. Due to the presence of this infrastructure required for other WP (see deliverable D3.2), it has been decided to leverage the platform for other components that are integral parts of the orchestration mechanisms (WP4). Therefore, the components from WP4 use the GitLab code repository to feed code information into Jenkins and finally create images and push commands into the Kubernetes cluster. A more detailed description of this process of component installation and usage will be provided in the Physics handbook in deliverable D6.2

8. Conclusions

This deliverable reports on the work done in WP4 in the totality of the PHYSICS project. It describes the prototype per isolated component in this period that targets the objectives and basic functionalities required from WP4 as described in the project architecture. An overview of each of the components is provided as well as some of the important interconnections between them. Some scenarios and experimentations per standalone component have been described in this deliverable with some outcomes and conclusions. The WP3 output (application graph semantic) is decorated in the WP4 pipeline to form a global Deployment graph with inputs from WP3 and WP5 at different stages. The Deployment graph is the main resource to obtain a final deployment in the target Kubernetes infrastructure. Instructions to download and configure the independent components are provided in the D6.2. The open source code of the toolkit is available in the official source code repository. Next efforts will be concentrated on the final integration between the different WPs to obtain an interoperated toolkit for PHYSICS that uses the advances developed in this WP4.

9. References

[1] 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.
[2] Heath, Tom, Berners-Lee, Tim, and Bizer, Christian, ‘Semantic services, interoperability and web applications: emerging concepts’, *Chapter Linked data: The story so far*, 2011.
[3] M. R. Mufid, A. Basofi, M. U. H. Al Rasyid, I. F. Rochimansyah, and others, ‘Design an mvc model using python for flask framework development’, in *2019 International Electronics Symposium (IES)*, IEEE, 2019, pp. 214–219.
[4] Roger, Sadjirin, Halim, Zulkifli, Canda, Roger, Sadjirin, Roslan, and Sahri, Zulazeze, ‘Machine learning model deployment as API service using Python Web Framework’, in *Digital Technology Empowerment Anticipates Expertise*, 2021.
[5] D. Fernandes and J. Bernardino, ‘Graph Databases Comparison: AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, and OrientDB.’, in *Data*, 2018, pp. 373–380.
[6] Y. Poulakis, G. Fatouros, G. Kousiouris, and D. Kyriazis, ‘HOCC:An ontology for holistic description of cluster settings’, in *19th International Conference, GECON 2022, September 13-15*, 2022.
[7] ‘Serverless Computing – AWS Lambda Pricing – Amazon Web Services’, *Amazon Web Services, Inc.* https://aws.amazon.com/lambda/pricing/ (accessed Aug. 03, 2023).
[8] B. Chesneau, ‘Gunicorn documentation’. 2017.
[9] R. J. Hyndman and G. Athanasopoulos, ‘Forecasting: principles and practice, 2013’, *URL: https://www. otexts. org/fpp [accessed 2018-02-15][WebCite Cache ID 6xFJlXCQI]*, 2014.
[10] ‘OpenAPI Specification’. 2021. [Online]. Available: https://spec.openapis.org/oas/v3.1.0
[11] W. Zheng, L. Zou, W. Peng, X. Yan, S. Song, and D. Zhao, ‘Semantic SPARQL Similarity Search over RDF Knowledge Graphs’, *Proc. VLDB Endow.*, vol. 9, no. 11, pp. 840–851, Jul. 2016, doi: 10.14778/2983200.2983201.
[12] 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.
[13] A. Raouzaiou *et al.*, ‘D5.3 – WP 5 Scientific Report and Prototype Description – Y3’, Jan. 2021, Accessed: Sep. 13, 2023. [Online]. Available: https://zenodo.org/record/4442314
[14] ‘BigDataStack | High-performance data-centric stack for big data applications and operations’. https://bigdatastack.eu/ (accessed Sep. 13, 2023).
[15] 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.
[16] 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.
[17] ‘gkousiou/octavefunction2 - Docker Image | Docker Hub’. https://hub.docker.com/r/gkousiou/octavefunction2 (accessed Sep. 13, 2023).
[18] R. Ferreira da Silva, R. Filgueira, E. Deelman, E. Pairo-Castineira, I. M. Overton, and M. P. Atkinson, ‘Using simple PID-inspired controllers for online resilient resource management of distributed scientific workflows’, *Future Generation Computer Systems*, vol. 95, pp. 615–628, Jun. 2019, doi: 10.1016/j.future.2019.01.015.

[19] I. Bouras *et al.*, ‘Mapping of Quality of Service Requirements to Resource Demands for IaaS’:, in *Proceedings of the 9th International Conference on Cloud Computing and Services Science*, Heraklion, Crete, Greece: SCITEPRESS - Science and Technology Publications, 2019, pp. 263–270. doi: 10.5220/0007676902630270.

[20] X. Chen, H. Wang, Y. Ma, X. Zheng, and L. Guo, ‘Self-adaptive resource allocation for cloud-based software services based on iterative QoS prediction model’, *Future Generation Computer Systems*, vol. 105, pp. 287–296, Apr. 2020, doi: 10.1016/j.future.2019.12.005.

[21] 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’, *Journal of Systems and Software*, vol. 84, no. 8, pp. 1270–1291, Aug. 2011, doi: 10.1016/j.jss.2011.04.013.

[22] D. N. Jha, S. Garg, P. P. Jayaraman, R. Buyya, Z. Li, and R. Ranjan, ‘A Holistic Evaluation of Docker Containers for Interfering Microservices’, in *2018 IEEE International Conference on Services Computing (SCC)*, San Francisco, CA, USA: IEEE, Jul. 2018, pp. 33–40. doi: 10.1109/SCC.2018.00012.

[23] J. Park, H. Kim, and K. Lee, ‘Evaluating Concurrent Executions of Multiple Function-as-a-Service Runtimes with MicroVM’, in *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, Beijing, China: IEEE, Oct. 2020, pp. 532–536. doi: 10.1109/CLOUD49709.2020.00080.

[24] G. Aumala, E. Boza, L. Ortiz-Aviles, G. Totoy, and C. Abad, ‘Beyond Load Balancing: Package-Aware Scheduling for Serverless Platforms’, in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, Larnaca, Cyprus: IEEE, May 2019, pp. 282–291. doi: 10.1109/CCGRID.2019.00042.

[25] J. Bravo Ferreira, M. Cello, and J. O. Iglesias, ‘More Sharing, More Benefits? A Study of Library Sharing in Container-Based Infrastructures’, in *Euro-Par 2017: Parallel Processing*, F. F. Rivera, T. F. Pena, and J. C. Cabaleiro, Eds., in Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 358–371. doi: 10.1007/978-3-319-64203-1_26.

[26] D. Taibi, N. El Ioini, C. Pahl, and J. Niederkofler, ‘Patterns for Serverless Functions (Function-as-a-Service): A Multivocal Literature Review’:, in *Proceedings of the 10th International Conference on Cloud Computing and Services Science*, Prague, Czech Republic: SCITEPRESS - Science and Technology Publications, 2020, pp. 181–192. doi: 10.5220/0009578501810192.

[27] martinekuan, ‘Circuit Breaker pattern - Azure Architecture Center’. https://learn.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker (accessed Sep. 14, 2023).

[28] J. Grogan *et al.*, ‘A Multivocal Literature Review of Function-as-a-Service (FaaS) Infrastructures and Implications for Software Developers’, in *Systems, Software and Services Process Improvement*, M. Yilmaz, J. Niemann, P. Clarke, and R. Messnarz, Eds., in Communications in Computer and Information Science, vol. 1251. Cham: Springer International Publishing, 2020, pp. 58–75. doi: 10.1007/978-3-030-56441-4_5.

[29] A. Eivy and J. Weinman, ‘Be Wary of the Economics of “Serverless” Cloud Computing’, *IEEE Cloud Comput.*, vol. 4, no. 2, pp. 6–12, Mar. 2017, doi: 10.1109/MCC.2017.32.

[30] E. Van Eyk, A. Iosup, S. Seif, and M. Thömmes, ‘The SPEC cloud group’s research vision on FaaS and serverless architectures’, in *Proceedings of the 2nd International Workshop on Serverless Computing*, Las Vegas Nevada: ACM, Dec. 2017, pp. 1–4. doi: 10.1145/3154847.3154848.

[31] ‘Serverless Boom or Bust? An Analysis of Economic Incentives | USENIX’. https://www.usenix.org/conference/hotcloud20/presentation/lin (accessed Sep. 13, 2023).

[32] R. Cordingly, W. Shu, and W. J. Lloyd, ‘Predicting performance and cost of serverless computing functions with SAAF’, in *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*, IEEE, 2020, pp. 640–649.
[33] ‘Function Reference: ols’. https://octave.sourceforge.io/octave/function/ols.html (accessed Sep. 13, 2023).
[34] F. Khomh and S. A. Abtahizadeh, ‘Understanding the impact of cloud patterns on performance and energy consumption’, *Journal of Systems and Software*, vol. 141, pp. 151–170, Jul. 2018, doi: 10.1016/j.jss.2018.03.063.
[35] N. Mahmoudi, C. Lin, H. Khazaei, and M. Litoiu, ‘Optimizing serverless computing: introducing an adaptive function placement algorithm’, in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, in CASCON ’19. USA: IBM Corp., Nov. 2019, pp. 203–213.
[36] P. K. Gadepalli, G. Peach, L. Cherkasova, R. Aitken, and G. Parmer, ‘Challenges and Opportunities for Efficient Serverless Computing at the Edge’, in *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, Oct. 2019, pp. 261–2615. doi: 10.1109/SRDS47363.2019.00036.
[37] J. Kijak, P. Martyna, M. Pawlik, B. Balis, and M. Malawski, ‘Challenges for Scheduling Scientific Workflows on Cloud Functions’, in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, Jul. 2018, pp. 460–467. doi: 10.1109/CLOUD.2018.00065.
[38] J. Fang, J. Hu, J. Wei, T. Liu, and B. Wang, ‘An Efficient Resource Allocation Strategy for Edge-Computing Based Environmental Monitoring System’, *Sensors*, vol. 20, no. 21, p. 6125, Jan. 2020, doi: 10.3390/s20216125.
[39] Q. Luo, S. Hu, C. Li, G. Li, and W. Shi, ‘Resource Scheduling in Edge Computing: A Survey’, *IEEE Communications Surveys & Tutorials*, vol. 23, no. 4, pp. 2131–2165, 2021, doi: 10.1109/COMST.2021.3106401.
[40] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, ‘Optimizing inference serving on serverless platforms’, *Proceedings of the VLDB Endowment*, vol. 15, no. 10, Jun. 2022, doi: 10.14778/3547305.3547313.
[41] S. Kho Lin *et al.*, ‘Auto-Scaling a Defence Application across the Cloud Using Docker and Kubernetes’, in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, Dec. 2018, pp. 327–334. doi: 10.1109/UCC-Companion.2018.00076.
[42] C. Lu, Y. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son, ‘Feedback Control Architecture and Design Methodology for Service Delay Guarantees in Web Servers’, *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 9, pp. 1014–1027, Sep. 2006, doi: 10.1109/TPDS.2006.123.
[43] W. Pan, D. Mu, H. Wu, and L. Yao, ‘Feedback Control-Based QoS Guarantees in Web Application Servers’, in *2008 10th IEEE International Conference on High Performance Computing and Communications*, Sep. 2008, pp. 328–334. doi: 10.1109/HPCC.2008.106.
[44] Z. Cai and R. Buyya, ‘Inverse Queuing Model-Based Feedback Control for Elastic Container Provisioning of Web Systems in Kubernetes’, *IEEE Transactions on Computers*, vol. 71, no. 2, pp. 337–348, Feb. 2022, doi: 10.1109/TC.2021.3049598.
[45] T. Patikirikorala, A. Colman, J. Han, and L. Wang, ‘A multi-model framework to implement self-managing control systems for QoS management’, in *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, in SEAMS ’11. New York, NY, USA: Association for Computing Machinery, May 2011, pp. 218–227. doi: 10.1145/1988008.1988040.

[46] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.
[47] P. Singh, P. Gupta, K. Jyoti, and A. Nayyar, 'Research on Auto-Scaling of Web Applications in Cloud: Survey, Trends and Future Directions', *Scalable Computing: Practice and Experience*, vol. 20, no. 2, pp. 399–432, May 2019, doi: 10.12694/scpe.v20i2.1537.
[48] D. B. Shmoys and É. Tardos, 'An approximation algorithm for the generalized assignment problem', *Mathematical Programming*, vol. 62, no. 1, pp. 461–474, Feb. 1993, doi: 10.1007/BF01585178.
[49] L. Angelelli, A. A. da Silva, Y. Georgiou, M. Mercier, G. Mounié, and D. Trystram, 'Towards a Multi-objective Scheduling Policy for Serverless-based Edge-Cloud Continuum', in *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, May 2023, pp. 485–497. doi: 10.1109/CCGrid57682.2023.00052.
[50] T. Mandarin, 'KeyDB is a Fork of Redis that is 5X Faster', *Medium*, Feb. 10, 2020. https://medium.com/@tedmandarin/keydb-is-a-fork-of-redis-that-is-5x-faster-164757232bac (accessed Sep. 21, 2023).

Disclaimer

Copyright Message