BigDataStack
Holistic stack for big data applications and operations

Project Title	High-performance data-centric stack for big data applications and operations
Project Acronym	BigDataStack
Grant Agreement No	779747
Instrument	Research and Innovation action
Call	Information and Communication Technologies Call (H2020-ICT-2016-2017)
Start Date of Project	01/01/2018
Duration of Project	36 months
Project Website	http://bigdatastack.eu/

D3.3 – WP 3 Scientific Report and Prototype Description – Y3

Work Package	WP3 – Data-driven Infrastructure Management
Lead Author (Org)	Antonio Castillo Nieto (ATOS)
Contributing Author(s) (Org)	Ismael Cuadrado-Cordero, Orlando Avila-García (ATOS) Bernat Quesada (ATOS WDL), Jean Didier Totow, Christos Lyvas (UPRC), Sophia Karagiorgou (UBI), Nikos Drosos (SILO), Mauricio Fadel Argerich, Bin Cheng (NEC), Patricio Martinez Gracia, Jose Maria Zaragoza (LXS), Richard McCreadie, Zaiqiao Meng, Craig Macdonald (GLA), Luis Tomas Bolivar (RHT)
Internal Reviewer(s)	Yosef Moatti (IBM), Ricardo Jimenez-Peris (LXS), Dimosthenis Kyriazis (UPRC)
Due Date	30.10.2020
Date	30.10.2020

European Commission

The work described in this document has been conducted within the project BigDataStack. This project has received funding from the European Union's Horizon 2020 (H2020) research and innovation programme under the Grant Agreement no 779747. This document does not represent the opinion of the European Union, and the European Union is not responsible for any use that might be made of such content.

Version	1.0

Dissemination Level

X	PU: Public (*on-line platform)
	PP: Restricted to other programme participants (including the Commission)
	RE: Restricted to a group specified by the consortium (including the Commission)
	CO: Confidential, only for members of the consortium (including the Commission)

CC BY NC ND

Versioning and contribution history

Version	Date	Author	Notes
0.1	30.09.2020	Orlando Avila-García (ATOS)	Creation of the skeleton and first draft as a copy of D3.2.
0.2	06.10.2020	Orlando Avila-García (ATOS)	Draft of the Table of Contents (ToC) following contributions by ATOS, UPRC, GLA, NEC and RHT.
0.3	09.10.2020	Orlando Avila-García (ATOS)	Writing of Sections 1, 2, 3 and 4. Improvements of Section 8 TME and QoS Evaluation, specifically in Section 8.3 Section 8.4 and 8.5.
0.4	13.10.2020	Orlando Avila-García (ATOS)	A few aesthetic changes
0.5	23.10.2020	Antonio Castillo (ATOS)	Changes in sections 5, 6, 7 and 9 to reflect project progress.
0.6	23.10.2020	Antonio Castillo (ATOS)	Changes in section 6 Dynamic Orchestrator. New section for Realization Engine component.
0.7	26.10.2020	Antonio Castillo (ATOS)	Changes in section 9 TME.
0.8	26.10.2020	Antonio Castillo (ATOS)	Changes in section 7 Dynamic Orchestrator. Numbering corrections
0.9	27.10.2020	Antonio Castillo (ATOS)	Some corrections suggested by the reviewers. Changes in Executive Summary.
1.0	29.10.2020	Antonio Castillo (ATOS)	Added section 7.3.1.1 to describes the CEP integration with the infrastructure building block of BigDataStack.
1.1	30/10/2020	Antonio Castillo Nieto (ATOS)	Final format amendments and release of final version.

Disclaimer

This document contains information that is proprietary to the BigDataStack Consortium. Neither this document nor the information contained herein shall be used, duplicated or communicated by any means to a third party, in whole or parts, except with the prior consent of the BigDataStack Consortium.

Table of Contents

1. Executive Summary 10
2. Introduction 12
2.1. Relation to other deliverables 12
2.2. Relevant aspects unchanged from D3.2 and D3.1 13
2.3. Document structure 14
3. Solution Architecture 15
4. Implementation and Experimentation 16
4.1. Experimental Settings 16
4.1.1. Setting 5: Real-time Recommendation Model Building 16
4.1.2. Setting 6: Batch Recommendation Model Building 16
4.2. Implementation 17
4.3. Experimental Scenarios 19
4.3.1. Scenario 4: Real-time product recommendation analytics cost-readiness 19
4.3.2. Scenario 5: Batch product recommendation analytics throughput 21
5. Cluster Management 24
5.1. Requirements 24
5.2. Design Specifications and Implementation Details 24
5.2.1. Gateway 24
5.2.2. East/West Distributed Load Balancing 25
5.2.3. Cluster Management API extensions: Network Policy Support at Kuryr... 26
5.2.4. Cluster Management API extensions: RWX PVs at OpenShift on OpenStack through Manila support 28
5.2.5. Kubernetization of Kuryr-Kubernetes by adapting CRDs model 28
5.3. Integration Highlights 29
5.4. Experimentation Outcomes 30
5.4.1. Distributed OVN Load Balancer performance 30
5.4.2. Kuryr tuning for real use cases 31
5.4.3. Autoscale experiments through Infrastructure provided APIs 32
6. Realization Engine 34
6.1. Motivation 34
6.2. Requirements 35
6.3. Modular Object Design 38
6.3.1. (BigDataStack) Application 39
6.3.2. (BigDataStack) Object 39
6.3.3. (BigDataStack) Operation 40
6.3.4. (BigDataStack) Operation Sequences 41
6.3.5. (BigDataStack) Events 42
6.3.6. (BigDataStack) Metric 42
6.3.7. (BigDataStack) Service Level Objective 43
6.3.8. (BigDataStack) Resource Template 44
6.3.9. (BigDataStack) Application State 44
6.4. Updated Playbook Formatting 45
6.5. Realization Engine Architecture 46

cc BY NC ND

6.6. Containerized Services 48
6.6.1. Realization Engine and Application API 48
6.6.2. Realization UI 50
6.6.3. Cluster Monitoring 53
6.6.4. Operation Sequence (Container Service) 54
6.7. Generic BigDataStack Operations 54
6.7.1. Instantiate 55
6.7.2. SetParameters 55
6.7.3. GetParameterFromObjectLookup 56
6.7.4. Deploy 57
6.7.5. ExecuteCMD 57
6.7.6. Build 58
6.7.7. Delete 59
6.7.8. Scale 59
6.7.9. Wait 60
6.7.10. WaitFor 60
6.8. Summary 60
7. Dynamic Orchestration 61
7.1. Requirements 61
7.2. State-of-the-Art: RL for Applications' Configuration 63
7.3. Design Specifications 64
7.3.1. Adaptable Distributed Storage and Complex Event Processing Interplay 66
7.3.2. CEP Integration with the Infrastructure building block of BigDataStack ... 67
7.3.3. canYouScale method 67
7.3.4. infrastructureFinishedScaling method 67
7.3.5. infrastructureFinishedScalingDown method 67
7.3.6. Interplay with the Realization Engine 68
7.4. Implementation and Integration Highlights 68
7.5. Experimentation Outcomes 69
7.6. Next Steps 74
8. ADS Ranking & Deploy 75
8.1. Changes Since D3.2 75
8.2. Terminology 76
8.3. Requirements 77
8.4. Design Specifications 82
8.4.1. Updated Architecture 82
8.4.2. Recommend Resources Operation 84
8.4.3. Apply Operation 84
8.4.4. Connection with the Realization UI 85
8.5. ADS Ranking Tier 2 (Machine Learned Ranking) 85
8.5.1. Related Work 85
8.5.2. Modelling Deployment Ranking as a Learning Task 87
8.5.3. Aggregating Across Service Level Objectives 88
8.5.4. Models and Feature Sets 89
8.6. Experimentation Outcomes 91
8.6.1. Dataset Structure 91
8.6.2. Dataset 1: Real-time Data Server (Streaming) 92

CC BY NC ND

8.6.3. Dataset 2: Training a Deep Learning Model (Batch Processing) 94
8.6.4. Metrics 96
8.6.5. Baselines 97
8.6.6. Training Procedure 98
8.6.7. ADS Ranking Performance Results 98
8.7. Summary 100

9. Triple Monitoring & QoS Evaluation 101
9.1. Requirements 102
9.2. Design Specifications 102
9.2.1. TME Scaling and Long-Term Persistence 104
9.3. Experimentation Outcomes 104
9.4. Implementation and Integration Highlights 105
9.5. Conclusions 105

10. Information-Driven Networking 107
10.1. Requirements 107
10.2. Design Specifications 107
10.3. Experimentation Outcomes 110
10.4. Implementation and Integration Highlights 113
10.5. Conclusions 114

11. References 115

List of tables

Table 1 - Data-driven Infrastructure Management capability experimentation phases. .. 18
Table 2 - Data-driven Infrastructure Management capability implementation plan. 19
Table 3 - Requirement (1) for Dynamic Orchestrator 62
Table 4 - Requirement (2) for Dynamic Orchestrator 62
Table 5 - Requirement (3) for Dynamic Orchestrator 62
Table 6 - Requirement (4) for Dynamic Orchestrator 63
Table 7 - SLO satisfaction for Vanilla DQN agent vs. Tutor4RL agent. 72
Table 8 - Requirement (1) for ADS Ranking 77
Table 9 - Requirement (2) for ADS Ranking 78
Table 10 - Requirement (3) for ADS Ranking 78
Table 11 - Requirement (4) for ADS Ranking 79
Table 12 - Requirement (5) for ADS Ranking 79
Table 13 - Requirement (6) for ADS Ranking 80
Table 14 - Requirement (7) for ADS Ranking 80
Table 15 - Requirement (8) for ADS Ranking 80
Table 16 - Requirement (1) for ADS Deploy 81
Table 17 - Requirement (2) for ADS Deploy 81
Table 18 - Requirement (3) for ADS Deploy 81
Table 19 - Requirement (4) for ADS Deploy 81
Table 20 - Requirement (5) for ADS Deploy 82
Table 21 - Requirement (6) for ADS Deploy 82
Table 22 - Realtime Data Server Deployment Ranking Dataset Statistics 93
Table 23 - Statistics for the Deep Learning Deployment Ranking Dataset 95
Table 24 - Deployment Ranking Performance on the Real-time Data Server dataset .... **Error! Bookmark not defined.**
Table 25 - Deployment Ranking Performance on the Deep Learning dataset 99

List of figures

Figure 1: Data-Driven Information Management (DDIM) process. 15
Figure 2: Experimental setting 5 - Real-time analytics process to produce the Product Recommendations table (data flow view) 16
Figure 3: Experimental setting 6 - Batch analytics process to produce the Product Recommendations table (data flow view). 17
Figure 4: Throughput improvements (POD to POD) 31
Figure 5: Throughput improvements (POD to SVC) 31
Figure 6: Realization Engine Application Model 39
Figure 7: Updated BigDataStack Playbook Format 46
Figure 8: Realization Engine Architecture 47
Figure 9: Realization Engine Manager Architecture 48
Figure 10: Realization UI Namespace Overview 51
Figure 11: Realization UI Applications View 52
Figure 12: Realization UI Operation States View 53
Figure 13: Instantiate Operation Configuration 55
Figure 14: Set Parameters Operation Configuration 56

CC BY NC ND

Figure 15: Get Parameters from Object Lookup Operation Configuration 57
Figure 16: Deploy Operation Configuration 57
Figure 17: ExecuteCMD Operation Configuration 58
Figure 18: Build Operation Configuration 59
Figure 19: Delete Operation Configuration 59
Figure 20: Scale Operation Configuration 59
Figure 21: Wait Operation Configuration 60
Figure 22: WaitFor Operation Configuration 60
Figure 23: High level vision of Tutor4RL 65
Figure 24: Register with Dynamic Orchestrator Operation Configuration 68
Figure 25: Example of streaming analytics application 69
Figure 26: Comparison of performance between Tutor4RL and a plain DQN agent. 71
Figure 27: DO performance to manage 2 SLOs: costPerHour < 0.03 and responseTime < 200 73
Figure 28: Guide (#1 and #2) and constrain (#3) functions for DO. 74
Figure 29: ADS-Ranking and ADS-Deploy Processing Architecture 83
Figure 30: Cost Per Hour Delta Scoring Function 88
Figure 31: Realtime Data Server Architecture 92
Figure 32: Deep Learning Architecture 94
Figure 33: Evaluation by data points 101
Figure 34: Evaluation by percentile 102
Figure 35: Architecture of the Triple Monitoring Engine 103
Figure 36: Example of configuration of TME & QoS Evaluation for experimental setting 5 for scenario 4. 104
Figure 37: Triple Monitoring Engine & QoS Evaluation integrations. 105
Figure 38: Proxy Pod prioritizes Weighted Load Traffic to the Producer App 108
Figure 39: Initialization of the Proxy Pod which splits the events 109
Figure 40: An indicative network policy definition for controlling HTTP GET/POST requests 110
Figure 41: Mapping of Information-Driven Networking tool with BDS Use Cases 111
Figure 42: Producer logs according to the event type 112
Figure 43: Service Mesh Health Check through Kiali 113
Figure 44: Total requests collected by Prometheus 113

CC BY NC ND

Acronyms

ADS	Application and Data Services
ADW	Application Dimensioning Workbench
AWS	Amazon Web Services
CD	Continuous Delivery
CDP	Candidate Deployment Pattern
CEP	Complex Event Processing
CI	Continuous Integration
CNI	Container Network Interface
CRD	Kubernetes Custom Resource Definition
DDIM	Data-Driven Infrastructure Management
DNS	Domain Naming System
DO	Dynamic Orchestration
EKS	AWS Elastic Kubernetes Service
GCP	Google Cloud Platform
KPI	Key-Performance Indicators
K8S	Kubernetes
LbaaS	Load Balancer as a Service
OKD	OpenShift Origin Kubernetes Distribution
OVN	Open Virtual Networking
OVS	Open Virtual Service
QoS	Quality of service
QoSE	Quality of service evaluation
RL	Reinforcement Learning
RPS	Request per second
TME	Triple Monitoring Engine
SDN	Software-Defined Network
SLA	Service-Level Agreement
SLO	Service-Level Objective
NIC	Network Interface Controller
VM	Virtual Machine

1. Executive Summary

This is the Scientific Report and Prototype Description (Y3), reflecting the work done in the scope of the Data-Driven Infrastructure Management (DDIM) capability of the overall BigDataStack environment, including Cluster Management, Dynamic Orchestration, Realization Engine (known as ADS Ranking & Deploy in Y1 and Y2), Triple Monitoring & QoS Evaluation, and Information-Driven Networking. For all components, additional requirements have been identified and refined, new design solutions have been proposed, experimentation has been conducted and evaluation results have been collected for the respective implementations.

Cluster Management has been improved to provide extended APIs to manage the Infrastructure and Applications (such as Network Policies for fine-grain network access tuning for the applications), and to improve the performance such as the support for distributed load balancing for East/West traffic; speed up on the control plane actions (services creation time); resource consumption savings (remove the need of having a VM per service); and Read Write Many (RWX) support to enable pods sharing the same volumes.

The Realization Engine is a new component of the BigDataStack platform that has been developed as an addition to Task 3.3 (Dynamic Deployment Patterns & Runtime Re-Configuration). The goal of the Realization Engine is to provide a central suite of containerized services that enable configuration, deployment and subsequent management of user applications and their components.

The Dynamic Orchestrator has also been improved to make our Reinforcement Learning (RL) algorithm more flexible and adequate to deal with the complexity needed for orchestrating BigDataStack applications. We have developed a novel Reinforcement Learning-based approach called Tutor4RL, which combines domain knowledge with machine learning for achieving a good initial performance, a common problem in RL and in particular for DQN. We have introduced constrain functions, to supervise the behavior of the agent at every point, avoiding unnecessary and incorrect changes in the deployment of applications. Finally, we have integrated the DO with the Data-as-a-Service layer of BigDataStack, which contains stateful components such as the Adaptable Distributed Storage (ADS) and the Complex Event Processing (CEP), for adapting these components dynamically during runtime.

Both ADS Ranking and ADS Deploy were subject to significant updates to factor in the new Realization Engine component of BigDataStack, as well as better integrate them with that component via operations. The ADS Ranking component was updated to add support for supervised ranking via learning to rank, demonstrating that it is able to produce effective rankings of deployments for the user across two categories of application.

TME and QoSEvaluator was testing and evaluating in real-world conditions with high scalability and availability. This results in the integration with Thanos1, and the open source and highly available Prometheus setup with long term storage capabilities. This has allowed

[1] https://thanos.io/

cc BY NC ND

us to go one step further (beyond Prometheus) in the integration within the cloud-native foundation ecosystem.

The development focus with respect to the Information-driven Networking includes the deployment and configuration of Istio service mesh with sidecar injection enabled which is exposed through the telemetry application of Kiali Dashboard. Furthermore, service mesh interactions are recorded by the Triple Monitoring and QoS Evaluation which facilitates to validate and realize the enforced network policies and therefore prioritize traffic through weighted load balancing, perform access controls and limit traffic rates across diverse protocols and runtimes.

CC BY NC ND

2. Introduction

This deliverable presents the *Scientific Report and Prototype Description* of Data-Driven Infrastructure Management (DDIM) capability for Y3 of the BigDataStack Project, specifically, work done under the WP3. The document presents the improvements in the designs and implementations of the main components of the DDIM to give support to advance experimental settings and scenarios addressed in Y3. Like in Y2, a particular focus has been put on the Machine Learning (ML) algorithms used to bring data-driven decisions and actions to infrastructure operations, located in the Dynamic Orchestrator (DO) and the Realization Engine (formerly known as ADS Ranking & Deploy), respectively. The rest of components play a supportive role within the DDIM capability and are implemented over well-known CNCF (Could Native Computing Foundation)[2] open source projects in the cloud-native ecosystem: Cluster Management, based on *Kubernetes* (OpenShift distribution), Information-Driven Networking, based on *Istio* service mesh framework, and Triple Monitoring and QoS Evaluation, based on *Prometheus* monitoring system.

2.1. Relation to other deliverables

This document is related to the following past and immediately upcoming deliverables:

- D2.6 – Conceptual model and Reference architecture III (M30). The description of the high-level architecture of BigDataStack as well as the interplay and integration between the main components. The architecture of the Data-Driven Infrastructure Management as well as the design of the components have been devised to fit into the overall architecture.
- D2.3 – Requirements & State of the Art Analysis III (M22). The specification of BigDataStack requirements is centralized in this deliverable. **Only modifications in the requirements of DDIM components have described in this deliverable, in the component-specific subsections.**
- D3.2 – WP3 Scientific Report and Prototype Description – Y3 (M23). It described the solution as well as the experimental results produced in Y2. D3.3 presents the results obtained in Y3, which are necessarily an increment or refinement with respect to those presented in D3.2. **Therefore, please note those aspects of the solution that did not change during Y3 will be referred to in either D3.2 (Y2) or D3.1 (Y1) reports.**
- D4.3 – WP4 Scientific Report and Prototype Description – Y3 (M34). D4.3 makes references to some of the requirements and components which are designed, implemented and experimented with at WP4, while also the D4.3 references and raises requirements that are being described in the current document. In fact, the Data-Driven Infrastructure Management is meant to provide infrastructure services (Infrastructure-as-a-Service) to those components.
- D5.3 – WP5 Scientific Report and Prototype Description – Y3 (M34). D5.3 makes references to some of the requirements and components which are designed, implemented and experimented with at WP5; this is because the tools developed at

[2] https://www.cncf.io/

cc BY NC ND

WP5 will interact with the services and resources provided by the infrastructure to implement certain functionality supporting the different BigDataStack stakeholders.

- D6.2 – Use case description and implementation – Y3 (M34). D3.3 describes partially the use cases which are subject to the experimental validation carried out in Y3. Refer to this deliverable for more details about the use case applications, their motivation, requirements, development and results.

2.2. Relevant aspects unchanged from D3.2 and D3.1

As described in the previous section, **this deliverable presents the Scientific Report and Prototype Description for Y3 for the work done in WP3. Therefore, much of the development and research work reported in this deliverable is a continuation or extension of the work reported in an equivalent report for Y1 (D3.1) and Y2 (D3.2).**

However, in order to avoid the duplication of content, those aspects of the work which have remained unchanged for the last year are not reported again here but property referred to in D3.2 and D3.1. This is the case for:

i. Unchanged from D3.1:
   - The Solution Architecture (Section 3), including the architecture vision, assumptions, platform roles, example scenarios and the high-level design of the Data-Driven Infrastructure Management (DDIM) capability.
   - Experimental settings 1 and 2—included for the reader's convenience.
   - Experimental scenarios 1—included for the reader's convenience.

ii. Unchanged from D3.2:
   - Experimental settings 3 and 4—included for the reader's convenience.
   - Experimental scenarios 2 and 3—included for the reader's convenience.
   - The requirements specification of four out of the five building blocks of the architecture remained unchanged for the most part: Cluster Management (Section 5), Dynamic Orchestrator (Section 6), Realization Engine (Section 7), and Information-Driven Networking (Section 9).
   - The design specification of four out of the five building blocks of the architecture remained unchanged for the most part: Cluster Management (Section 5), Dynamic Orchestrator (Section 6), Triple Monitoring Engine and QoS Evaluation (Section 8), and Information-Driven Networking (Section 9).
   - Global experimentation outcomes at M23 reported in Section 10 regarding deployment and dynamic adaptation of one kind of big data analytics: product recommendation systems—the Connected Consumer (WDL) use case was taken as experimental subject.

CC BY NC ND

2.3. Document structure

The document is structured as follows: Section 3 describes the solution architecture of the Data-Driven Infrastructure Management (DDIM) capability of BigDataStack. Section 4 reports the Implementation and Experimentation: Starting with the experimental settings (Section 4.1), it describes the DDIM capability development roadmap giving support to the research (Section 4.2), and then finalizes with the description of experimental scenarios (Section 4.3).

The following five sections are dedicated to the requirements specification, design specifications, the presentation of experimental results, the description of interesting aspects of the implementation and integration of the component within the whole architecture, and some next steps: Cluster Management (Section 5), Dynamic Orchestration (Section 6), ADS Ranking & Deploy (Section 7), Triple Monitoring & QoS Evaluation (Section 8) and Information-Driven Networking (Section 9).

CC BY NC ND

3. Solution Architecture

For a full description of the technical solution for the Data-driven Infrastructure Management (DDIM) capability architecture, please refer to D3.1 (WP3 Scientific Report and Prototype Description – Y1) and D3.2 (WP3 Scientific Report and Prototype Description – Y2).

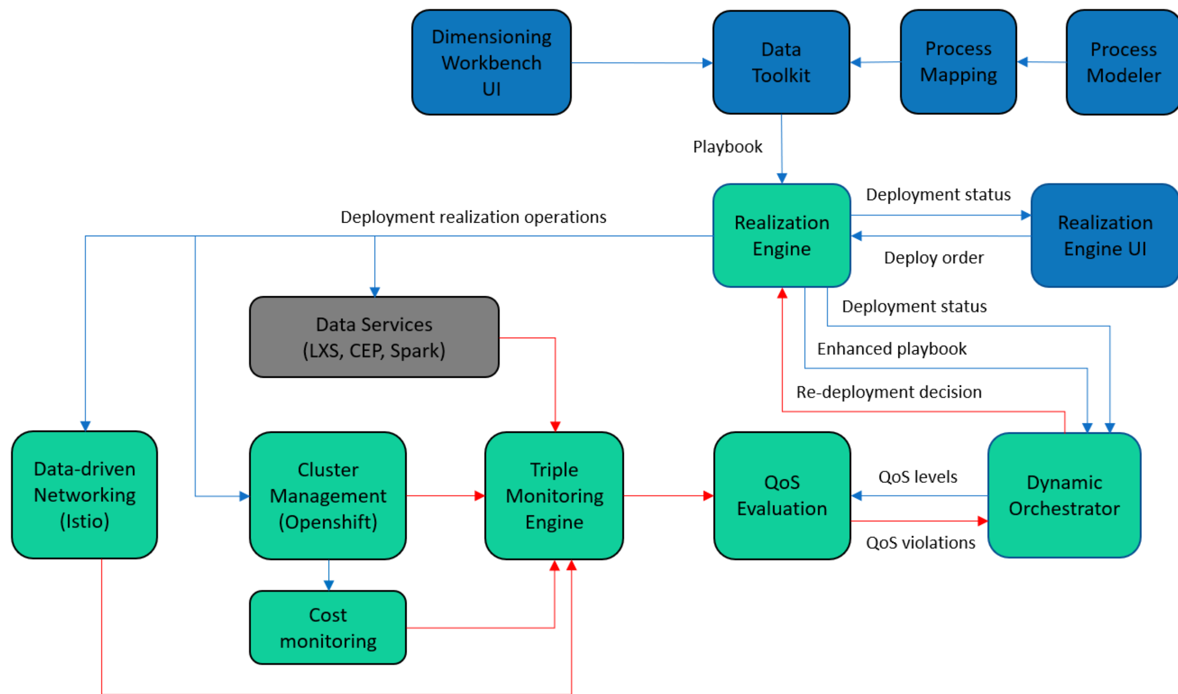


Figure 1: Data-Driven Information Management (DDIM) process.

Figure 1 shows the standard Data-driven Infrastructure Management (DDIM) process, including the flow of the main messages, as a collaboration between components inside and outside the DDIM: In green, components belonging to the DDIM capability (developed in WP3); in grey, components belonging to the Data as a Service capability (developed in WP4); in blue, components of the GUI (developed in WP5).

Main DDIM building blocks are: 1) Cluster Management (WP3-T3.1) based on OpenShift container orchestration platform running on either OpenStack infrastructure-as-a-service (IaaS) or bare metal; 2) Realization Engine (known previously as ADS-Ranking & Deploy in D3.1 and D3.2, WP3-T3.3) as the self-optimized deployment realization service; 3) Dynamic Orchestration (WP3-T3.2) providing runtime adaptation of big data analytics applications and services; 4) Data-Driven Networking (WP3-T3.4), based on a serve mesh model to enforce networking policies of application and data services; and 5) Triple Monitoring and QoS Evaluation (WP3-T3.5) providing monitoring and QoS checks for big data analytics systems at different levels (application, data, networking and cluster resources).

Apart from those 5 main DDIM building blocks, the cost monitoring (Cost Estimation component of Realization Engine) provides support for the Triple Monitoring Engine and this component belong to T3.3

bigdatastack.eu

4. Implementation and Experimentation

This section introduces the **new experimental settings and scenarios** WP3 developed in the Y3 phase, to answer important questions and validate certain hypothesis to develop the Data-Driven Infrastructure Management (DDIM) capability.

4.1. Experimental Settings

In the following sections, we describe the **new experimental settings** developed in Y3, supporting experiments with big data analytics systems with an increasing level of complexity with respect to Y1 and Y2, in terms of the number of use case application components as well as BigDataStack Platform components engaged. **Please refer to D3.2 for a description of experimental settings from 1 to 4.**

4.1.1. Setting 5: Real-time Recommendation Model Building

This setting deploys the real-time analytics process which keeps the Product Recommendations table up to date in between runs of the batch analytics process. This process is made of two services as shown by Figure 2.

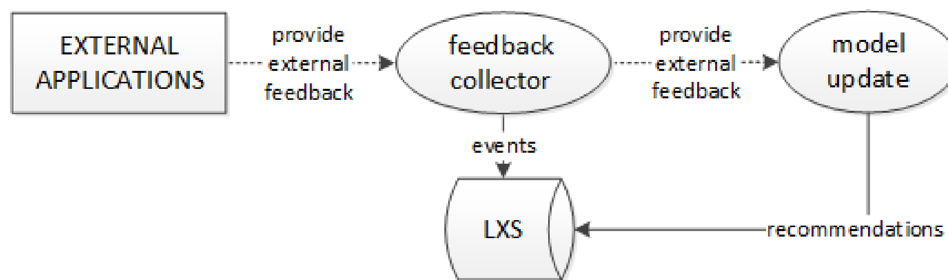


Figure 2: Experimental setting 5 - Real-time analytics process to produce the Product Recommendations table (data flow view)

The product recommendation real-time analytics process is split into two main services: *Feedback Collector*, which receives behavioural events of users visiting the EROSKI's ecommerce web site, and *Model Update*, which runs the real-time event streaming analytics that carry out incremental changes to the Product Recommendations table stored in LeanXcale database (data flow view). The actual analytics is made by the Batch Recommendation. Model Update runs as a Spark Streaming job. It keeps the recommendation table up to date in between batch analytics process runs (see D6.2 for more details of this process).

4.1.2. Setting 6: Batch Recommendation Model Building

This setting deploys the batch analytics process which produces the Product Recommendations table used by the Recommendation Provider service to run its recommendation inference. This process runs as a Spark job, and it is made of several analytics services as shown by Figure 3 (see D6.2 for more details of this process).

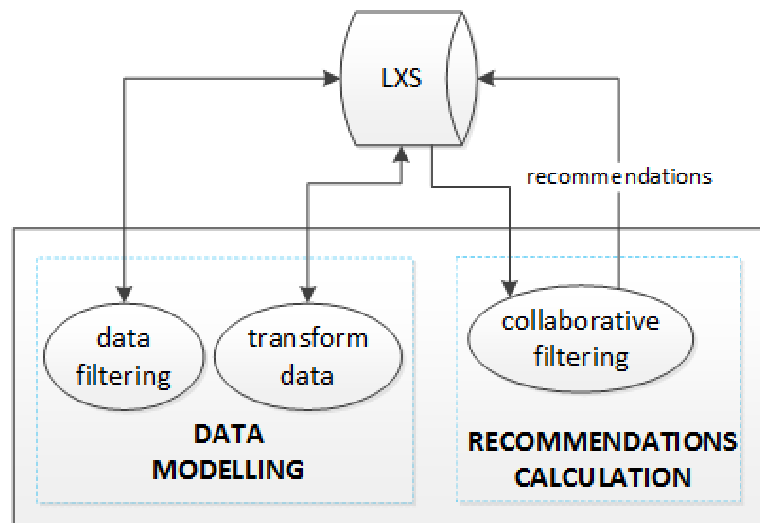


Figure 3: Experimental setting 6 - Batch analytics process to produce the Product Recommendations table (data flow view).

The product recommendation batch analytics process is split into three main services, *Data Filtering*, *Transform Data* and *Collaborative Filtering*, and returns Product Recommendations table stored in LeanXcale database (see D6.2 for more details of this process).

4.2. Implementation

Table 1 summarizes the experimentation (evaluation and validation) plan for the Data-driven Infrastructure Management capability between M24 and M36:

	M24	M30	M36
Milestone	Performance Optimization with settings 1, 2, 3, 4	Settings 5 and 6 Implementation and Testing	Scenarios 5 and 6 Validation and Optimization
Objective	WP3, WP4 and WP5 components as well as their collaboration optimized to provide cost-effective orchestrated capabilities for scenarios 1, 2 and 3.	WP3, WP4 and WP5 implement changes needed to support big data real-time and batch analytics processes outlined in settings 5 and 6, respectively.	WP3, WP4 and WP5 components as well as their collaboration optimized to provide cost-effective orchestrated capabilities for scenarios 4 and 5.
Success criteria	ALL WP3, WP4 and WP5 services are fully integrated and deployed on Kubernetes, validated in scenarios 1, 2 and 3.	ALL WP3 services are deployed and running on Kubernetes to implement experimental settings 5 and 6.	ALL WP3, WP4 and WP5 services are fully integrated and deployed on Kubernetes, validated in scenarios 4 and 5.
	Experimentation with Setting 3 and 4	Experimentation with Setting 1	Experimentation with Setting 2 and 3

Table 1 - Data-driven Infrastructure Management capability experimentation phases.

Table 2 summarizes the Data-driven Infrastructure Management capability implementation roadmap for Y3:

	M24	M30	M36
Experimental setting supported	1, 2, 3, 4	5, 6	5, 6
Experimental scenario enacted	1, 2	3, 4	4, 5
Cluster Management	OpenStack integration, Cluster performance improvements, Operators, Gateway East/West Distributed Load Balancing	- Complete the E/W Distributed LoadBalancing support (not only the kuryr part, but also the integration into OpenShift) - Network Policy Implementation in Kuryr - Extra fixes/improvements on the OpenStack and Operators integration	- Network Policy testing coverage and bug fixes - Manila support to enable pods with RWX volumes - Kuryr-kubernetes modernization through CRD model adoption
Dynamic Orchestrator	Agent Interpreter ADS Interplay	Tutor4RL: RL framework for combining ML with external domain knowledge Implementation of guides and constrains to express domain knowledge in Tutor4RL Improvements and bug fixes in interactions with components in WP3	Refinements for Tutor4RL, including guide and constrain functions Refinements for orchestration of multiple applications with different metrics, SLOs and actions simultaneously Changes in implementation of interactions with Realization Engine, CEP and ADS, including the implementation of REST-RabbitMQ proxy for communication

CC BY NC ND

			between DO and components outside WP3 that utilize REST APIs
Ranking & Deployment	ADS-Ranking, ADS-Deploy Global Decision Tracker		
Triple Monitoring Engine & QoS Evaluation	Prometheus, Graphana, Metrics at application, data, resources cluster and networking levels, Manager, QoS evaluation QoS evaluation proxy Resource Cluster metrics	QoS Evaluation Confidence Levels	Triple Monitoring Engine (TME) Scaling, Long-Term Persistence
Information-driven Networking	Kubernetes Networking & Policies Enforcement, Istio	- Implementation of the Istio weighted load balancing mechanisms - Integration with Kiali and Triple Monitoring Engine and QoS Evaluation - Implementation of different network policies coupled with the Demos	- Network policy testing coverage and bug fixes - Access control and policy enforcement

Table 2 - Data-driven Infrastructure Management capability implementation plan.

4.3. Experimental Scenarios

This section explains the **new experimental use case scenarios**, including success criteria, developed in Y3 to be used in the context of WP3 to verify and validate the behavioural invariances of DDIM components in order to ensure trustworthy run of component-specific experiments. **Please refer to D3.2 for a description of experimental scenarios from 1 to 3.**

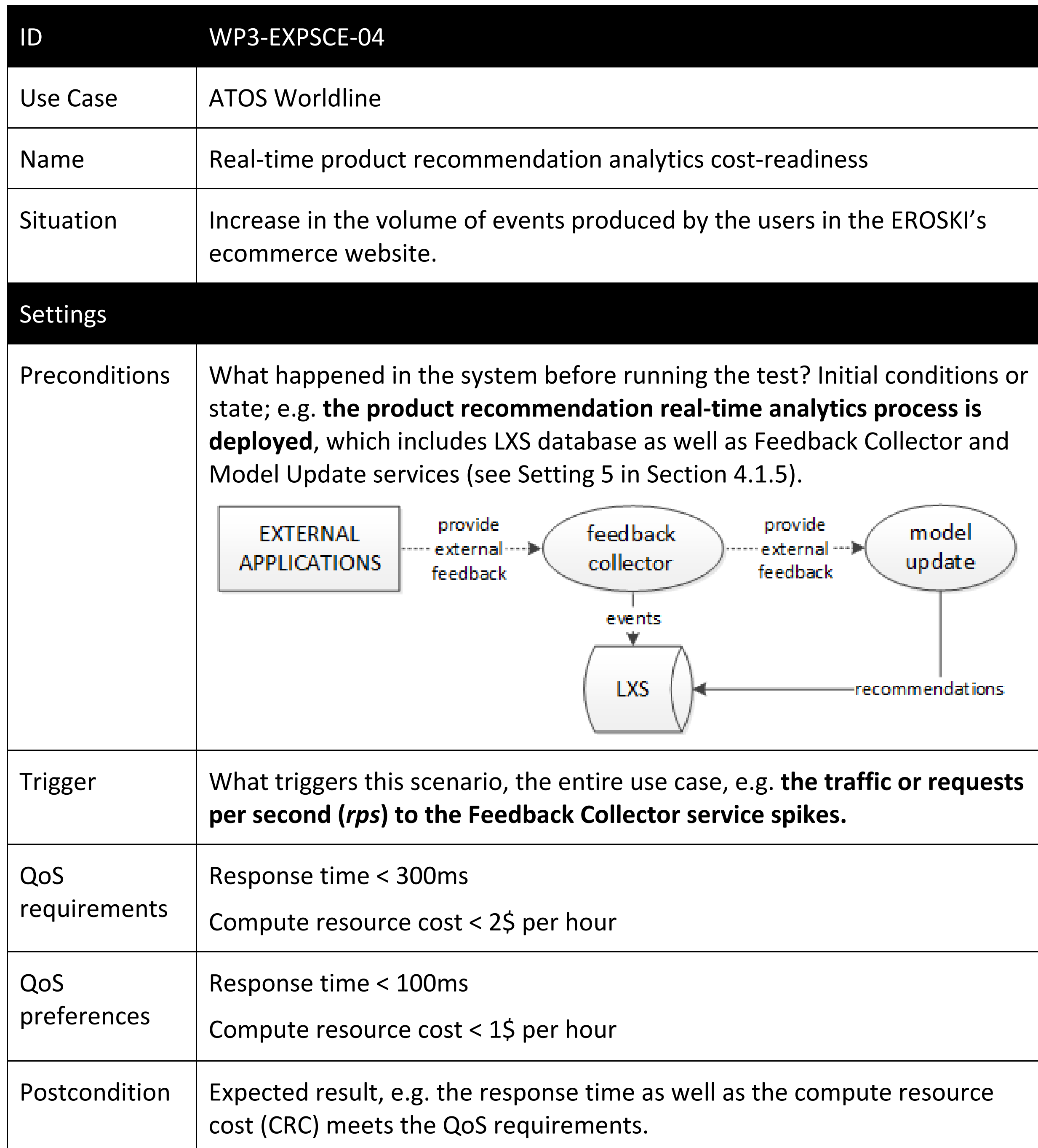
4.3.1. Scenario 4: Real-time product recommendation analytics cost-readiness

This scenario deals with the real-time analytics of behavioural event streams of users visiting the EROSKI's ecommerce website. The analytics process makes incremental updates to the

CC BY NC ND

Product Recommendations table based on micro-batches of events, to keep the table up to date in between runs of the batch analytics process.

In this scenario, the Data Scientist is concerned with the time to value of certain type of events, in particular, to speed up the changes the Product Recommendations table need to undergo in the impact of events of type PRODUCT_RECOMMENDATION_REMOVED[3] so that the recommender is not recommending anymore a product that has been rejected to a given user.



ID	WP3-EXPSCE-04
Use Case	ATOS Worldline
Name	Real-time product recommendation analytics cost-readiness
Situation	Increase in the volume of events produced by the users in the EROSKI's ecommerce website.
Settings	
Preconditions	What happened in the system before running the test? Initial conditions or state; e.g. **the product recommendation real-time analytics process is deployed**, which includes LXS database as well as Feedback Collector and Model Update services (see Setting 5 in Section 4.1.5). EXTERNAL APPLICATIONS – provide external feedback → feedback collector – provide external feedback → model update feedback collector – events → LXS model update – recommendations → LXS
Trigger	What triggers this scenario, the entire use case, e.g. **the traffic or requests per second (*rps*) to the Feedback Collector service spikes.**
QoS requirements	Response time < 300ms Compute resource cost < 2$ per hour
QoS preferences	Response time < 100ms Compute resource cost < 1$ per hour
Postcondition	Expected result, e.g. the response time as well as the compute resource cost (CRC) meets the QoS requirements.

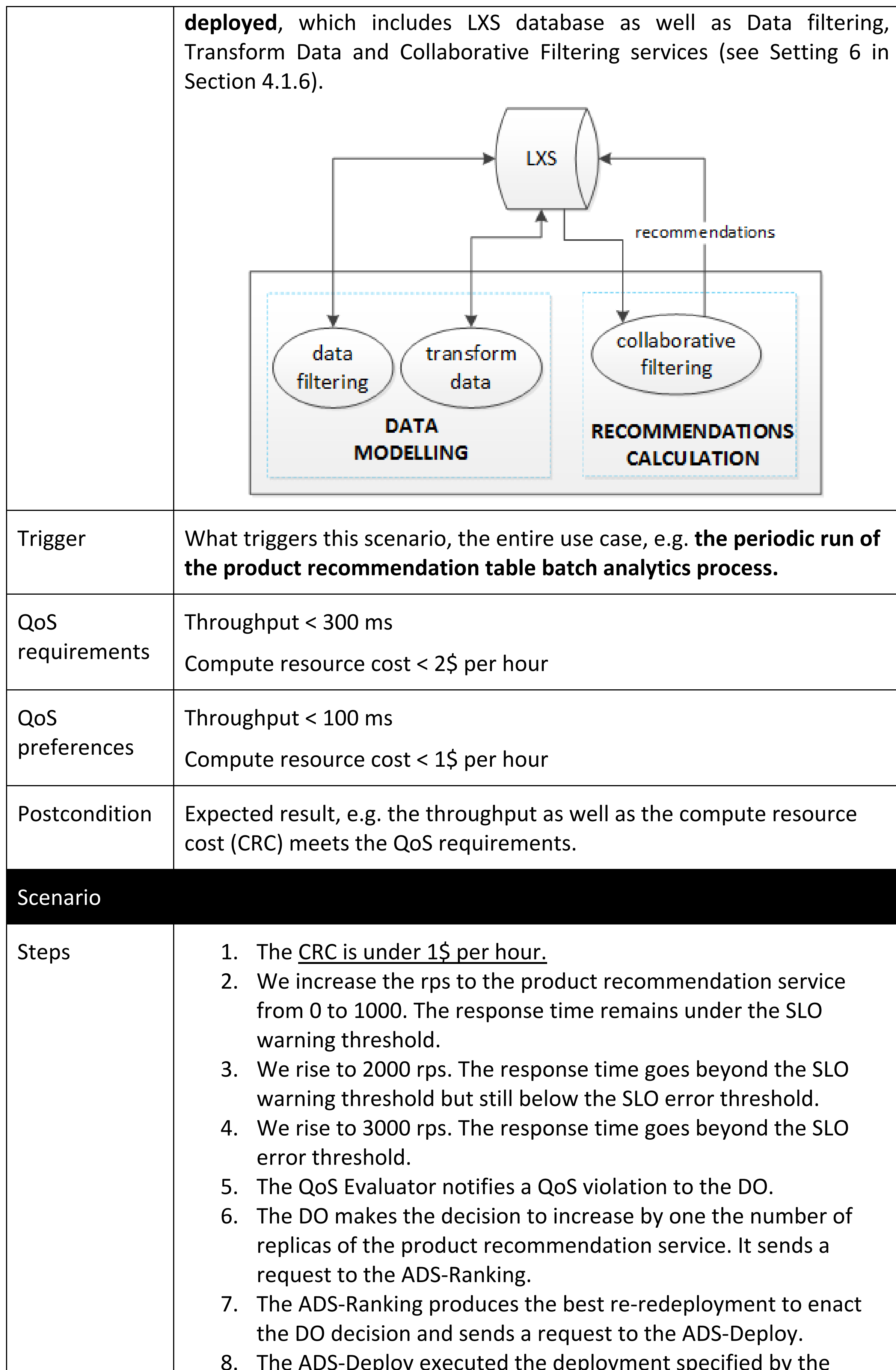
[3] This event informs a given user has explicitly expressed through a click that she does not find a given recommendation interesting.

Scenario	
Steps	1. The CRC is under 1$ per hour. 2. We increase the rps to the product recommendation service from 0 to 1000. The response time remains under the SLO warning threshold. 3. We rise to 2000 rps. The response time goes beyond the SLO warning threshold but still below the SLO error threshold. 4. We rise to 3000 rps. The response time goes beyond the SLO error threshold. 5. The QoS Evaluator notifies a QoS violation to the DO. 6. The DO makes the decision to increase by one the number of replicas of the product recommendation service. It sends a request to the ADS-Ranking. 7. The ADS-Ranking produces the best re-redeployment to enact the DO decision and sends a request to the ADS-Deploy. 8. The ADS-Deploy executed the deployment specified by the ADS-Ranking by sending request to the cluster manager (OpenShift). 9. OpenShift increases by one the number of replicas of the product recommendation pod. 10. The response time of the product recommendation service drops below the SLO warning threshold. 11. The CRC rises to 1.5$ per hour so beyond the SLO warning threshold but still below the SLO error threshold.

4.3.2. Scenario 5: Batch product recommendation analytics throughput

This scenario represents a situation where the application suffers a traffic spike that obligates the DDIM to scale out the application deployment so to keep its response time at certain SLO, like in Scenario 1, but adding a second SLO to ensure operational costs remain under certain threshold. Thus, this scenario exemplifies how the operational "cost" can be managed and enforced as just another SLO or QoS attribute by the DDIM.

ID	WP3-EXPSCE-05
Use Case	ATOS Worldline
Name	Cost-effectiveness of the product recommendation service
Situation	Spike in the volume of traffic (requests per second - *rps*) to the online serving layer of the product recommendation system.
Settings	
Preconditions	What happened in the system before running the test? Initial conditions or state; e.g. **the product recommendation batch analytics process is**



	deployed, which includes LXS database as well as Data filtering, Transform Data and Collaborative Filtering services (see Setting 6 in Section 4.1.6).
Trigger	What triggers this scenario, the entire use case, e.g. **the periodic run of the product recommendation table batch analytics process.**
QoS requirements	Throughput < 300 ms Compute resource cost < 2$ per hour
QoS preferences	Throughput < 100 ms Compute resource cost < 1$ per hour
Postcondition	Expected result, e.g. the throughput as well as the compute resource cost (CRC) meets the QoS requirements.
Scenario	
Steps	1. The CRC is under 1$ per hour. 2. We increase the rps to the product recommendation service from 0 to 1000. The response time remains under the SLO warning threshold. 3. We rise to 2000 rps. The response time goes beyond the SLO warning threshold but still below the SLO error threshold. 4. We rise to 3000 rps. The response time goes beyond the SLO error threshold. 5. The QoS Evaluator notifies a QoS violation to the DO. 6. The DO makes the decision to increase by one the number of replicas of the product recommendation service. It sends a request to the ADS-Ranking. 7. The ADS-Ranking produces the best re-redeployment to enact the DO decision and sends a request to the ADS-Deploy. 8. The ADS-Deploy executed the deployment specified by the

cc BY NC ND

	ADSRanking by sending request to the cluster manager (Openshift). 9. Openshift increases by one the number of replicas of the product recommendation pod. 10. The response time of the product recommendation service drops below the SLO warning threshold. 11. The CRC rises to 1.5$ per hour so beyond the SLO warning threshold but still below the SLO error threshold.

cc BY NC ND

5. Cluster Management

The cluster management component's responsibilities are both to deploy the BigDataStack components as requested and to keep its status healthy overtime. This not only includes containers but also the related services and even the OpenShift Kubernetes cluster itself by exposing the needed Infrastructure APIs. The cluster management is in charge of adapt the current deployments to the new preferred status requested by the upper layers, in order for example to increase the size of the cluster, or scale up/down given applications. In addition, during the last year of the project more improvements has been done at the cluster management internals to provide extended APIs to manage the Infrastructure and Applications (such as Network Policies for fine-grain network access tuning for the applications), and to improve the performance such as the support for distributed load balancing for East/West traffic; speed up on the control plane actions (services creation time); resource consumption savings (remove the need of having a VM per service); and Read Write Many (RWX) support to enable pods sharing the same volumes.

5.1. Requirements

The main change comparing to the requirements documented in D3.2 is described below.

	Id	Level of detail	Type	Actor	Priority
	REQ-CM-0X	System	ENV	Developer	DES
Name	Manila Support at OpenShift on OpenStack: Enable ReadWriteMany (RWX) PVs when running OpenShift cluster on top of OpenStack VMs				
Description	Most of the OpenStack Cinder backend drivers do not support the attachment of volumes to multiple VMs. This means that pods running inside different OpenShift nodes (aka VMs) cannot access the same Volume (i.e., the same PV/PVC). To avoid this problem we need to add a new operator in charge of installing and configuring the needed operators/controllers to make use of Manila (instead of Cinder) as a storage class that pods can use to get their PVs attacked.				
Additional Information	This is needed by some applications that may require access to shared block storage, not just object storage. Note Manila is an OpenStack project whose main objective is to create shared NFS as a service				

5.2. Design Specifications and Implementation Details

The design for this component (specified in Section 5 of D3.1) has mostly remained valid for Y2. The following sections describe the aspects of the design that have been changed.

5.2.1. Gateway

The gateway for the BigDataStack engine can also be implemented as part of OpenShift, in 2 different ways depending on the final requirements:

- By using OpenShift routes: Route is a way to expose OpenShift services by giving it an externally reachable hostname, like www.example.com. It has the option to perform the routing based on paths, i.e., we can use it to redirect some queries to

the CEP component (i.e., www.example.com/cep/…) and others to the Alarm component (i.e., www.example.com/alarms/...). The initial design targets to use this, being able to assign a common OpenStack Floating IP for all the ingress traffic to OpenShift Apps, in this case BigDataStack components.

- By using Istio service mesh: A service mesh is a network of microservices that enables applications and the interactions among them. It offers functionality like load-balancing, fine grain traffic control, access control, logging, tracing, etc., through *sidecars* containers associated to the applications pods. One offered functionality is Istio-Gateways which controls the exposure of services at the edge of the mesh. This could be used to tie gateways to specific virtual services that can perform the extra required actions that the gateway may require besides redirecting the traffic to the desired endpoint.

Even though we are also using Istio service mesh internally, for the Cluster Gateway we are using the OpenShift Ingress, I.e, the first option with the routes where a single public IP is used for accessing all the applications by leveraging the OpenShift route support and the services k8s models to expose applications.

5.2.2. East/West Distributed Load Balancing

In Kubernetes and OpenShift, the communication between the different application components and between applications (i.e., between the Pods) is not meant to be *pod* to *pod* (and using IPs) since pods are supposed to be disposable and therefore they can be replaced/deleted at any time. Pods are usually behind a service which abstracts the IP/name of the container(s) that is pointing to. This way, pods can talk to known services IPs (and names) and containers after that service can be recreated at any time without impacting the way the caller pods uses to reach them.

Given the above, the pod to svc to pod communication performance is quite important as it is the most usual pattern. When using Kuryr, Services are implemented as Octavia load balancers. This means that each K8s service will require Octavia load balancer, and with the default ‘amphora’ driver that means an OpenStack VM. This has 4 main implications:

1. Resource waste since lots of VMs will be needed for backing the services.
2. User experience as services will need more time to be up and running since the amphora VM must be created and configure.
3. Single point of failure for services as if the VM dies, a new one will need to be created to replace it.
4. Network latency as traffic needs to do extra hops to reach the amphora VM.

For these reasons, we have worked on the integration of OVN load balancer into OpenStack, including Octavia and Kuryr. The OVN load balancer is a distributed load balancer based on OVS/OVN flows. This means that it does not require any amphora VM to load balance the traffic and simply creates the needed flows locally on each OpenStack compute node. To make it easier to understand, it is like if an *iptable* rule was changing the Kubernetes service IP by one of the Kubernetes endpoints (pods) IPs and then the traffic was directly forwarded to the selected pod.

Thanks to this integration, the next advantages have been achieved:

 CC BY NC ND

- Speed up on the time needed to create a K8s service. Now it is more similar to bare metal OpenShift/Kubernetes deployment (with kube-proxy) and it will take only a few seconds instead of around 1 minute
- Resource savings: There is no need for extra resource when creating services. This approach used ovs flows instead of amphora VMs. In fact, this solution is also more scalable than kube-proxy solution based on iptables
- Due to the OVN load balancer distributed nature (flows in every node instead of a VM somewhere) there is no single point of failure
- Reduced latency with increased throughput: Distributed routing as the traffic goes directly pod to pod instead of having to jump to the OpenStack node that has the amphora VM and back
- No need to parse Security Groups at amphora load balancer to apply Kubernetes Network Policies, with the consequent reduction on Neutron OpenStack load, as well as in the time needed to enforce those policies.

This feature makes Kuryr-kubernetes a project much more appealing for companies and for different use cases, where having a VM per service was an impediment to use it, not only for the extra penalties on control and data plane performance, but mainly due to the excessive number of resources needed for the amphora VMs.

5.2.3. Cluster Management API extensions: Network Policy Support at Kuryr

Some applications need fine grain traffic control at the IP address or port level (OSI layer 3 or 4). For this reason, Kubernetes Network Policies API was defined. Network Policies are an application centric construct which allow you to specify how a pod is allowed to communicate (ingress and egress) with various network entities over the network. The entities the pod can communicate with are identified through a combination of the following, which is fully based on Kubernetes labels:

- Other pods that are allowed
- Namespaces that are allowed
- IP Blocks

In order to be able to provide this cluster management API for network management at Kuryr-Kubernetes, we need Kuryr to:

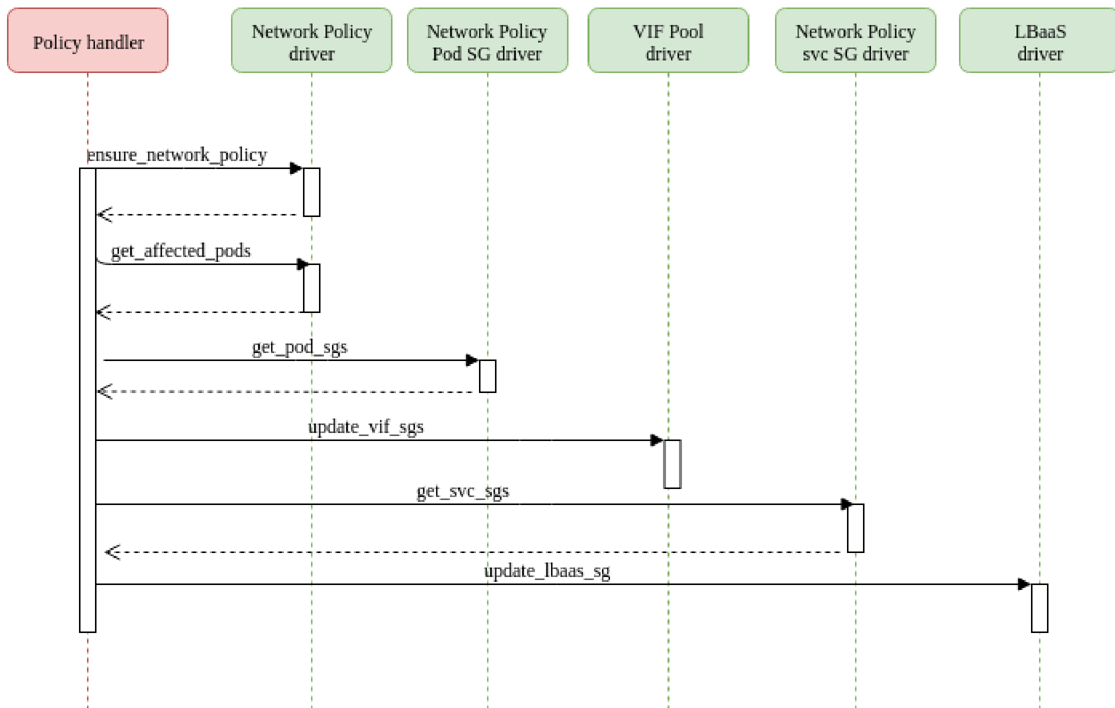
- React to the Network Policy objects creation/deletion/updates and process them
- Generate the corresponding OpenStack resources that provide the same isolation between pods and services as expected by the Network Policies. The OpenStack resources available for this are the Neutron Security Groups and Security Group Rules.

This is one of the largest and hardest features merged on upstream Kuryr-Kubernetes, with a lot of implications, such as new handlers and drivers, as well as with modifications to almost any existing handler to react to them. There have been more than 50 commits

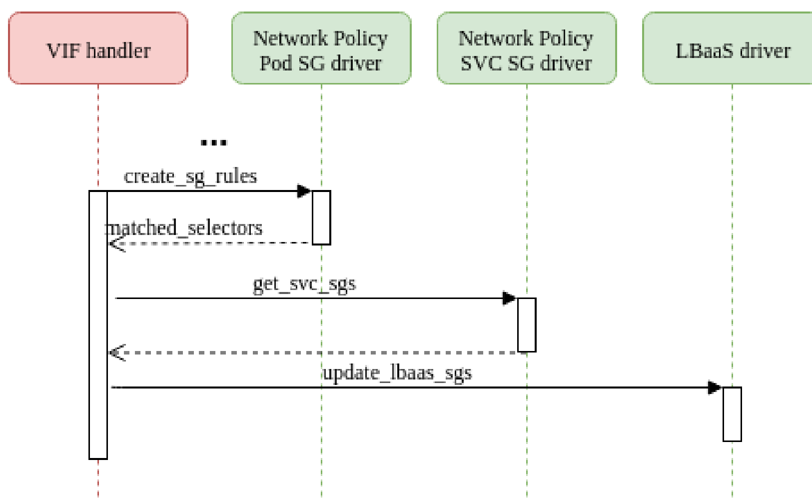
cc BY NC ND

upstream related to it between the different pieces for the feature, the extra testing code and the follow up bug fixed due to uncover corner cases. For more information about the proposed solution as well as the new handlers and drivers, and the modifications to the existing ones, you can check the developer reference document that was created for it at: https://github.com/openstack/kuryr-kubernetes/blob/master/doc/source/devref/network_policy.rst

The flow that Kuryr drivers/handlers follow when a new network policy is created is the following:



As it can be seen, the new policy handlers not only have to reach out the new network policy drivers (3 of them) but also existing drivers such as the vif pool (for pods creation) and the lbaas driver (for services creation). Changes are not only limited to that, but also other handlers are affected, such as the vif handler (the one in charge of creating the resources needed for the pods):



5.2.4. Cluster Management API extensions: RWX PVs at OpenShift on OpenStack through Manila support

Applications that need to maintain some state or data (Pods) need access to persistent volumes (PVs). The default backend for those when running OpenShift on top of OpenStack is Cinder. However, Cinder does not have support for multi-attachment of volumes to different VMs (at least not all the cinder backend drivers). This impose some limitations on the applications running on top as each one will need to get them on PV/PVC and therefore both of them won't be able to read/write from the same one.

In order to avoid this limitation, we have added support for Manila at OpenShift. Manila is an OpenStack project that derived from Cinder project, and that provides canonical storage provisioning control plane for shared or distributed file systems, similarly to the way Cinder provides such a canonical control plane for block storage. This allows to define a new type of OpenShift Storage Class that enables the creation of volumes backed up by NFS and therefore shareable between different pods at the same time, hence providing RWX (read write many) access for them.

To add this support, we have created a new Manila CSI Driver Operator which is in charge of the deployment and configuration, and that allows the provisioning of dynamic manila CSI volumes. More information about its usage can be found at https://docs.openshift.com/container-platform/4.5/storage/container_storage_interface/persistent-storage-csi-manila.html

5.2.5. Kubernetization of Kuryr-Kubernetes by adapting CRDs model

There is a current trend of moving containerized applications to the Kubernetes/operators model which allows you to extend Kubernetes API with application specific object/APIs. This model pursued by Kubernetes and CNCF communities, gives some advantages and de-facto standardization that were fitting really well into the Kuryr-Kubernetes model. Therefore, we initiated the effort on modernization the Kuryr internals by adopting this model.

In contrast to regular OpenStack projects like Nova or Cinder, Kuryr-Kubernetes behaviour is driven by events happening in Kubernetes (e.g. Pod or Service being created) and not by user calling OpenStack REST API. A pattern when application acts upon events received from Kubernetes API and adjusts environment state accordingly is called "controller" in Kubernetes world. Moreover, Kuryr does not use any database, and it used to store some information on Kubernetes object annotations. Thanks to the adoption of Kubernetes Custom Resources (called CRDs), we now store that state of the OpenStack resources on specific CRDs on the Kubernetes environment instead. This give us several advantages such as:

- Limit the number of calls to OpenStack, with the consequent performance improvement as well as reduction on its load
- Easier to debug/check the status by looking at the existing Kubernetes objects, in this case the Kuryr specific CRDs.

This CRDs adoption by Kuryr has been made at 4 main points:

CC BY NC ND

- KuryrNetworks: related to the namespace handling by Kuryr, which is in charge of creating subnets in OpenStack for OpenShift/Kubernetes namespaces
- KuryrPorts: related to the pod handling by Kuryr, which is in charge of creating the needed ports in OpenStack for OpenShift/Kubernetes pods
- KuryrLoadBalancers: related to the service handling by kuryr, which is in charge of creating the Load Balancers in OpenStack for OpenShift/Kubernetes services/endpoints
- KuryrNetworkPolicy: related to the network policy handling by Kuryr, which is in charge of creating the SecurityGroups/SecurityGroupRules in OpenStack for OpenShift/Kubernetes network policies.

This have been the biggest effort on Kuryr-Kubernetes during the last OpenStack release (Victoria) and it has already being noticed by the community such as the blog post article https://www.sdxcentral.com/articles/news/att-verizon-5g-deployments-boost-openstack-work/2020/10/ that mentioned the next:

The biggest move was the release of Victoria, which is the 22nd OpenStack release. Victoria builds on the previous Ussuri launch with more than 20,000 code changes that include native integration with Kubernetes and more support for diverse infrastructure deployments.

The Kubernetes integration is on the back of the Kuryr container networking plugin. It acts as the link that delivers the OpenStack networking into containers. Kuryr now has support for customer resource definitions (CRDs) that remove the need for it to use annotations to store data about OpenStack objects in the Kubernetes API.

"*Kuryr has adopted this as the way that it passes information back and forth between the underlying infrastructure that a Kubernetes cluster may be running on and the Kubernetes environment itself," **explained OSF Executive Director Jonathan Bryce**, in a press briefing. "This is great because it brings the two systems closer together using the native components on each side."*

5.3. Integration Highlights

For the first part of the project, initial support for OpenStack was included into the OpenShift-Ansible installer to handle the creation of OpenStack resources. This was based on OpenShift 3.11 as OpenShift 4.X was currently at a *being developed* phase and an OpenShift cluster was needed so as not to block the other components. As soon as the work on moving OpenShift to the operator's model was stable enough (OpenShift 4), we moved our testbed to that release. We did this as part of our testbed migration to the Massachusetts Open Cloud (MOC – https://massopen.cloud/). The objective of this cloud is to create a self-sustaining at-scale public cloud based on OpenStack. It serves as a marketplace for industry partners (Red Hat being one of them) as well as a place for researchers and industry to innovate and expose innovation to real users. We obtained an OpenStack user project with enough quota for our experiments and use cases:

- 1 TB RAM, >1 TB Storage (plus access to the ceph storage cluster), 30 volumes, 20 instances, 300 cores, 300 networks, ...

On top of this OpenStack cloud, and based on the extensions made to the OpenShift Installer to have better support when can be installed on top of OpenStack, we deployed

cc BY NC ND

OpenShift (4.5 version) on the testbed. This support extends the OpenShift installer to create/delete OpenStack VMs and later install the packages, configuration files, keys, services, etc., needed to install and configure the OpenShift cluster on top of them. It includes the basic operators and prepares the system for the new ones to be created as part of the BigDataStack project.

We followed the best practices (configuration) for deploying OpenShift on top of OpenStack already outlined within D3.1; the reader can also refer to that deliverable to see an account of the minimum number of each OpenStack resource types that are needed for a minimal installation of OpenShift on top of OpenStack.

On top of this OpenShift cluster, the rest of the Big Data Stack components are installed in a containerized mode. They are able to take advantage of the extensions made to the cluster management. Some of them transparently:

- Improved network performance thanks to Kuryr integration into the Cluster Network Operators
- Improved performance for services, both control plane (faster to create and more scalable), data plane (improved bandwidth with reduced latency), resource consumption (no need for extra VMs), and fault tolerance (no single point of failure)
- Kuryr CRD adoption

And some other by leveraging the new APIs

- OpenShift Routes for the Gateway and access of applications from the outside of the cluster
- OpenShift Cluster API that allow to easy scale up/down the cluster itself. Even with automatic scaling based on usage, as covered here: https://www.openshift.com/blog/autoscaling-with-openshift-on-openstack
- Fine-grain network access control to the applications by using Network Policies, thanks to its integration into Kuryr
- Support for Manila and thus the option to have RWX PVs in OpenShift, which makes possible for pods to share data through volumes.

5.4. Experimentation Outcomes

5.4.1. Distributed OVN Load Balancer performance

We did some initial experimentation that focused on the initial integration testing and scale testing of OVN-Octavia distributed load balancing for Kubernetes Services.

A performance comparison between Kuryr and OpenShift SDN was carried out, proving a performance boost of up to nine times better for throughput, as presented in the following figures, while additional results have been published online at the OpenShift blog[4].

[4] https://blog.openshift.com/accelerate-your-openshift-network-performance-on-openstack-with-kuryr

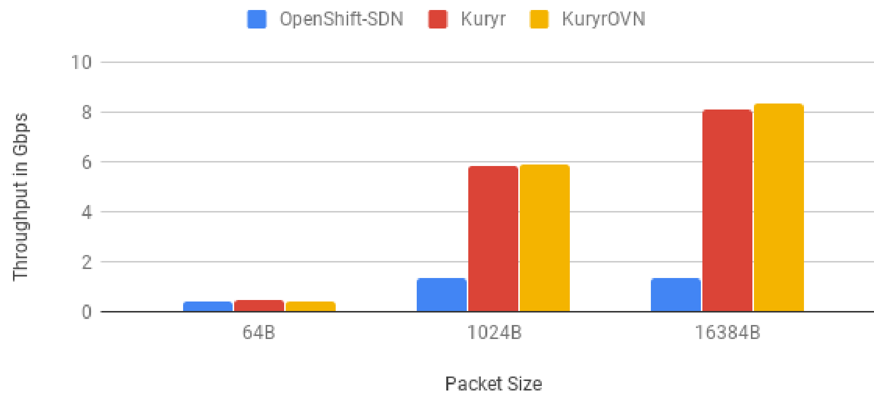


Figure 4: Throughput improvements (POD to POD)

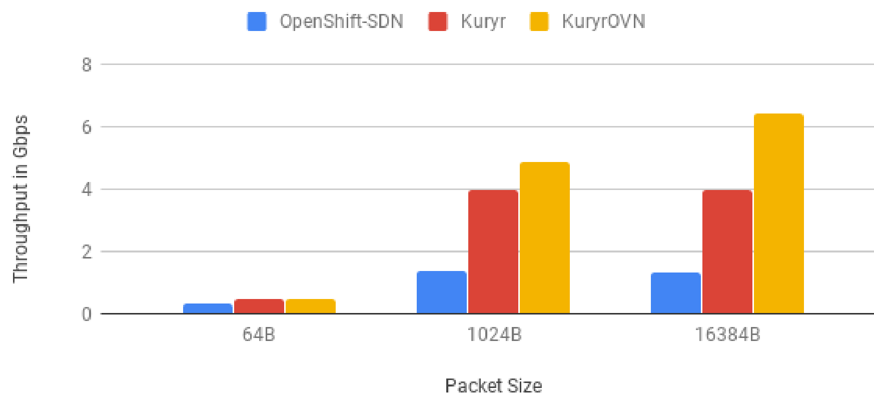


Figure 5: Throughput improvements (POD to SVC)

Thanks to the integration of the distributed OVN load balancer into Kuryr (and Octavia), the customers interest on this has raised and consequently Red Hat is performing some bigger scale testing with it, similarly to the already made scale testing for OpenStack itself in here but with OpenShift with Kuryr on top: https://www.redhat.com/en/blog/scaling-red-hat-openstack-platform-161-more-700-nodes

5.4.2. Kuryr tuning for real use cases

As part of the upstream development, components are not usually target to just one use case and they have some configuration knobs to be able to be adapted to the specifics of each case. This was the case for kuryr-kubernetes too.

Thanks to the Kuryr-Kubernetes extensions as part of BigDataStack, and due to its network performance improvements as well as its simplified model to expose applications to the outside work, some customer has already started testing OpenShift on top of OpenStack

with Kuryr. However, the above mention knobs needed to be adapted to the needs of the specific use cases. This was gathered in the next blog post:

- https://developers.redhat.com/blog/2020/10/02/customizing-and-tuning-the-kuryr-sdn-for-red-hat-openshift-3-11-on-red-hat-openstack-13/

As it can be seen there may be some needs to adapt, among others:

- Services and Load Balancer ranges, to accommodate for the specific ranges available at the customer, as well as to adapt it to the expected OpenShift cluster usage, i.e. number of expected services, pods, etc.
- Type of isolation required by the applications: namespace isolation vs network policy isolation. This depends on the application needs and if fine grain control is not needed (i.e., network policies), a simpler approach can be taken with the namespace isolation.
- Ports pre-creation to save time and expensive OpenStack calls. This allow to have certain amount of Neutron ports ready to be used by the OpenShift/Kubernetes pods. This improves the time needed for pods to be running by one order of magnitude as well as remarkably decreases the load on Neutron server on pods creation spikes. However, it comes at the expenses of more ports being created per OpenShift node and depending on the size of the network it may lead to problems, such as running out of subnet IPs. As a consequence, its configuration is needed considering both the size of the available networks (/24, /26, …) as well as the size of the cluster (number of nodes).

5.4.3. Autoscale experiments through Infrastructure provided APIs

Thanks to leveraging the CRD usage model and the OpenShift on OpenStack integration, it is possible to expose the infrastructure through OpenShift/Kubernetes APIs. In turns, this allows us for an easy way to increase or decrease the size of the OpenShift cluster, with just one single command (or click) and in an exactly the same way as you would do with the number of pods for an applications, i.e., just increasing the number of replicas.

What is more, this opens the floor for more advanced autoscaling techniques, based on either usage or predictions/estimations about future needs, which could trigger the scale up/down of the cluster to either get more resources for your applications or reduce the resource consumption when not needed.

As part of the OpenShift integration on top of OpenStack we have also added support for the autoscaler that automatically can trigger the OpenShift cluster scaling actions based on load, where both the maximum and minimum number of workers can be specified, e.g.:

Create Machine Autoscaler

Create by manually entering YAML or JSON definitions, or by dragging and dropping a file into the editor.

```yaml
apiVersion: "autoscaling.openshift.io/v1beta1"
kind: "MachineAutoscaler"
metadata:
  name: "ocpra-vgx2w-worker"
  namespace: "openshift-machine-api"
spec:
  minReplicas: 1
  maxReplicas: 8
  scaleTargetRef:
    apiVersion: machine.openshift.io/v1beta1
    kind: MachineSet
    name: ocpra-vgx2w-worker
```

For more information about this you can see the blog post and demo video in here:
https://www.openshift.com/blog/autoscaling-with-openshift-on-openstack

CC BY NC ND

6. Realization Engine

The Realization Engine is a new component of the BigDataStack platform that has been developed within WP3 in Y3 by the GLA partner as an addition to Task 3.3 (Dynamic Deployment Patterns & Runtime Re-Configuration). The goal of the Realization Engine is to provide a central suite of containerized services that enable configuration, deployment and subsequent management of user applications and their components. The main functionalities provided by the Realization Engine are:

- Registration and storage of user applications (either via complete BigDataStack Playbooks or in smaller units).
- Deconstruction of BigDataStack Playbooks into constituent components for easier management. These components are: the application definition; comprised object definitions (Deployment Configs, Jobs, Services, Routes, etc.); exported metrics; service level objectives; operation sequences; and application states.
- Provision of built-in object-level management actions for the user's application
- Support for complex deployment or alteration actions in the form of operation sequences.
- Live OpenShift cluster state monitoring, enabling synchronisation of application states between the cluster and Realization Engine supporting automated action triggering.
- Short-term time-series data storage for Realization Engine managed metrics.
- Provision of a REST API for accessing application, component, and cluster status, as well as triggering actions.
- Provision of a graphical user interface enabling run-time application monitoring and management
- Integration with the other WP3 components (e.g. ADS Ranking and ADS Deploy).

In the remainder of this section we will discuss why the Realization Engine was introduced as well as provide a technical overview for it. In particular, in Section 6.1 we summarise the motivation for the introduction of the Realization Engine. Section 6.2 describes the additional requirements identified for the Realization Engine. In Section 6.3 we describe how internal modelling of user applications has changed to enable better division of complex applications into components. Meanwhile, Section 6.4 details updates made to the BigDataStack Playbook format to reflect these modelling changes. Section 6.5 provides an architectural overview of the Realization Engine, while Section 6.6 provides a brief overview of each of the services comprised within it. Finally, Section 6.7 summarizes the different built-in operations within the Realization Engine and what they are used for.

6.1. Motivation

For the BigDataStack M18 review, a demonstration system was developed to illustrate the different components and added value of the BigDataStack project. Post the M18 review, the consortium internally performed a separate review of this system to identify potential issues with the platform design. As outcome of this review, there were two main architectural limitations identified that impacted WP3:

- The first limitation was that from a user-facing perspective, the application engineer

cc BY NC ND

lacked a means to manage their application post deployment. In effect, once the application was launched, management was fully automatic and if the user disagreed with alterations that the platform implemented they could not intervene. Hence, there was a need for additional (manual) management capabilities that had not been originally envisaged.
- The second limitation that was identified was regarding the underlying application modelling. In the original design, a user application was considered to be atomic, and as such could be orchestrated as a single unit. The implications of this are many-fold, however we summarize some of the more important down-stream impacts here. First, all deployment and alteration actions had to be natively supported by the user application (e.g. via a Kubernetes Operator) and then have a supported trigger within one of the BigDataStack components (e.g. ADS-Deploy or Adaptive Networking). This made the BigDataStack platform very rigid and difficult to adapt to new application types. Second, as any complex deployment and alteration actions had to be defined within the application, progress regarding those complex actions could not be tracked or visualised by BigDataStack. This could lead to incorrect statuses being shown to the user, while also causing issues for any active automated orchestration systems that rely on those states for decision making. As a result, it was decided that applications needed to be divisible into components that could be independently managed and tracked, as well as that all deployments and alterations needed to have clearly defined stages that were managed by BigDataStack.

To address these limitations, it was clear that there needed to be a centralized system that could track and manage the user application at component-level, which was not in the original platform architecture. Hence, the Realization Engine was designed and developed.

6.2. Requirements

In this section we provide an overview of the requirements that were identified for the Realization Engine. Realization Engine requirements are REQ-RE-XX.

	Id	**Level of detail**	**Type**	**Actor**	**Priority**
	REQ-RE-01	System	FUNC	Data Toolkit	MAN
Name	BigDataStack Playbook Registration				
Description	Once the application engineer and/or data scientist has defined the application via the Data Toolkit (or has generated a BigDataStack Playbook manually), the Realization Engine needs to ingest the application definition and divide it into individual components and store them.				
Additional Information	Sending of the application definition is performed by REST API in YAML format as a BigDataStack Playbook.				

cc BY NC ND

	Id	**Level of detail**	**Type**	**Actor**	**Priority**
	REQ-RE-02	System	FUNC	Application Engineer	MAN
Name	Operation Triggering				
Description	Once an application has been registered, the user should be able to trigger pre-defined actions that deploy or alter their application. Operations may be atomic or be comprised of multiple tasks.				
Additional Information					

	Id	**Level of detail**	**Type**	**Actor**	**Priority**
	REQ-RE-03	System	FUNC	Realization Engine	MAN
Name	Application Component State Tracking				
Description	Once an application component is deployed on the cluster, the Realization Engine needs to monitor the state of that component and generate alerts when state-changes occur.				
Additional Information	This enables state reporting within the Realization Engine itself as well as can be used to inform other orchestration software of component states.				

	Id	**Level of detail**	**Type**	**Actor**	**Priority**
	REQ-RE-04	System	FUNC	Application Services	MAN
Name	API Suite				
Description	The Realization Engine should enable other components or services to access the information about applications that it manages, as well as enable actions to be triggered for them.				
Additional Information	This enables other components like the Dynamic Orchestrator to both get information about applications for decision making, as well as expose what actions can be performed at any one time.				

	Id	**Level of detail**	**Type**	**Actor**	**Priority**
	REQ-RE-05	System	FUNC	Application Engineer	MAN
Name	GUI				
Description	The Realization Engine should provide a graphical user interface that enables access the information about applications that it manages, as well as enable actions to be triggered.				
Additional Information					

	Id	Level of detail	Type	Actor	Priority
	REQ-RE-06	System	FUNC	ADS-Ranking	MAN
Name	Integration with ADS-Ranking				
Description	The Realization Engine should provide native support for the ADS-Ranking (Deployment Recommender Service) also developed within T3.3.				
Additional Information					

	Id	Level of detail	Type	Actor	Priority
	REQ-RE-07	System	FUNC	ADS-Deploy	MAN
Name	Integration with ADS-Deploy				
Description	The Realization Engine should provide native support for the ADS-Deploy to facilitate deployment of application components				
Additional Information					

	Id	Level of detail	Type	Actor	Priority
	REQ-RE-08	System	FUNC	Dynamic Orchestrator	MAN
Name	Integration with the Dynamic Orchestrator				
Description	The Realization Engine should provide native support for Dynamic Orchestrator configuration upon deployment of an application component.				
Additional Information	This involves sending information about the application component and service level objectives to the dynamic orchestrator.				

	Id	Level of detail	Type	Actor	Priority
	REQ-RE-09	System	FUNC	-	MAN
Name	Local Timeseries Metric Storage				
Description	The Realization Engine should also provide optional support for local time-series data for cases where it is deployed without the Triple Monitoring Engine.				
Additional Information	This de-couples the Realization Engine from the Triple Monitoring Engine, enabling it to optionally be used in a stand-alone mode.				

	Id	Level of detail	Type	Actor	Priority
	REQ-RE-10	System	FUNC	Application Engineer	MAN
Name	BigDataStack Pilot Support				

Description	The Realization Engine should support sufficient operations out-of-the-box to enable deployment and management of the BigDataStack Pilots
Additional Information	

6.3. Modular Object Design

To meet the above requirements (particularly REQ-RE-01, REQ-RE-02 and REQ-RE-03) as well as to tackle the second limitation discussed in Section 6.1, how the user application was modelled, needed to be revised. The core of this revision was the transition from a user application being considered a single atomic unit to an application being instead seen as a grouping of interconnected components that can have actions performed upon them. This enables more granular tracking and alterations to be made to a user application. At the same time, we also added explicit modelling of supporting data structures to hold information linked to an application component, such as exported metrics, resource templates and service level objectives.

The following figure illustrates the new conceptual application model used by the Realization Engine. In particular, under this model, the user account or 'owner' owns one or more applications and can also define metrics. A single application has a state, zero or more object (templates) representing the different components of the application, zero or more operation sequences representing actions that can be performed for the application, and a series of events generated about the application. An object template (application component) can be instantiated multiple times, producing object instances. Object instances may have an associated resource template describing the resources assigned to that object. An object instance contains a definition of an underlying Kubernetes or OpenShift object that contains the deployment information. Operation sequences represent actions to perform on the application and contain multiple atomic operations. An operation targets either an object template or instance, performing either some alteration or deployment action upon it. Service level objectives can be attached to an object instance, which track a metric exported by or about that object.

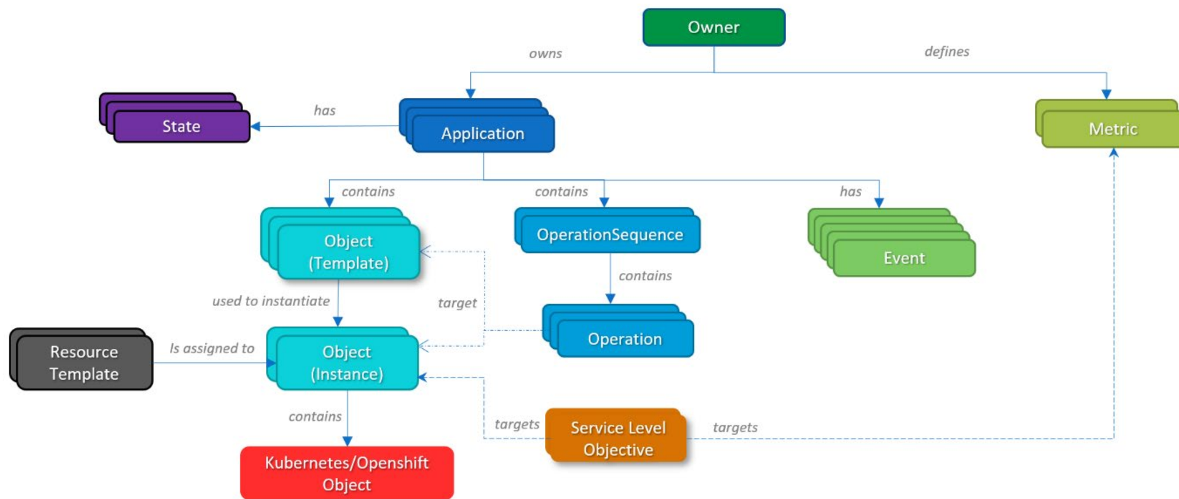


Figure 6: Realization Engine Application Model

In the remainder of this section we summarize each of the main objects/classes that the Realization Engine uses internally to model a user application.

6.3.1. (BigDataStack) Application

The Application class represents at a high-level the concept of a user application. A user application has an owner (a Kubernetes/OpenShift user), a namespace/project (the Kubernetes namespace or OpenShift project that the application will be deployed to) and an identifier (appID). Together, these three pieces of information uniquely identify the application. An application also has zero or more types associated to it. These types specify information about how components within that application should be processed, typically during first registration. For example, the '*inferMissingValues*' type tells the Realization Engine to check that all application metadata is specified within a component during registration, and if not to add it. Finally, an application also has a name and a description associated to it. These are used for visualisation within the Realization Engine graphical user interface.

BigDataStack Application		
Field	**Type**	**Size**
owner	VARCHAR	140
namespace	VARCHAR	140
appID	VARCHAR	100
types	VARCHAR (List<String>)	1000
name	VARCHAR	140
description	TEXT	65535

6.3.2. (BigDataStack) Object

A BigDataStack Object is the structure that the Realization Engine uses to represent a single component within a user application. To aid in compatibility with both Kubernetes and OpenShift, a BigDataStack Object has a 1-to-1 relationship with an underlying Kubernetes or OpenShift object, such as a Deployment Config, Job, Service, Volume Claim, etc. The

CC BY NC ND

differences between a BigDataStack Object and one of these underlying objects are two-fold. First, a BigDataStack Object can be either a Template or an Instance. When a BigDataStack Object is first registered it is stored as a Template. Instances can then be spawned from stored templates (via the Instantiate Operation discussed later), enabling a single template to be reused to create multiple instances. It is also worth noting that a template can be stored in an incomplete form, and then be modified during instantiation to fill in missing information, enabling customisation at deploy-time. The second main difference is that as multiple instances of a template can exist at one time, each BigDataStack Object has an instance number that is needed to uniquely identify an instance.

BigDataStack Object		
Field	**Type**	**Size**
owner	VARCHAR	140
namespace	VARCHAR	140
appID	VARCHAR	100
objectID	VARCHAR	100
instance	INT	-
type	VARCHAR	100
status	VARCHAR (Set<String>)	1000
yamlSource	TEXT	65535

In terms of fields comprising a BigDataStack Object, each object contains the secondary keys owner, namespace and appID, connecting that object to a user application. To uniquely identify an individual object within an application, an objectID (that uniquely identifies the template) and instance number are used (where the template is instance 0). Additionally, a BigDataStack Object has a type, which corresponds to the underlying type of the Kubernetes or OpenShift object, a status which represents the aggregate state of the underlying instances and finally a yamlSource field, that contains the raw source for the underlying type of the Kubernetes or OpenShift object.

6.3.3. (BigDataStack) Operation

A BigDataStack Operation is a representation of an 'action' that can be performed on a BigDataStack Object. For example, spawning an instance from an object template is an (Instantiate) operation. Similarly, deploying an object instance onto a cluster is an (Apply) operation. The Realization Engine defines a set of standard operations that can be performed on any BigDataStack Object. It is through these operations that users (or programmatic orchestrators) can deploy or alter their applications.

BigDataStack Operation		
Field	**Type**	**Size**
className	VARCHAR	140
objectID	VARCHAR	100
state	VARCHAR	100

A BigDataStack Operation is mapped to a (java) class within the Realization Engine that contains the logic for executing that operation. An operation always targets an object, and hence requires an objectID as a target, although additional parameters may be provided depending on the implementation. For example, the ExecuteCMD operation (which executes one or more commands on one or more containers) requires an 'instancelookup' string, which defines the matching criteria to determine which instances of the specified

CC BY NC ND

object should be targeted. In cases where multiple operations are grouped together into an operation sequence (see the next section), then the state of each operation will be recorded an updated. An operation can be in the following states: 'NotStarted', 'InProgress', 'Completed' or 'Failed'. The currently supported operations within the Realization Engine are summarized later in Section 6.7.

6.3.4. (BigDataStack) Operation Sequences

A BigDataStack Operation Sequence, as its name suggests, is a sequence of BigDataStack Operations. The goal of an Operation Sequence is to provide a way to group atomic Operations together, enabling the formation of more complex higher-level actions that the user may wish to perform. For example, a common deployment pattern is comprised of 'Instantiate' (create a new instance of an object), 'SetSequenceParameters' (that sets parameters within the new object instance), and 'Apply' (which creates the object instance on the cloud/cluster). However, for more complex applications, an operation sequence may contain 10's to 100's of individual operations. For instance, for the ATOSWL Grocery Recommendation Pilot, an operation sequence exists to deploy that application from first principles, which is comprised of 46 operations.

Like a BigDataStack Object, an Operation Sequence can be either a template or an instance. When the user registers a new operation sequence it is stored as a template. When the user triggers that operation sequence, an instance of that sequence is spawned, and then executed in a separate container. An operation sequence will process the operations contained within in sequential order. The 'mode' field of an operation sequence defines the behaviour of the processing in the case of sequence restarts or failures. An operation sequence can be run in the following modes:

BigDataStack Operation Sequence		
Field	**Type**	**Size**
owner	VARCHAR	140
namespace	VARCHAR	140
appID	VARCHAR	100
sequenceID	VARCHAR	100
instance	INT	-
name	VARCHAR	140
description	TEXT	1000
mode	VARCHAR	100
parameters	VARCHAR (Map<String,String>)	5000
operations	TEXT (List<Operation>)	65535

- **Run**: Executes all operations regardless of state, will exit on an operation failure.
- **Continue**: Will execute all operations that are not in 'Completed' state already. This enables an operation sequence to be restarted in the event of a transient failure.
- **RunIgnoreFailures**: Will execute all operations regardless of state and will still continue to the next operation even if a failure is detected.

Regarding the internal representation of a BigDataStack Operation Sequence, each sequence contains the secondary keys: owner, namespace and appID, linking the sequence to an application. A sequence is uniquely identified within an application by its sequenceID and instance number (where the template is instance 0). A sequence has both a name and

description, which are used for display within the Realization Engine graphical user interface. Finally, the mode field defines the mode of operations, the parameters field is a <key,value> mapping that contains parameters that can be used to customise objects created or used by the sequence, and the operations field specifies the operations to perform.

6.3.5. (BigDataStack) Events

A BigDataStack Event represents a notification of some underlying change relating to an application. An event may be generated for a variety of reasons, such as the user registering a new application, a state change in a component detected by OpenShift, operation completion, or a quality of service violation, among others. The goal of a BigDataStack Event is to provide a standardised format for reporting application changes that can both be displayed to the user in an informative manner, while also being a functional trigger that can be used for automated orchestration.

Events are keyed to a particular application (i.e. have the secondary keys owner, namespace and appID). Most events will reference a particular object instance that the event is about (defined by objectID and instance). The event itself then has a number (eventNo), to uniquely identify that event for the application. For the purposes of display, an event has a title and description. Finally, a type field specifies the type of event (which is usually used to specify what caused the event to be created) and a severity level.

BigDataStack events are most commonly generated by the Realization Engine itself, as it makes changes to the user's application and observes run-time state changes in that application. However, other components can create new events via an API endpoint. Events are persistently stored within the Realization Engine. Additionally, if enabled, events can be pushed to a RabbitMQ mailbox that other services can subscribe to in cases where push notification of application changes is desirable.

BigDataStack Event		
Field	**Type**	**Size**
owner	VARCHAR	140
namespace	VARCHAR	140
appID	VARCHAR	100
objectID	VARCHAR	100
Instance	INT	-
eventNo	INT	-
eventTime	BIGINT	-
title	VARCHAR	140
description	TEXT	1000
type	VARCHAR	100
severity	VARCHAR	100

6.3.6. (BigDataStack) Metric

User applications and other services within the BigDataStack ecosystem export metrics that provide valuable run-time information about user applications and the cluster itself. A BigDataStack metric is a high-level description of the properties of such a metric. Note that a metric definition here only represents the

BigDataStack Metric		
Field	**Type**	**Size**
owner	VARCHAR	140
name	VARCHAR	140
metricClassname	VARCHAR	20
summary	VARCHAR	3000
maximumValue	VARCHAR	40
minimumValue	VARCHAR	40
higherIsBetter	BOOLEAN	-
displayUnit	VARCHAR	100

CC BY NC ND

concept of the metric, it does not directly refer to a concrete time-series being generated by an object. A BigDataStack Metric has two main uses within the Realization Engine, namely: 1) BigDataStack Metrics and BigDataStack Objects are linked together within Service Level Objectives (see Section 6.3.7); and 2) the details of the metric are used for display within the Realization Engine graphical user interface.

Metrics are high-level constructs and as such are not directly linked to an application, but rather only to a particular user that registered them. A metric is uniquely identified by its name. It also contains a display summary that explains what the metric measures, in addition to a display unit. A metric definition also has functional information about that metric that is useful when performing comparisons with that metric. In particular, the metric definition provides valid bounds for the metric value (maximumValue and minimumValue), along with a reference to the (java) class that can be used to parse such values (metricClassname). A higherIsBetter field is also included, indicating whether higher or lower values are typically seen as desirable for this metric.

6.3.7. (BigDataStack) Service Level Objective

To enable down-stream monitoring of a user's application for quality of service failures, as well as to enable subsequent automated orchestration to rectify such failures, the user needs to have a way to define what quality of service means for their application. This is achieved via BigDataStack Service Level Objective (SLO) definitions. In effect, a Service Level Objective connects a BigDataStack Object (the application component to track) and a BigDataStack Metric (what to track about the component). The SLO definition then sets a target value or threshold to compare against (value) along with a type of comparison to perform (e.g. 'lessThan'). To enable users to set hard targets that must be met, as well as softer targets that should be met if possible, each SLO specifies whether it is a requirement (hard target) or not, along with how severe a failure it is if the SLO is not met.

BigDataStack Service Level Objective

Field	Type	Size
owner	VARCHAR	140
namespace	VARCHAR	140
appID	VARCHAR	100
objectID	VARCHAR	100
Instance	INT	-
metricName	VARCHAR	140
sloIndex	INT	-
type	VARCHAR	100
value	DOUBLE	-
breachServerity	VARCHAR	100
isRequirement	BOOLEAN	-

SLOs are primarily used by the Quality of Service (QoS) Evaluation component of BigDataStack, which is responsible for monitoring the status of each SLO. Some SLOs are also used by ADS-Ranking to help in validating different resource templates for a user application, i.e. to estimate whether it is likely that a deployment with a fixed set of resources will meet the SLOs set by the user (see Section 8.5).

6.3.8. (BigDataStack) Resource Template

A Resource Template represents a set of resources to be allocated to a BigDataStack Object from the cluster. When deploying a BigDataStack Object, it is good practice to include a Resource Template, such that the cluster knows what the object needs to function, reducing the risk that the component will fail due to a lack of resources later on. Resource capacity can be specified either in terms of a request (the minimum amount of the resource needed for the object to function) and/or a limit (the maximum amount of the resource that the object would like). When considering resources, there are three resource types of interest: CPU, Memory and GPUs. CPU capacity represents the number of compute cores that the object has available to it. Memory capacity is the amount of RAM available to the object. GPU capacity is simply the number of GPU cards assigned to the object[5]. As a work-around for the current limitations of in-built GPU scheduling within Kubernetes and OpenShift, the resource template also contains a node selector field. This is useful in cases where the underlying infrastructure uses homogeneous GPU configurations (i.e. one physical compute node only contains one type of GPU) with appropriate labels, effectively enabling the GPU type to be set by forcing the scheduler to place the object on a particular node type.

BigDataStack Resource Template		
Field	**Type**	**Size**
owner	VARCHAR	140
namespace	VARCHAR	140
appID	VARCHAR	100
objectID	VARCHAR	100
Instance	INT	-
request.cpu	VARCHAR	100
request.memory	VARCHAR	100
limit.cpu	VARCHAR	100
limit.memory	VARCHAR	100
numGPU	INT	-
nodeSelector	VARCHAR	500

6.3.9. (BigDataStack) Application State

As an optional feature, it is possible for the user to define custom states for their application along with criteria that must be met for the application to be considered as in those states. In a simple case, the user might define a state 'Services OK', where the criteria to be met is that all of the application components are in a 'Running' state. This can be useful to an application engineer or other user maintaining the application, as they can easily get a view on whether the application is functioning correctly. A more complex application of state definitions would be to encode all possible states that an application could be in, and then

BigDataStack Application State		
Field	**Type**	**Size**
owner	VARCHAR	140
namespace	VARCHAR	140
appID	VARCHAR	100
appStateID	VARCHAR	100
name	VARCHAR	1000
notInStates	VARCHAR (List<String>)	1000
sequences	VARCHAR (List<String>)	1000
conditions	VARCHAR (List<Condition>)	5000

[5] GPU support in OpenShift and Kubernetes clusters currently fully featured. In the future it would be desirable to be able to specify GPU types as well as share GPUs among multiple objects, but this is not currently possible.

use those states as triggers for automated orchestration.

An application state has an identifier that uniquely identifies it (appStateID), along with a name (for display to the user). For the purposes of identifying whether the application is in a particular state, a list of application conditions are specified. An application condition can check either the state of an object or the state of an operation sequence, dependant on the information specified. If one or more objectIDs are specified, the condition will return true if there are at least the target number of instances of each object with the specified state. Meanwhile, if the sequenceID is specified, it will return true if the specified sequence is in the target state and not in any of the other states as listed within the 'notInStates' field of the condition. As well as the conditions, the application state can also define a separate 'notInStates' field, which enables checking whether the application is currently in any other of a list of application states. This is useful if the user wants to stop an application from being in multiple states simultaneously.

BigDataStack Application Condition		
Field	**Type**	**Size**
objectIDs	VARCHAR (List<String>)	140
numInstances	INT	-
state	VARCHAR	100
sequenceID	VARCHAR	100
notInStates	VARCHAR (List<String>)	1000

Finally, the application state can also contain a list of operation sequence identifiers. This is such that the user can set particular actions to become available depending on the state of the application. If the sequence identifier list exists, the Realization Engine will filter the available list of operation sequences shown to the user based on the current application state.

6.4. Updated Playbook Formatting

As the way that the underlying application is modelled has changed, this in turn mandated associated changes to the format of the BigDataStack playbook that it ingests (see REQ-RE-01). A BigDataStack playbook represents a single application, and should be able to provide all of the needed information about that application (although some information can be omitted and then added later). As such, the updated format playbook is structured into the different modular objects specified above. In particular, the BigDataStack Application data is specified at the top, followed by lists of related objects (BigDataStack Objects, BigDataStack Metrics, BigDataStack Service Level Objectives, BigDataStack Operation Sequences and BigDataStack Application States), as illustrated in Figure 7.

bigdatastack.eu

```
# -----------------------------------------------------------------------------------------------
# Top-Level Application Definition
# -----------------------------------------------------------------------------------------------
appID: "appID"
name: "Display name for the application"
description: "A description of the user application"
owner: "$owner$" # Openshift username to deploy this app as
namespace: "$namespace$" # Openshift Project / Kubernetes namespace to deploy this app within
types:
 - playbook
 - inferMissingValues

# -----------------------------------------------------------------------------------------------
# Application Modelling Information
# -----------------------------------------------------------------------------------------------

# Object List
objects:
  - ...
  - ...

# Metric List
metrics:
  - ...
  - ...

# Service Level Objective List
slos:
  - ...
  - ...

# Operation Sequence List
sequences:
  - ...
  - ...

# Application State List
states:
  - ...
  - ...
```

Figure 7: Updated BigDataStack Playbook Format

6.5. Realization Engine Architecture

Having discussed the internal modelling that the Realization Engine uses to represent the user application and associated information, we next describe the overall architecture of the Realization Engine itself. The Realization Engine is implemented as a series of containerized services, with distinct roles and functionalities designed to meet the requirements discussed in Section 6.2. Figure 8 provides an overview of the different containerized services within the Realization Engine along with the communication flows between those services. The green boxes denote the core containerized services provided by the Realization Engine. Orange boxes indicate off-the-shelf data-stores or data exchange services. Meanwhile blue boxes indicate other BigDataStack services used by the Realization Engine. The role of each of the main services (green boxes) are as follows:

- **Realization Engine (and Application API)**: This is a containerized service that houses the main application management logic. It also exposes the Realization Engine API that provides other components with access to user application states and actions (REQ-RE-04), as well as enabling the registration of new applications by the Data Toolkit (REQ-RE-01).
- **Realization UI**: This is a graphical user interface exposed by the Realization Engine that enables users to view the state of their applications, as well as trigger actions for them (REQ-RE-05).

bigdatastack.eu

- **Cluster Monitoring**: This component is responsible for synchronizing the state of the underlying Kubernetes/OpenShift objects running on the cluster with their associated BigDataStack Object definitions stored in the State DB.
- **Resource Monitoring**: This component acts as a bridge between OpenShift Monitoring (a built-in set of services to OpenShift that track node and pod-level resource usage) and the Realization Engine. This enables the Realization Engine to access live CPU and Memory usage by the application components.
- **Cost Estimation**: The cost estimation component, as its name suggests, generates estimated costs (in US dollars) for the different application components. By doing so, it enables service level objectives such as cost per hour or total cost to be evaluated.
- **Log Search**: This component hosts a search engine that indexes the logs of each running container within the user application and provides custom search functionality for those logs.

The other main component of the Realization Engine is the Operation Sequence (shown as a red box in Figure 8). Previously in Section 6.3.4 we introduced the idea of an operation sequence as a representation of a high-level 'action' that the user could trigger for their application, which was comprised of a series of atomic operations. When one of these operation sequences is triggered (e.g. via API call to the Realization Engine), a new temporary containerized service will be launched that performs the operation sequence. Internally, the operations called within that sequence may then interact with other BigDataStack services to obtain needed functionalities. For example, ADS-Ranking may be called to produce a Resource Template for a BigDataStack Object (REQ-RE-06), ADS-Deploy may be called to deploy a BigDataStack Object (REQ-RE-07), and the Dynamic Orchestrator may be called to register a new BigDataStack Object to be managed (REQ-RE-08). In the following section we describe each of the BigDataStack services in more detail, while in Section 6.7 we describe the operations that can be included within an operation sequence (high-level action).

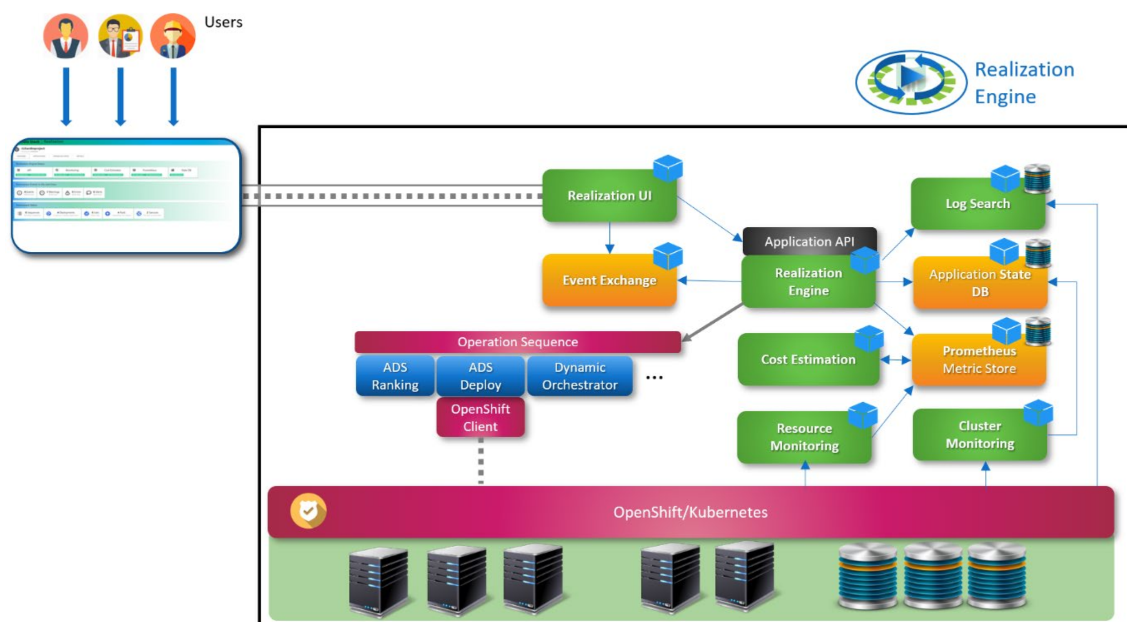


Figure 8: Realization Engine Architecture

bigdatastack.eu

6.6. Containerized Services

In this section we describe the main containerized services within the Realization Engine in more detail. In particular, we describe: the Realization Engine and Application API; the Realization UI; Cluster Monitoring; and the Operation Sequence services. Information about the Resource Monitoring and Cost Estimation components can be found in D5.3 (as they were developed as part of the Application Dimensioning Workbench and then later moved to the Realization Engine).

6.6.1. Realization Engine and Application API

The Realization Engine (and associated API) containerised service is the core of the suite of services that comprised the Realization Engine as a whole. This service is responsible for registering new user applications, updating or adding new objects to those applications, performing actions on those applications (or at least launching a separate service to do so), providing access on-demand to application information and state, as well as providing in some cases short-cuts for accessing information from dependant services (e.g. application cost). Internally, the Realization Engine is a Java-based program compiled as a Jar. The core of this program the Manager class that contains the logic for all of the responsibilities listed above. To achieve this, the Manager maintains a wide array of configured clients, enabling it to both send and receive information about any managed application, access information about the cluster itself, create/delete objects on the cluster, send/receive events, as well as access time-series data stored in the local Prometheus metric data store. The structure of these clients is shown in Figure 9. Of note is that it is this wide availability of information in a central location that makes the Realization Engine a powerful tool, as it effectively makes the Realization Engine a 'one-stop-shop' for all of the distributed and disparate information about the user's application.

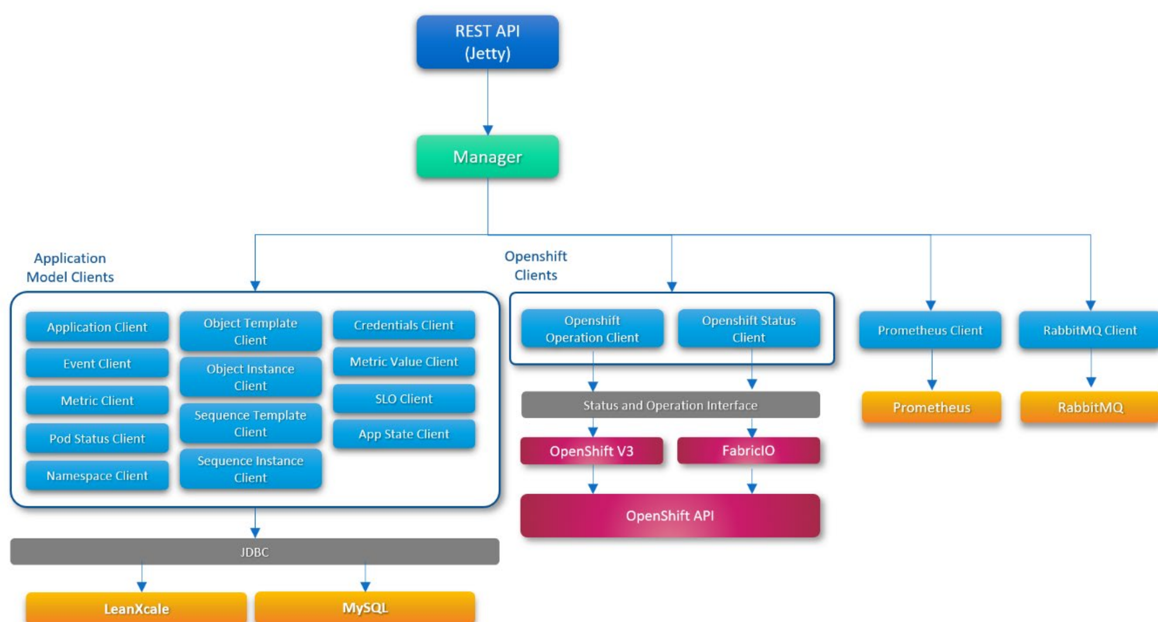


Figure 9: Realization Engine Manager Architecture

cc BY NC ND

On initialization, the Realization Engine container instantiates a new instance of Manager, loading relevant credentials from a file, and performs a check that the minimum required services for the Realization Engine to function are running and accessible, i.e. a copy of the Application State Database, the OpenShift API, and the local Prometheus timeseries database. Assuming this succeeds, the container will then start a Jetty webserver and hosts a set of REST API endpoints that exposes the functionality of Manager to other services. In particular, the API endpoints exposed are listed below. Within an endpoint, values in {} indicate a parameter, e.g. {owner} should be replaced with the owner to target. HTTP POST endpoints require an appropriate object to be included in the message body (if not otherwise specified in JSON format):

Category	HTTP Request Type	Endpoint	Return Type
Register Application Model (YAML Format)	POST	/registeryaml/playbook/{owner}/{namespace}	boolean
	POST	/registeryaml/playbook	boolean
	POST	/registeryaml/application	boolean
	POST	/registeryaml/object	boolean
	POST	/registeryaml/slo	boolean
	POST	/registeryaml/metric	boolean
	POST	/registeryaml/namespace	boolean
	POST	/registeryaml/operationSequence	boolean
Register Application Model (JSON Format)	POST	/registerjson/playbook/{owner}/{namespace}	boolean
	POST	/registerjson/playbook	boolean
	POST	/registerjson/application	boolean
	POST	/registerjson/object	boolean
	POST	/registerjson/slo	boolean
	POST	/registerjson/metric	boolean
	POST	/registerjson/namespace	boolean
	POST	/registerjson/operationSequence	boolean
Retrieve User Applications	GET	/list/{owner}	List<BigDataStack Application>
	GET	/list/{owner}/apps	List<BigDataStack Application>
Retrieve Object Templates	GET	/list/{owner}/objectTemplates	List<BigDataStack Object>
	GET	/list/{owner}/{appID}/objectTemplates	List<BigDataStack Object>
	GET	/get/{owner}/{appID}/objects/{objectID}/template	BigDataStack Object
Retrieve Object Instances	GET	/list/{owner}/objects	List<BigDataStack Object>
	GET	/list/{owner}/{appID}/objects	List<BigDataStack Object>
	GET	/list/{owner}/{appID}/objects/{objectID}	List<BigDataStack Object>
	GET	/get/{owner}/{appID}/objects/{objectID}/instance/{instance}	BigDataStack Object
Retrieve Sequence Templates	GET	/list/{owner}/{appID}/sequenceTemplates	List<BigDataStack Operation Sequence>
	GET	/get/{owner}/{appID}/sequence/{sequenceID}/template	BigDataStack Operation Sequence
Retrieve Sequence Instances	GET	/list/{owner}/{appID}/sequences	List<BigDataStack Operation Sequence>
	GET	/list/{owner}/{appID}/sequence/{sequenceID}	List<BigDataStack Operation Sequence>
	GET	/get/{owner}/{appID}/sequence/{sequenceID}/instance/{instance}	BigDataStack Operation Sequence
Retrieve Pod Statuses	GET	/list/{owner}/{appID}/objects/{objectID}/pods	List<BigDataStack Pod Status>
Retrieve Service Level Objectives	GET	/list/{owner}/{appID}/objects/{objectID}/slos/{metricName}	List<BigDataStack Service Level Objective>
	GET	/list/{owner}/{appID}/objects/{objectID}/instance/{instance}/slos/{metricName}	List<BigDataStack Service Level Objective>
Retrieve Metrics	GET	/list/{owner}/metrics	List<BigDataStack Metric>
	GET	/get/{owner}/metrics/{metricName}	BigDataStack Metric
Retrieve Metric Values	GET	/list/{owner}/{appID}/metrics/{metricName}	List<BigDataStack Metric Value>
	GET	/list/{owner}/{appID}/metrics/{metricName}/{objectID}	List<BigDataStack Metric Value>
Retrieve Events for Application	GET	/list/{owner}/{appID}/events	List<BigDataStack Event>
	GET	/list/{owner}/{appID}/events/{objectID}	List<BigDataStack Event>
Retrieve Events for a User	GET	/list/{owner}/events	List<BigDataStack Event>
	GET	/list/{owner}/events/all	List<BigDataStack Event>
	GET	/list/{owner}/events/error	List<BigDataStack Event>
	GET	/list/{owner}/events/alert	List<BigDataStack Event>

cc BY NC ND

	GET	/list/{owner}/events/info	List<BigDataStack Event>
	GET	/list/{owner}/events/warning	List<BigDataStack Event>
	GET	/list/{owner}/events/all/{type}	List<BigDataStack Event>
	GET	/list/{owner}/events/error/{type}	List<BigDataStack Event>
	GET	/list/{owner}/events/alert/{type}	List<BigDataStack Event>
	GET	/list/{owner}/events/info/{type}	List<BigDataStack Event>
	GET	/list/{owner}/events/warning/{type}	List<BigDataStack Event>
Retrieve the Most Recent K Events for a User	GET	/list/{owner}/kevents?k={depth}	List<BigDataStack Event>
	GET	/list/{owner}/kevents/all?k={depth}	List<BigDataStack Event>
	GET	/list/{owner}/kevents/error?k={depth}	List<BigDataStack Event>
	GET	/list/{owner}/kevents/alert?k={depth}	List<BigDataStack Event>
	GET	/list/{owner}/kevents/info?k={depth}	List<BigDataStack Event>
	GET	/list/{owner}/kevents/warning?k={depth}	List<BigDataStack Event>
	GET	/list/{owner}/kevents/all/{type}?k={depth}	List<BigDataStack Event>
	GET	/list/{owner}/kevents/error/{type}?k={depth}	List<BigDataStack Event>
	GET	/list/{owner}/kevents/alert/{type}?k={depth}	List<BigDataStack Event>
	GET	/list/{owner}/kevents/info/{type}?k={depth}	List<BigDataStack Event>
	GET	/list/{owner}/kevents/warning/{type}?k={depth}	List<BigDataStack Event>
Instant Prometheus Queries	GET	/query/{owner}/{appID}/{namespace}/metrics/{metricName}/{objectID}	BigDataStack Metric Value
	GET	/query/{owner}/{appID}/{namespace}/metrics/{metricName}/{objectID}/{instanceID}	BigDataStack Metric Value
Register New Event	POST	/event/{owner}/{appID}/{objectID}	boolean
Execute Operation Sequence	GET	/exe/{owner}/{appID}/{sequenceID}/start	boolean
	POST	/exe/{owner}/{appID}/{sequenceID}/start	boolean

6.6.2. Realization UI

The Realization UI is an optional component that hosts a Web-based front-end, enabling configuration, deployment and monitoring of user applications by the application engineer or other users. In effect, it provides a user-friendly way for the application engineer to access the functionality of the Realization Engine. The Realization UI integrates with the larger BigDataStack offering through integration with the BigDataStack Visualisation Service (the primary UI for BigDataStack).

The Realization UI is implemented using the Play Framework, which combines a Java/Scala back-end service with an HTML/JavaScript front-end. Communication between the backend and the user browser is handled via asynchronous websockets, enabling the back-end server to push updates to the user as they happen. The component is compiled and run within a container using the SBT build tool, and then exposed via a service and route in OpenShift.

When the user first opens the Realization Engine UI, they will be asked for their OpenShift credentials as well as the project/namespace that they want to manage. This is such that any data requests or actions triggered are performed as that particular user, rather than as whomever launched the realization engine itself. Once the user has submitted their credentials, they are forwarded to the project/namespace overview screen, as shown in Figure 10. The namespace overview screen provides a high-level view of the state of the project/namespace itself, and is a useful way to determine if there is anything that urgently needs the application engineer's attention. In particular, as can be seen from Figure 10, the overview screen provides visual indicators for the state of the Realization Engine itself in the first row (which can be important, as if a component like cluster monitoring was not active, then component states may not be up-to-date). Below that event statistics are provided, such that the user can see if there have been any warnings, errors or other alerts recently. If there were such alerts, the user can use the notifications side bar on the right-hand side to view those events. The final row at the bottom of the overview page gives statistics of the different types of BigDataStack Objects currently running on the cluster.

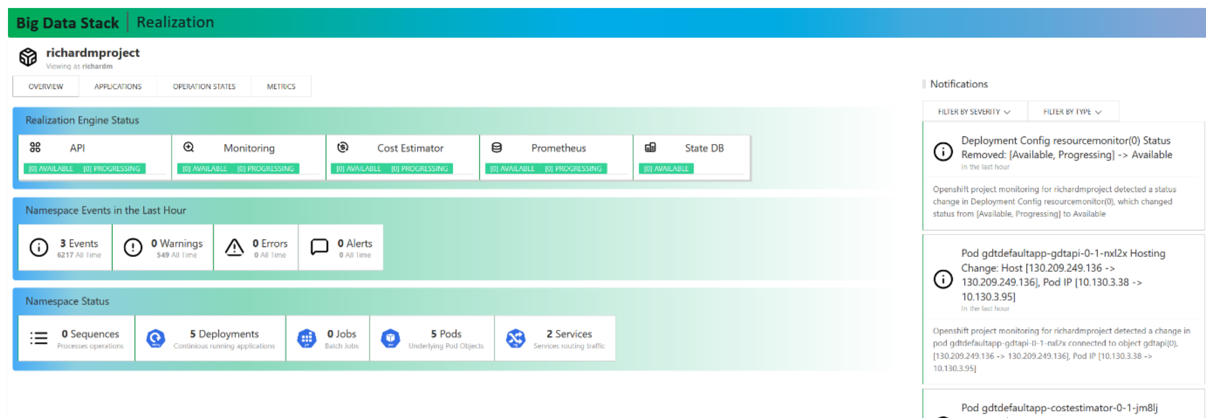


Figure 10: Realization UI Namespace Overview

The second screen of the Realization UI is the Applications screen, which is shown in Figure 11. This screen provides the user access to information about the different applications that are registered to the current namespace/project as well as allows them to trigger actions for those applications. The main part of the view is comprised of a list of registered applications along with descriptions for those applications. Clicking on an application will load information about that application. In particular four pieces of information are checked and displayed:

- **Available Operations and Sequences**: This is the list of actions that the user can perform for the application. By default, a request is made to the Realization API to get all operation sequences registered for the application, which are then rendered within the UI, from where the user can choose to trigger them. However, if the user application contains application states and sequences associated to those states, then only the sequences that are valid for the active states will be shown.
- **External Endpoints**: This renders a list of BigDataStack Objects of type 'Route', which represent external HTTP endpoints being exposed by the application, such as application specific user interfaces.
- **Active Deployments**: This renders the list of currently running application components, along with their states.
- **Ended Deployments**: This renders the list of ended (deleted or failed) application components.

bigdatastack.eu

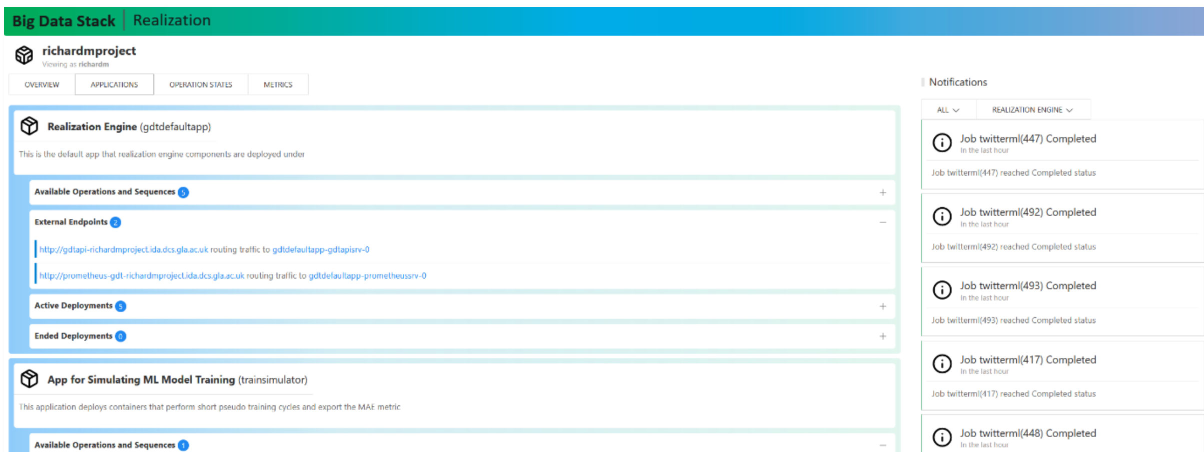


Figure 11: Realization UI Applications View

The third screen within the UI is the Operation States screen, which is illustrated in Figure 12. The goal of this screen is to provide the user more insights into the state of applications while actions are taking place. The user can either enter this screen via the tab at the top of the UI, or they will be sent here if they trigger the launch of a new operation sequence from the Applications screen. As with the Applications screen, the main view is comprised of a list of applications registered to the namespace/project. However, instead of focusing on application information, this view shows information about the different operation sequences triggered for each application. In particular, for an application, operation sequences are grouped by their current state (Running, Complete, Failed or Pending). When an operation sequence is clicked, it will expand to show a detailed state view for that sequence. In particular, the sequenceID and instance is shown at the top, with the creation time and stage information shown directly below. Below that is the sequence description followed by a breakdown of the sequence state. The left-hand pane lists the different BigDataStack Operations that comprise the sequence, along with their individual states, followed by information about any custom parameters that were set for the sequence. Meanwhile the right-hand pane lists any events that were generated by the current operation sequence, where events are colour coded. It is worth noting that the Operation States screen is dynamic, in that any operation sequence that is currently running will have its information automatically updated within the screen when a state change is detected. Furthermore, operation sequences in Running or Pending states can be cancelled by the user by pressing a button.

bigdatastack.eu

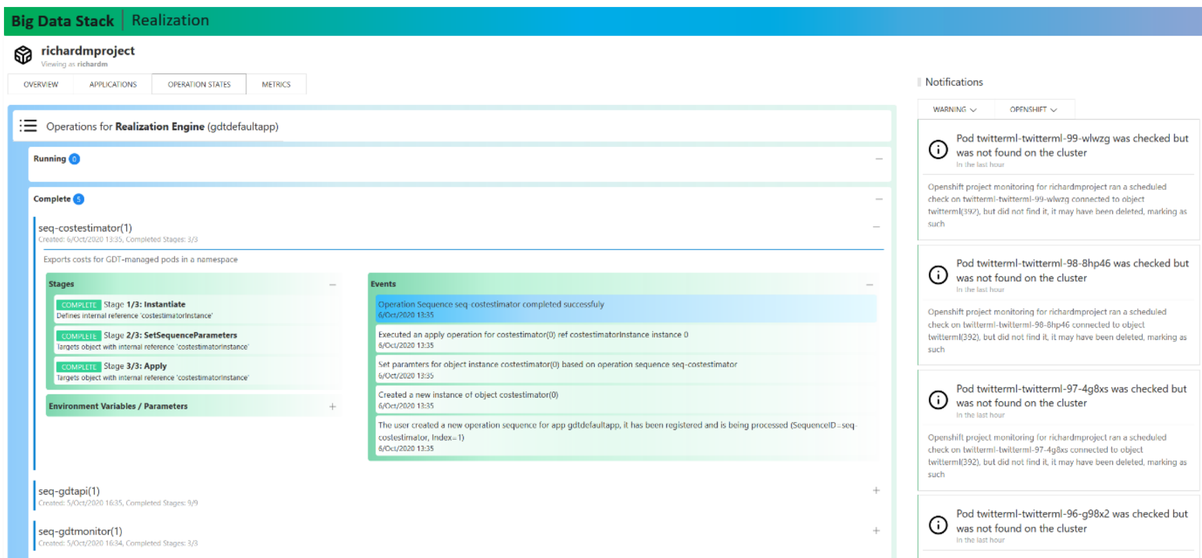


Figure 12: Realization UI Operation States View

Through the use of the Realization UI, the user can check to see what application components are currently active, track changes as they are made to the application and intervene (cancel) that sequence if needed. They are also able to manually trigger new operation sequences (actions) as needed, as well as monitor application status and changes via the events system. This solves the first limitation that was discussed previously in Section 6.1, as well as meeting REQ-RE-05.

6.6.3. Cluster Monitoring

One of the challenges when adding an additional modelling layer on-top of Kubernetes and OpenShift is how to enable state tracking (REQ-RE-03). When a BigDataStack Object is deployed onto a Kubernetes/OpenShift cluster, associated Kubernetes/OpenShift objects are created (e.g. a DeploymentConfig or Job), which may then spawn further objects (e.g. Pods). To facilitate the effective management of BigDataStack Objects, the run-time state of those objects is needed, which is derived from the states of the underlying Kubernetes/OpenShift objects. Hence, a service is needed that generates states for each BigDataStack Object by analysing the states of the associated Kubernetes/OpenShift objects on the cluster. This is the role of the Cluster Monitoring service.

Functionally, the cluster monitoring service periodically (every 10 seconds) looks up the list of BigDataStack Objects for which associated state information can be collected, i.e. any BigDataStack Object that will result in a Pod object being spawned on the cluster. It then sequentially processes each BigDataStack Object in turn, querying the cluster via the OpenShift API to determine the states of the associated Kubernetes/OpenShift objects, to determine the BigDataStack Object state. In most cases, this simply replicates the current states assigned to the Kubernetes/OpenShift object to the BigDataStack Object. However, as the cluster can be in states where a BigDataStack Object exists but no underlying Kubernetes/OpenShift object exists (e.g. because the underlying object was deleted or has not been created yet), additional logic exists to detect and set appropriate states for those scenarios.

bigdatastack.eu

Furthermore, the cluster monitoring service has a second function, which checks all Pod objects running in the managed namespace/project, maps them to BigDataStack Objects where possible, and saves their states in the Application State DB. The reason for this is two-fold. First, it enables Pod states to be queried within the Realization Engine for a given BigDataStack Application or BigDataStack Object. Second, it enables changes in the underlying Pods connected to a BigDataStack Object to be detected, tracked and exposed (such as cases where a Pod is moved to a different physical host).

The final function of the cluster monitoring service is as an event generator. Any changes detected by either function of the cluster monitoring service will result in a BigDataStack Event being generated and published. By default, these events will be saved in the Application State DB for the containing BigDataStack Application. Meanwhile, if a RabbitMQ instance is available, the Event will also be published as a push notification. In this way, application changes are exposed in such a way that they can be used as triggers for orchestration.

6.6.4. Operation Sequence (Container Service)

The Operation Sequence container service is a unique service within BigDataStack, in that it is not a continuous service. Instead, an operation sequence container is a temporary service that is solely concerned with processing a BigDataStack Operation Sequence that has been triggered (either by the application engineer or by some other orchestration service). The reason for this service is that as operation sequences become more complex, it can take multiple minutes to complete them and depending upon the operations involved, may require a non-negligible amount of resources. Hence, it is good practice to separate out the processing of an operation sequence from the rest of the platform.

When an operation sequence is triggered within the Realization Engine, internally this first takes the operation sequence template and generates an instance from it, and stores that instance in the Application State DB. The Realization Engine then creates a new Pod object on the Kubernetes/OpenShift cluster to run the operation sequence service targeting the new instance. Once the Pod object has been created, the responsibility for that operation sequence is passed to the Pod, freeing the Realization Engine for other work. Once the new Pod reaches running state, it will first load the target BigDataStack Operation Sequence instance from the Application State DB. Subsequently, it will process each BigDataStack Operation within the sequence in order, reporting operation outcomes as BigDataStack Events, whilst simultaneously updating the state information housed within the BigDataStack Operation Sequence instance itself. Once the operation sequence is complete, the Pod exits, freeing those resources back into the cluster.

In the next section, we describe the different BigDataStack Operations that can be included within a BigDataStack Operation Sequence.

6.7. Generic BigDataStack Operations

As described previously, within the Realization Engine the different ‘actions’ that the user can perform on an application are defined in terms of atomic BigDataStack Operations,

where multiple such operations can be combined into a BigDataStack Operation Sequence. A single BigDataStack Operation conceptionally performs a single alteration or deployment action on a BigDataStack Object. By combining different BigDataStack Operations together, the application engineer can encode complex processing logic into actions that can be executed with a single click.

The Realization Engine provides several built-in operations that enable common tasks to be performed on the cluster. We can divide these into generic operations (Instantiate, SetParameters, GetParameterFromObjectLookup, Deploy, ExecuteCMD, Build, Delete, Scale, Wait and WaitFor) and BigDataStack-specific operations (RecommendResources, Apply, RegisterWithDynamicOrchestrator, Benchmark and GetResourceTemplates). Each operation loads a configuration mapping when first initialized, which is how an application engineer can customise an operation for their particular application. In the remainder of this section we will describe the generic set of operations. Information regarding BigDataStack-specific operations can be found alongside their associated component descriptions (e.g. information about RecommendResources that is part of ADS-Ranking can be found later in this deliverable in Section 8.4.2).

6.7.1. Instantiate

The Instantiate operation is a core part of the deployment process of a user application. As discussed earlier, when the application engineer registers a component of their application, that component is stored as a BigDataStack Object template. However, it is not templates that are deployed, but instead instances produced from those templates. The instantiate operation is responsible for generating a BigDataStack Object instance from a BigDataStack Object template.

When an Instantiate operation is started, it first loads the 'objectID' of the BigDataStack Object template that it targets. Assuming a valid template is found, it then clones that template and assigns it a unique instance number, and stores the new BigDataStack Object instance back to the Application State Database. The final step of the instantiate operation is to record a mapping between the new instance and a reference key, such that the new instance can be referred to by subsequent operations within a surrounding sequence. This is achieved by loading a user-defined key 'defineInstanceRef' from the operation configuration and then storing a mapping between that key and the <objectID, instance> pair that uniquely identifies the new object instance. An example operation configuration for an instantiate operation is shown below:

```
className: Instantiate # Spawn a new copy from the template
objectID: "feedbackcollector"
defineInstanceRef: "feedbackcollectorInstance"
```

Figure 13: Instantiate Operation Configuration

6.7.2. SetParameters

The SetParameters operation is a generic operation that enables placeholder values within a BigDataStack Object to be replaced with defined parameters. In particular, a BigDataStack Object is allowed to contain placeholder values, which are represented by a key token,

where key can be any basic character string. The idea is that when an application engineer initially configures an operation sequence, there may be information needed that cannot be known until run-time, or that they want to be set at run-time. For example, if a component requires the IP address of a database that is deployed earlier in the operation sequence, then that cannot be known until the database starts. Otherwise, in the case of machine learning jobs, the application engineer may deliberately leave configurable such as hyperparameters as placeholders, such that they can easily launch multiple learning jobs with different parameter sets using the same operation sequence.

SetParameters sources a set of key-value pairs to replace placeholders with from the parameters field in the containing operation sequence (see Section 6.3.4), which is a string to string map. In particular, for each key 'k' within that map, it performs a regular expression search for all instances of 'k' and replaces any matches with the associated value from the map. The parameter map itself can be populated in three main ways:

- **Automatic Application and Object Population**: By default, the Realization Engine will insert key-value pairs detailing information about the application and target object into this mapping. This will typically provide values for: owner, namespace, appID, objectID and instance.
- **User Specified Defaults within the Operation Sequence**: The operation sequence definition contains a parameters field where values can be set.
- **User Specified Values Provided at Trigger Time**: When the user triggers an operation sequence, they can optionally provide a set of key-value pairs that are used to update the parameter map.
- **Via Operations**: Other operations may update this parameter map as the operation sequence progresses.

To enable SetParameters to only alter a specified BigDataStack Object, which may have been created via a previous Instantiate operation, the operation configuration for SetParameters requires an 'instanceRef' value, as illustrated below:

```
className: SetParameters # Fill in any missing placeholders
instanceRef: "feedbackcollectorInstance"
```

Figure 14: Set Parameters Operation Configuration

6.7.3. GetParameterFromObjectLookup

A relatively common scenario is where we need to set an application parameter at run-time through an object look-up. An example use-case here could be that we need to determine at run-time the name or IP address of a dependant Pod or Service, which could not have been known when the operation sequence was originally created. This is the role of the GetParameterFromObjectLookup operation. More precisely, this operation performs a query of OpenShift/Kubernetes objects and sets a parameter within the containing operation sequence based on the response (e.g. that can be used later by the SetParameters operation).

```
className: GetParameterFromObjectLookup # Generates a parameter value by querying the cluster state
parameter: "serviceFeedbackCollectorLookup"
criteria: "$namespace$:service:$appID$-srv-feedbackcollector-.*"
multipleMatches: "SelectFirst"
```

Figure 15: Get Parameters from Object Lookup Operation Configuration

The GetParameterFromObjectLookup operation takes as input a key ('parameter') which is the parameter name that will be written into the parent operation sequence parameter map. To facilitate the search operation over the OpenShift/Kubernetes objects on the cluster, it also takes as input a 'criteria' string, which is the query, and a 'multipleMatches' field that specifies what to do if multiple objects are found that match the query. The query is formatted as follows:

- <**namespace/project** to search>:<**object type**>:<**object name** java regular expression>

The multiple matches criteria can then be any of the following:

- **SelectFirst**: It will select the first object found as the parameter value
- **Allow**: It will write an array containing all matched objects as the parameter value

6.7.4. Deploy

Deploy is a simple operation that takes a BigDataStack Object instance and then creates the underlying OpenShift/Kubernetes Object on the cluster using the built-in OpenShift operation client within the Realization Engine. This operation is rarely used in BigDataStack, as the 'Apply' operation (see Section 8.4.3) provides similar base functionality, while integrating with ADS-Deploy to provide additional functions. In practice, Deploy acts as a back-up option in scenarios where the Realization Engine is deployed stand-alone without ADS-Deploy. Deploy uses an 'instanceRef' field in its configuration to identify the BigDataStack Object instance to target for deployment. This is typically defined as part of the 'Instantiate' operation, although it can also be set through a GetParameterFromObjectLookup operation.

```
className: Deploy # Create deployment config on the cluster
instanceRef: "feedbackcollectorInstance"
```

Figure 16: Deploy Operation Configuration

6.7.5. ExecuteCMD

Within BigDataStack, some of the Pilot use-cases involve what we refer to as 'tier 2' applications. These are applications that require multiple consecutive steps to deploy, because they first need to deploy a management framework, and then once that is ready, launch the true application on-top of the management framework. Apache Spark is a common example of a tier 2 app, where the spark cluster first needs to be deployed

(comprised of master and worker nodes), and then once the spark cluster reports ready, the application is launched by submitting the application Jar to one of the master nodes. To make this type of applications possible, we need the ability to queue operations to perform the different steps, as provided by operation sequences. However, we also (like in the case of Spark) need the ability to execute commands on a container to complete the process (e.g. submitting a Spark Job Jar to a master node). This is the role of the ExecuteCMD operation.

```
className: "ExecuteCMD"
objectID: "spark-master"
instancelookupCriteria: "first"
commands:
 - - "bin/spark-submit"
   - "--class"
   - "org.bigdatastack.wl.streaming.MainProcess"
   - "--master"
   - "spark://0.0.0.0:7077"
   - "--deploy-mode"
   - "cluster"
   - "/tmp/als-spark-streaming-assembly-0.1.jar"
```

Figure 17: ExecuteCMD Operation Configuration

The ExecuteCMD operation targets a BigDataStack Object instance, and then can run a command on one or more of any underlying Pods (and associated containers) connected to that object. To achieve this, it takes as input from its configuration an 'objectID' (identifying the BigDataStack Object instance and a 'instanceLookupCriteria' field, which can have the following values:

- **First**: The commands will be run only on the first running Pod connected to the target object instance (if a Pod has multiple containers, then the command will be attempted on all containers).
- **All**: The commands will be run on all running Pods connected to the target object instance.

Commands to run are specified as a two-layer array via the 'commands' value in the configuration. The first layer array represents the different commands to run in sequence, while the second layer of the array contains the components of each command to run.

6.7.6. Build

The Build command is a simple operation that enables the triggering of a container build process within OpenShift, using its source-to-image sub-system. It takes as input within its configuration a target BigDataStack Object, which must be a template and be of type BuildConfig, otherwise the operation will fail. The operation will then simply trigger the start of the build process. This operation is almost always followed by a WaitFor operation targeting the same object, i.e. waiting for the new image to be ready.

bigdatastack.eu

```
className: "Build" # build the database image
objectID: "bc-lxsdb"
```

Figure 18: Build Operation Configuration

6.7.7. Delete

The delete command enables a user to delete the underlying OpenShift/Kubernetes objects for one or more BigDataStack Object instances. Note that this does not delete the BigDataStack Object instance itself, which will remain with a 'deleted' state (this is to allows for persistent history for a BigDataStack Object). To identify the object to perform the deletion for it uses an 'instanceRef' in the same way as the 'Deploy' operation. This is typically set via a previous GetParameterFromObjectLookup operation which performs a run-time look-up of the object(s) to delete.

```
className: "Delete" # Delete objects
instanceRef: "spark-workers"
```

Figure 19: Delete Operation Configuration

6.7.8. Scale

The Scale operation provides an in-built method for altering the replication factor for BigDataStack Object instances of type DeploymentConfig at run-time. In effect, this allows continuous applications a means to scale up and down in response to the environment (e.g. user traffic volumes). However, in general, it is not recommended to use this operation for Realization managed apps. Instead, scaling should be handled through the creation/deletion of instances for the BigDataStack Object template, rather than altering the replication factor of an existing instance. The reason for this is that the Realization Engine can track (and report events) for individual object instances, but cannot distinguish between different replicas for the same instance when reporting events, making data-driven orchestration more difficult.

```
className: "Scale" # Alter scaling factor
instanceRef: "sparkWorkerInstance"
replication: "+2"
```

Figure 20: Scale Operation Configuration

To identify the object(s) to scale, it uses an 'instanceRef' in the same way as the 'Deploy' and 'Delete' operations. This is typically set via a previous GetParameterFromObjectLookup operation which performs a run-time look-up of the object(s) to scale. The other configuration provided to the Scale operation is 'replication', which sets the replication factor. This is a string, that can either contain a target replication number (e.g. "3") or can be a relative adjustment (e.g. "+1" or "-1").

CC BY NC ND

6.7.9. Wait

The Wait operation simply inserts a pre-defined wait in seconds before starting the next operation. This can be used in cases where a fixed amount of time is needed for some internal application process to complete that is not exposed by the Pod state. The Wait operation takes only a single configuration value 'seconds', which is the number of seconds to wait for.

```
className: "Wait" # Wait 10 seconds
seconds: "10"
```

Figure 21: Wait Operation Configuration

6.7.10. WaitFor

The WaitFor operation enables an application sequence to pause until a particular BigDataStack Object instance reaches a pre-defined state. This is often used where there are dependencies between application components, where one needs to be available (e.g. a database) before the next can start.

```
className: WaitFor # Wait until the deployment config is reporting available
instanceRef: "kafkaInstance"
waitForStatus: "Available"
```

Figure 22: WaitFor Operation Configuration

To identify the object(s) to wait for, it uses an 'instanceRef' in the same way as the 'Deploy' 'Scale' and 'Delete' operations. WaitFor is commonly used during deployment operations, and hence the instanceRef is normally generated via the Instantiate operation. Additionally, a 'waitForStatus' value is provided in the configuration, that specifies the state that the BigDataStack Object instance must be in for the wait process to end.

6.8. Summary

In this section we have summarized why the Realization Engine was introduced as well as provide a technical overview for it. In particular, we have provided a summary of the modelling changes to the overall deployment and management of applications within BigDataStack, as well as summarized the services and operations provided by the Realization Engine. At the time of writing, the Realization Engine is fully functional and meets the requirements listed in Section 6.2, although it is under continuous further development. It is currently envisaged that the Realization Engine will undergo an open source release later in 2020 as a stand-alone component.

cc BY NC ND

7. Dynamic Orchestration

The Dynamic Orchestrator (DO) works alongside the Triple Monitoring Engine (TME) to monitor and trigger the redeployment of BigDataStack applications during runtime to ensure they comply with their Service Level Objectives (SLOs.) The DO receives and manages monitoring requests when a new application or service is deployed into the BigDataStack platform, informing the TME and the Quality of Service (QoS) component what metrics and SLOs should be monitored. When any violation to these SLOs exist, the QoS informs the DO, and the DO is in charge of deciding what redeployment change is necessary, if any.

Since month 18, we have worked on several improvements for the DO:

- In M18, the DO decision mechanism was based on a Tabular Q-learning logic; this has been updated to use Deep Q-learning, in particular DQN (Mnih, 2013), allowing us to deal with a larger action space and a continuous state space, two improvements that make our Reinforcement Learning (RL) algorithm more flexible and adequate to deal with the complexity needed for orchestrating BigDataStack applications.
- We have developed a novel Reinforcement Learning-based approach called Tutor4RL, which combines domain knowledge with machine learning for achieving a good initial performance, a common problem in RL and in particular for DQN. This is done through programmable functions – called guide functions - that guide the behavior of the agent in its initial steps and until the agent gathers sufficient experience to manage the application properly.
- In Y3, we have introduced constrain functions, to supervise the behavior of the agent at every point, avoiding unnecessary and incorrect changes in the deployment of applications. This results in a more stable and robust behavior for the DO without sacrificing the agent's learning capabilities.
- In addition, in Y3 we have implemented and tested different guide and constrain functions that generally work for most applications, adjusting the general Tutor4RL framework to be able to manage changing action and state spaces, necessary to manage multiple BigDataStack applications with different metrics, SLOs and redeployment actions.
- Finally, we have integrated the DO with the Data-as-a-Service layer of BigDataStack, which contains stateful components such as the Adaptable Distributed Storage (ADS) and the Complex Event Processing (CEP), for adapting these components dynamically during runtime as described in section 6.2.1.

7.1. Requirements

Modified requirements:

	Id	Level of detail	Type	Actor	Priority
	REQ-DO-01	System	FUNC	Application Engineer, Data Engineer	MAN
Name	Playbook Enrichment				

cc BY NC ND

Description	The Dynamic Orchestrator shall ingest the monitoring request when an application or service is deployed and enrich its playbook with information about the QoS metrics and intervals to be considered by the Triple Monitoring to monitor the QoS during runtime.
Additional Information	N/A

Table 3 - Requirement (1) for Dynamic Orchestrator

	Id	**Level of detail**	**Type**	**Actor**	**Priority**
	REQ-DO-02	Stakeholder	FUNC	Application Engineer, Data Engineer	MAN
Name	Runtime Re-deployment				
Description	When an application or service is running, the Dynamic Orchestrator shall determine if a deployment change should be performed when there is a violation of an application requirement or Service Level Objective (SLO) and send a signal to the Realization Engine to trigger a change in the deployment to try to satisfy the requirements or SLOs.				
Additional Information	The Triple Monitoring detects this violation and sends an alert to the Dynamic Orchestrator to start this process.				

Table 4 - Requirement (2) for Dynamic Orchestrator

	Id	**Level of detail**	**Type**	**Actor**	**Priority**
	REQ-DO-04	System	FUNC	Application Engineer, Data Engineer	MAN
Name	Resources Limits				
Description	The orchestrator shall be able to retrieve the possible actions, or sequences, for each application through the Realization Engine, and use this information in its own decisions.				
Additional Information	The complete list of deployment parameters might vary according to the application/service and its actual deployment.				

Table 5 - Requirement (3) for Dynamic Orchestrator

	Id	**Level of detail**	**Type**	**Actor**	**Priority**
	REQ-DO-07	System	FUNC	Application Engineer	DES
Name	Orchestration of Data-as-a-Service components				
Description	The orchestrator will also orchestrate the redeployment of stateful components of the Data-as-a-Service layer, such as the Adaptable Distributed Storage and CEP components, for improving applications' performance in terms of adaptable storage and replication of CEP queries.				

cc BY NC ND

Additional Information	N/A

Table 6 - Requirement (4) for Dynamic Orchestrator

7.2. State-of-the-Art: RL for Applications' Configuration

There is extensive research in the fields related to the use of RL for applications and systems' configuration. Natural Adaptive Video Streaming with Pensieve[6] presents a system that generates adaptive bit rate (ABR) algorithms using RL. These algorithms are used for video streaming and must balance a variety of Quality of Experience (QoE) goals. This work successfully uses a variant of deep RL, A3C, to create algorithms that adapt to a wide range of environments and QoE. In Chameleon[7], the performance of video analytics applications is optimized by performing automatic adaptation of its configurations. The application's behavior is customized to the execution context by selecting different parameter configurations; the best parameter configuration is selected by a logic inspired by greedy hill climbing combined with periodical online profiling. However, these two works are centered around applications that use deep convolutional neural networks for video processing/streaming use cases, while in BigDataStack we aim to offer a flexible orchestration logic that can be applied to any kind of application.

In addition, there are also different approaches for bootstrapping RL, obtaining a better performance from the first moment the agent begins to operate. A simple approach is to explore the state space randomly, but this approach is usually time-consuming and costly when the state/action space is large. The drawback of this approach has been reported by our previous study[8] in the case of leveraging RL to automatically decide the configuration and deployment actions of a data processing pipeline in a cloud and edge environment. Another approach, is to gain experience via simulation. With enough computational resources we can easily produce lots of experience data in a short time, but it is difficult to ensure that the simulated experiences are realistic enough to reflect the actual situations in the observed system.

Recently, there has been a new trend to leverage external knowledge to improve the exploration efficiency of RL agents. For example, in [9] and [10], prior knowledge like pre-trained

---

[6] Mao, H., Netravali, R., & Alizadeh, M. (2017, August). Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (pp. 197-210).

[7] Jiang, J., Ananthanarayanan, G., Bodik, P., Sen, S., & Stoica, I. (2018, August). Chameleon: scalable adaptation of video analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (pp. 253-266).

[8] Argerich, M. F., Cheng, B., & Fürst, J. (2019, April). Reinforcement learning based orchestration for elastic services. In *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)* (pp. 352-357). IEEE.

[9] Moreno, D. L., Regueiro, C. V., Iglesias, R., & Barro, S. (2004). Using prior knowledge to improve reinforcement learning in mobile robotics. *Proc. Towards Autonomous Robotics Systems. Univ. of Essex, UK.*

[10] Hester, T., Vecerik, M., Pietquin, O., Lanctot, M., Schaul, T., Piot, B., ... & Osband, I. (2017). Deep q-learning from demonstrations. *arXiv preprint arXiv:1704.03732*.

CC BY NC ND

models and policies are used to bootstrap the exploration phase of a RL agent. However, this type of prior knowledge still originates in previous training and is limited by the availability of such data.
Instead of relying on any pre-trained model, we explore how to utilize a set of programmable knowledge functions to guide the exploration of a RL agent so that we can quickly bootstrap a RL agent to make effective decisions, even just after a few exploration steps. We call our method Tutor4RL.Unlike existing approaches, Tutor4RL requires not any previous training. Therefore, it is a more practical approach for the use of RF in real systems. To the best of our knowledge, Tutor4RL is the first to apply programmable knowledge functions into RL for improving the sample efficiency problem of RL.

7.3. Design Specifications

During the second phase of the project, we have Finalized the development of our new approach called ***Tutor4RL***. Tutor4RL takes as input domain knowledge guidelines that are used to constraint, explore and learn from the environment in which the agent is deployed, while learning from its own experience the best actions to achieve its goal in different states.

We have modified the RL framework by adding a component we call the Tutor. The tutor possesses external knowledge and helps the agent to improve its decisions, especially in the initial phase of learning when the agent is inexperienced. In each step, the tutor takes as input the state of the environment and outputs the action to take, in a similar way to the agent's policy. However, the tutor is implemented as a series of programmable functions that can be defined by domain experts and interacts with the agent during the training phase. We call these functions knowledge functions and they can be of two types:

- Constrain functions: are programmable functions that constrain the selection of actions in a given state, “disabling” certain options that must not be taken by the agent. For example, if the developer of the application has decided a maximum budget for the application, even the application load is high and this could be fixed by adding more resources to the deployment, this should not be done if the budget of the user has already reached its maximum.
- Guide functions: are programmable functions that express domain heuristics that the agent will use to guide its decisions, especially in moments of high uncertainty, e.g. start of the learning process or when an unseen state is given. Each guide function takes the current RL state and reward as the inputs and then outputs a vector to represent the weight of each preferred action according to the encoded domain knowledge. For example, a developer could create a guide function that detects the number of current users for an application and if the number is higher than a certain threshold, more resources might be deployed for the application.

bigdatastack.eu

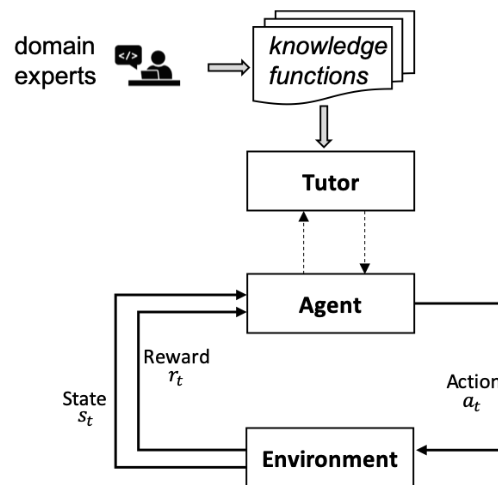


Figure 23: High level vision of Tutor4RL

The benefit coming from using Tutor4RL is twofold:

- During training, the tutor enables a reasonable performance, opposed of the unreliable performance from an inexperienced agent, while generating experience for the agent's training. Furthermore, the experience generated by the tutor is important because it provides examples of good behaviour, as it already uses domain knowledge for its decisions.
- The knowledge of the tutor does not need to be perfect or extensive. The tutor might have partial knowledge about the environment, i.e. know what should be done in certain cases only; or might not have a perfectly accurate knowledge about what actions should be taken for a given state. Instead, the tutor provides some "rules of thumb" the agent can follow during training, and based on experience, the agent can improve upon the decisions of the tutor, achieving a higher reward than it.

The main functioning of Tutor4RL is as follows:

1. Application developer (i.e., the domain expert) defines guide and constrain functions These functions encode domain knowledge of the developer that guide and constrain the RF agent during its initial stage. This is important for new applications and/or a new system execution context, where traditional RL would need to explore the state space randomly and thereby negatively impact QoS of the application. If the application has been deployed before, Tutor4RL can use the historical data from that previous deployment and encodes it as a guide function.
2. The Triple Monitoring Engine and QoS Evaluation informs the Interpreter about the current system metrics and the SLO violations, respectively.
3. These metrics are taken as input by the agent and the tutor and both output a vector with valuations for each action.

The RL agent selects an action, from its policy or from the suggestions provided by the tutor, which should be executed by the Realization Engine and sends it.

CC BY NC ND

7.3.1. Adaptable Distributed Storage and Complex Event Processing Interplay

The Adaptable Distributed Storage and Complex Event Processing (CEP) components (as described in 4.2) will interact with the DO to scale in/out its resources. As a result, the storage can be re-configured automatically, moving data regions across its current nodes and scale in or out to be adapted under diverse workloads. And CEP sub-queries can be replicated to increase throughput and process a higher number of events per second if the system was overloaded with the previous configuration. To realize these redeployments, the DO monitors several metrics related to each of these components and triggers the changes in deployment when necessary via a request to the component.In the case of the Adaptable Distributed Storage, the re-configuration can be started by the Elasticity Manager, a subcomponent of the Adaptable Distributed Storage, or by the DO. The DO needs to consider that there is a second dynamic adaptation mechanism acting at the storage layer level. This second adaptation component (i.e. Elasticity Manager) will request the DO for more resources if needed; in fact, this has been specified as a requirement imposed on the Adaptable Distributed Storage (see REQ-ADS-06) by the DO. More specifically, the Adaptable Distributed Storage will notify information regarding pending redeployments of the storage, when the process of data reconfiguration starts and finishes, along with the current deployment of this layer.

In the setting of Tutor4RL, the Adaptable Distributed Storage logic is seen as a Guide function, so it is used by the agent to improve its performance. This information helps the DO to determine in what cases the Adaptable Distributed Storage should be scaled up or down, first by observing the behaviour of the already implementing logic, and then repeating and potentially, improving these decisions thanks to having a broader picture of the application and system status.

The communication with both, the Adaptable Distributed Storage and the CEP is implemented via REST API calls. These calls are structured as follows:

1. Monitoring request: everytime a new CEP query or an ADS engine is launched, a new monitoring request is sent to the DO, with information about the SLOs and metrics to be monitored.
2. Once the query or engine is running and a change is needed, a request for redeployment can be sent
    2.1. The ADS requests the DO to scale up, the DO will return “True” if resources allow it and update its internal state (learning step.)
        2.1.1. If resources do not allow this change, the DO will return False and flow finishes here. ■
    2.2. DO requests ADS/CEP to scale up/down, the component will return “True” if its state allows this change
        2.2.1. If the state does not allow this change, the component returns “False” and flow finishes here. ■
        2.2.2. If the request is to scale down and the component’s state allows this, the ADS/CEP will scale down and the flow finishes. ■
3. The DO will request the Realization Engine (RE) to scale up the CEP/ADS.
4. The RE will execute the deployment change. ■

CC BY NC ND

More information about the protocol between the DO and the Data-as-a-Service components can be found at the D4.3.

7.3.2. CEP Integration with the Infrastructure building block of BigDataStack

The CEP interacts with the mechanisms described in Section 6.4.1 of D4.3 to integrate with the infrastructure building block. The CEP driver implements the methods needed by the infrastructe (Section 6.5.2 of D4.3): canYouScale, infrastructureFinishedScaling and infrastructureFinishedScalingDown.

7.3.3. canYouScale method

The canYouScale method sends a can_scale action to the orchestrator indicating the query and subquery to be scaled and checks if the subquery is registered and deployed in the system. If the subquery is not registered, the orchestrator sends back a response of type SQ_NOT_ABLE_TO_SCALE. If the subquery is deployed, checks if the subquery is stateful and if the tuples are not grouped by some fields, the subquery cannot be scaled and the SQ_NOT_ABLE_TO_SCALE message is sent Back. Otherwise, if the query is stateless or is stateful and the tuples are grouped in different windows, the subquery can be scaled and the response message of type SQ_CAN_BE_SCALED is sent back.

7.3.4. infrastructureFinishedScaling method

The infrastructureFinishedScaling method sends a scale_out action to the orchestrator indicating the pod where the new subquery instance is going to be deployed, the query, subquery and subquery instance to scale out. Once the orchestrator receives all this information starts the scale out process.s

7.3.5. infrastructureFinishedScalingDown method

The infrastructureFinishedScalingDown method is called at least three times by the Dynamic Orchestrator (DO) component (WP3). The can_scale_down action is sent to the orchestrator to check if the subquery instance to be scaled down is deployed in the system and if there is more than one instance of this subquery running. A message is returned to the DO with the answer, YES or NO.

The second time the DO calls this method, the orchestrator replies with a WAIT message, specifying the amount of seconds to wait (30 seconds) and the pod where the subquery removed was running. Then, the DO waits in order to check if the scale_down process has finished. Next time the DO calls this method, a message with a YES or WAIT will be send back. If the answer is YES, the DO can remove the pod where the subquery instance was running in order to save resources.

CC BY NC ND

7.3.6. Interplay with the Realization Engine

To enable applications to be registered with BigDataStack to be managed by the Dynamic Orchestrator (DO), the DO needs to integrate with the Realization Engine to obtain information about the application (components) to manage. This is achieved via a BigDataStack-specific Operation (see Section 6.3.3), 'RegisterWithDynamicOrchestrator' that can be included within an application deployment. In particular, once an application component, represented by a BigDataStack Object instance has been launched (either by the Deploy or Apply operations) and has reached running state, this new operation can be called, which will pass information about the component and any associated service level objectives to the DO, such that data-driven orchestration can commence.

```
className: "RegisterWithDynamicOrchestrator" # Start data-driven orchestration
instanceRef: "feedbackCollectorInstance"
```

Figure 24: Register with Dynamic Orchestrator Operation Configuration

To identify the target object to orchestrate, it uses an 'instanceRef' in the same way as the 'Deploy', 'Scale' and 'Delete' operations. RegisterWithDynamicOrchestrator is normally used during deployment operations, and hence the instanceRef is normally generated via the Instantiate operation. Internally, the RegisterWithDynamicOrchestrator operation queries the Application State Database to collect up-to-date information about the target BigDataStack Object instance, as well as connected information about service level objectives, resource templates and exported metrics. This is then sent via REST API call to the DO, which starts its orchestration process. There is also a related operation 'EndDynamicOrchestration', that notifies the DO to stop orchestration for a BigDataStack Object instance.

7.4. Implementation and Integration Highlights

The DO has been fully designed and implemented to provide the following overall functionality:

1. Every time a new application is launch, the Realization Engine (RE) sends a monitoring request to the DO. This request contains all the information necessary for the DO to track the application: application and object identifiers, metrics and SLOs to monitor and possible redeployment actions to execute for this application.
2. The DO informs the Triple Monitoring Engine (TME) and QoS Evaluation (QoS) about the new application with details about the metrics and SLOs that need to be tracked.
3. Periodically, the TME and QoS inform the DO about the current system metrics and the SLO violations if any, respectively.
4. When all the metrics for a given application have been updated since their last processing, the DO converts them into states and rewards. The states represent the system status as a continuous vector of fixed length. The rewards indicate to the Reinforcement Learning agent if an executed action was "good" or "bad" in terms of

cc BY NC ND

requirements and SLOs compliance (e.g. if the requirements and SLO violations disappeared after the execution of an action).

5. At this point, the DO triggers a new step in the RL (Reinforcement Learning) agent. In this step, the state and reward are fed to the RL agent and the agent selects an action, from its policy or from the suggestions provided by the tutor, that should be executed by the RE. The actions are type of changes in the deployment such as change the number of replicas, change the number of vCPUs or change the vRAM assigned—note these are just some of the changes that are being considered, the full list of deployment changes still needs to be determined. The action to keep the current deployment, called "none" action, is valid for all applications.
6. If the action is different to the "none" action, the DO requests this action to the RE.
7. The process is repeated from step 3.

For Tutor4RL, we have defined a set of guides and constrains that are used for all applications. These guides and constrains work for most applications and provide a starting behaviour for the DO, that will learn from the experience of orchestrating each application and will improve over time upon this default behaviour.

7.5. Experimentation Outcomes

In the first half of this project, we have implemented an early version of the DO using Tabular Q-learning and tested it in simulations of a streaming application in which the load of the application increases (see [44]) for a detailed description and evaluation of this prototype). This streaming application can find lost children based on the processing of camera data. It can be split in two components: (1) an offline module, which is trained with pictures of the child in a server and (2) an online module, a face detection and matching service that is deployed in several devices and is in charge of finding the child (see [44]).

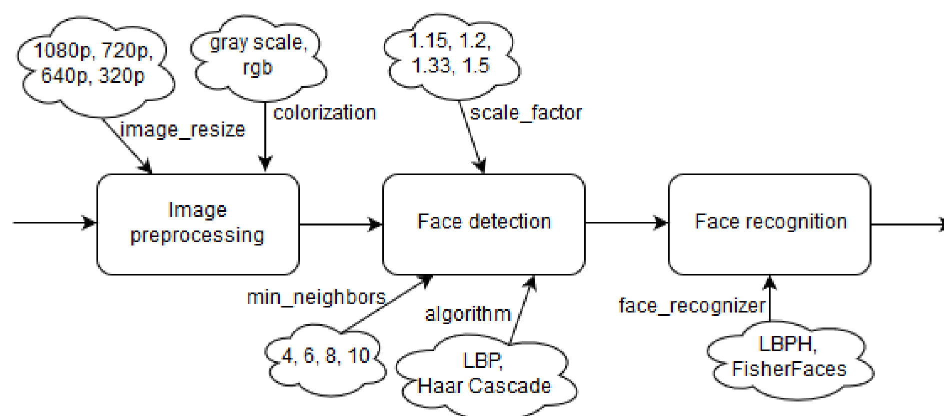


Figure 25: Example of streaming analytics application

We have shown that RL can be used efficiently (up to 25% better precision than a state-of-the-art heuristics) to dynamically orchestrate such a data processing pipeline like the ones in BigDataStack. However, we noticed two issues with applying traditional RL:

i. Bad performance during the "training phase" of the RL agent, and
ii. Missing constrains to avoid clearly wrong actions.

Both issues are very relevant to BigDataStack: BigDataStack applications need to be ready from the start and the DO should ideally avoid clearly wrong actions. We started to address both issues with Tutor4RL.

Tutor4RL adds two features to traditional RL: Guide Functions and Constrain Functions. These functions enable the user to give some initial knowledge to the RL agent to direct its initial exploration.

During the second phase of the project, we implemented a prototype of Tutor4RL with standard RL libraries in order to provide a fair comparison of it against other heavily used RL algorithms. Specifically, we have modified the library Keras-RL to implement a tutored Deep Q-Network (DQN) agent.

An important question towards our model is when the tutor should decide for the agent and vice-versa. In a similar way on how Epsilon greedy exploration works, we defined *Tau* as the threshold parameter for the agent to control when it will use the suggested actions from the tutor instead of using its own. The initial value of *Tau* is a parameter of our model and the best value to initialize it depends on the use in which Tutor4RL is used. This parameter is linearly reduced while the agent gathers more experience and learns to take better decisions.

To test Tutor4RL, we have used the library OpenAI *gym* [42], which provides several environments ready to be used with RL. As we are testing a DQN agent, we decided to use the Atari game Breakout [43] which is a complex use case in which we can observe how the agent performs in cases in which reward is sparse and episodes are long in time steps. This is a different use case than the one we are addressing in BigDataStack, but we have chosen it because it is heavily used in the RL literature, so it lets us compare Tutor4RL with the state-of-the-art in a straightforward manner.

In Breakout, the state of the environment in each time step is the video games' frame in pixels. The actions are four: no operation, fire (which throws the ball to start the game), left and right. The reward is the points achieved in the game, given each time a brick is broken.

We implemented a simple guide function that encapsulates some basic knowledge about the game: the function takes as input each frame, searches for the ball and the position of the bar, and moves the bar to the left if the ball is to the left of the bar or to the right if the ball is on that side. If the ball is not seen, then the action chosen is "fire" to start the game.

We have compared the functioning of Tutor4RL by also training a plain DQN agent for the same use case. The results can be seen in the plot below:

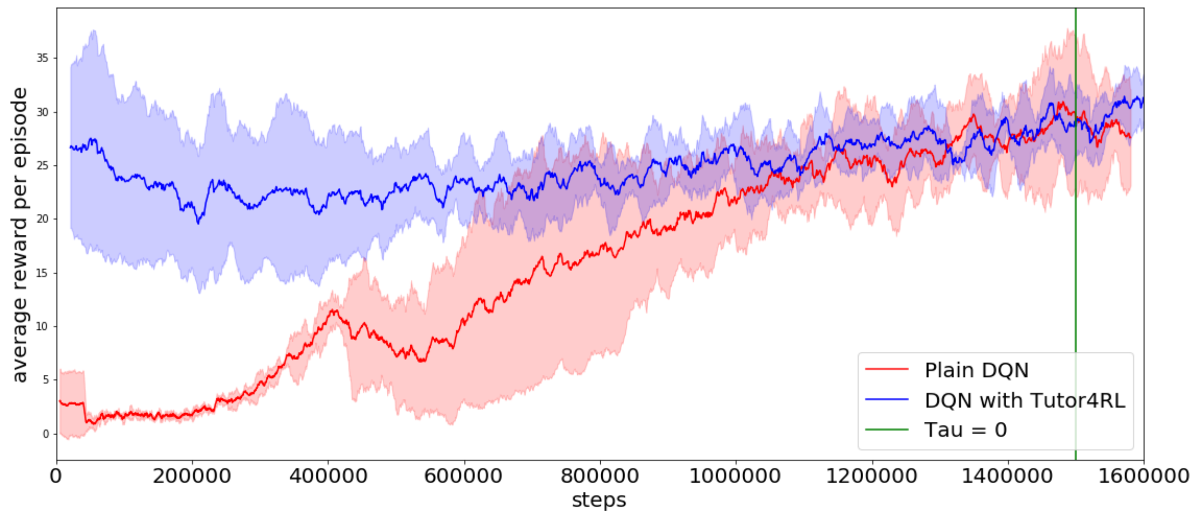


Figure 26: Comparison of performance between Tutor4RL and a plain DQN agent.

As it is possible to see, from the initial steps the DQN agent with Tutor4RL manages to achieve a reasonably high reward while the plain DQN agent performs very poorly, because of its inexperience. As the agents perform more steps, the plain DQN agent catches up, but it's not until step 1 million that it manages to achieve a similar reward to the tutored DQN agent. *Tau* is decreased in every step, starting with a value of 1 and reaching 0 in step 1.5 million. It is important to note that after this step, the tutor is not used anymore but the agent keeps up with its high reward.

In Y3, we completed the implementation of Tutor4RL for the DO. We have added the mechanisms for managing constrain functions, and a way to manage changing state and action spaces with the same Tutor. The latter is a requirement that we have noticed for the BigDataStack use case: each application can have a different state space, i.e. it can have different metrics and SLOs, as well as a different action space, i.e. different sequences or redeployment changes. However, we want the tutor to support all different applications, with guides and constrains that express rule of thumbs for applications' deployment in the cloud. An example of this can be as simple as: "if no SLO is being violated, do not perform any change", and this in fact, constitutes a guide for our agents. A constrain for example, is "if response time or latency are close to a maximum threshold, do not remove replicas of the process". These are simple examples we have included in our current tutor and work in most cases for applications. The set of guides and constrains can be modified by a BigDataStack platform administrator.

We have compared Tutor4RL performance against vanilla DQN in a scenario where the DO is in charge of controlling two metrics: cost per hour (which varies according to resources used by application) and response time. These are two opposite objectives: if we increase the use of resources, the response time decreases but the cost per hour increases, and if we decrease the use of resources, the opposite is true. However, the SLOs specify thresholds for each metric: cost per hour should be less or equal to $0.03 and response time should be less than 200ms.

The DO must find the sweet spot that satisfies these two SLOs as long as the application allows it. In fact, it might happen that the application load is too high, and then there is no way of satisfying both SLOs, in these cases the DO behaviour will tend to find the

bigdatastack.eu

configuration that violates the SLOs proportionally less. However, we believe this are corner cases in which even a human might not be sure what to do and therefore we have not evaluated the DO's performance in these situations.

In Figure 27, we see the performance of the vanilla DQN agent (left) and the Tutor4RL agent (right) for managing this scenario with two SLOs. Note than on the images we have marked with horizontal lines the thresholds for SLOs and with a vertical line, the moment in which the guide functions from the Tutor are not used anymore, until that point the functions are used on and off with a diminishing frequency from 0.9 to 0. As we can see, the Tutor4RL agent performs better than the vanilla agent, reducing the amount of deployment changes performed and achieving a better satisfaction of SLOs as shown in Table 7. We still see that once the guides are completely abandoned, the agent commits some mistakes, but it can quickly correct its error. We can avoid this by adding constrains such as not changing the deployment configuration if no SLO is violated, but we wanted to show a case in which the agent is freer e in its actions and therefore show its learned behaviour better.

SLO satisfaction	Vanilla DQN	Tutor4RL
Response time (< 200ms)	82.37%	84.03%
Cost per hour (< $0.03)	56.23%	95.75%

Table 7 - SLO satisfaction for Vanilla DQN agent vs. Tutor4RL agent

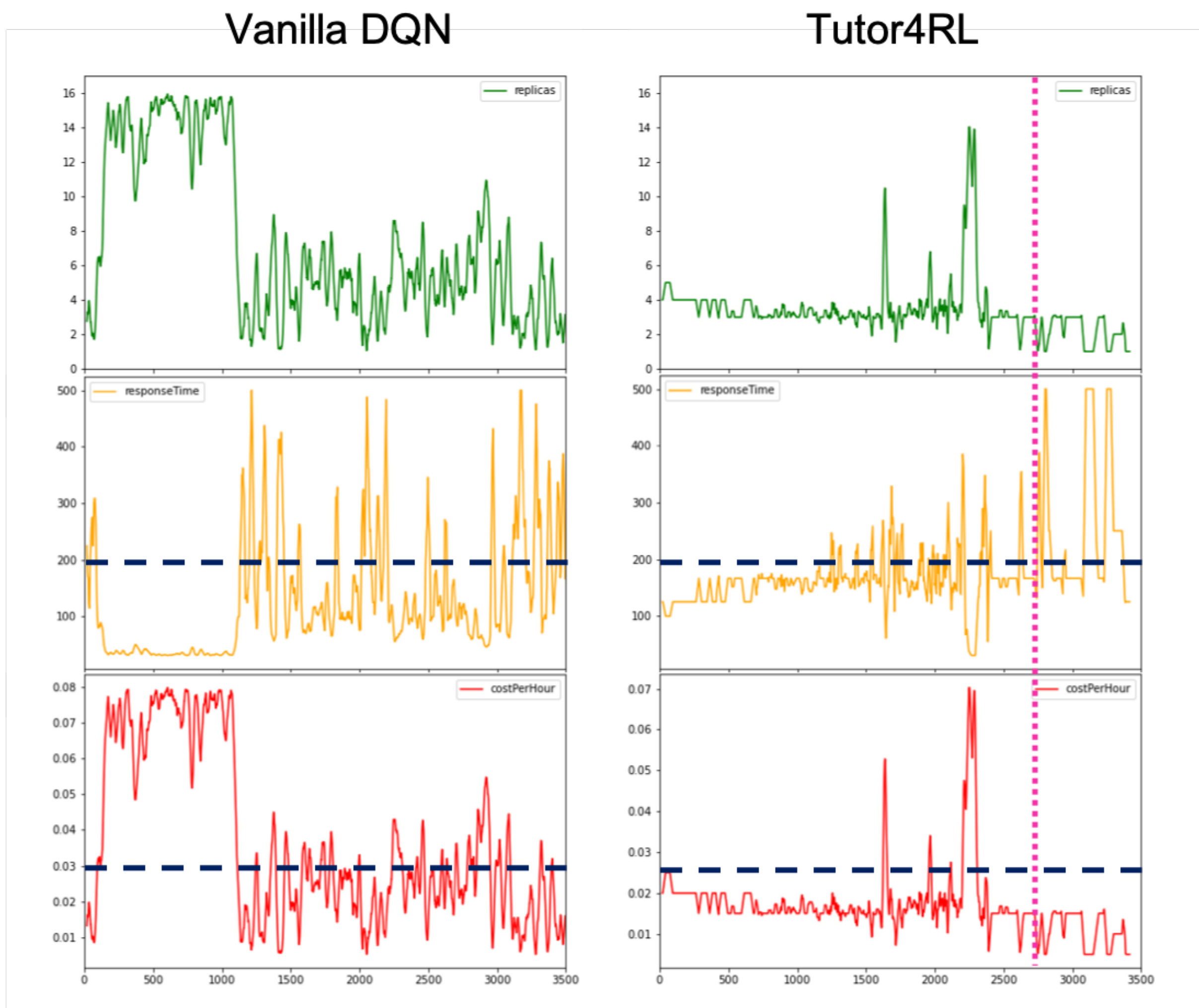


Figure 27: DO performance to manage 2 SLOs: costPerHour < 0.03 and responseTime < 200

On the figure above, Vanilla DQN is shown on the left, while Tutor4RL, with 2 guides and 1 constrain, is shown on the right. The horizontal blue dashed lines show the SLO threshold for the metrics and the pink dotted line show the moment in which guides are not used anymore.

This has been achieved by using 2 guides and 1 constrain for the Tutor, showing the benefits of our approach. The guide (#1 and #2) and constrain (#3) functions are shown in Figure 28. In #1, we avoid unnecessary deployment changes if the reward is positive, i.e. no SLO is being violated. This has been implemented as a guide and will therefore stop being used when Tau equals to 0. #2 is another guide function and checks the response time, if it is less than 100ms (50% of the maximum threshold for the response time SLO), a replica might be removed. Function #3 is a constrain and will be always applied to the agent's action vector. This function avoids the removal of a replica when response time is 80% of the maximum threshold.

guides_constrains.py

```
#1
def avoid_deployment_changes(state, reward, actions):
    if reward > 0:
        actions['None'] += 1
    return actions_dict

#2
def remove_replicas_if_response_time_low(state, reward, actions):
    if state['responseTime'] < 100:
        actions_dict['remove replicas'] += 1
    return actions_dict

#3
def dont_remove_replicas_if_response_time_high(state, reward, actions):
    if state['responseTime'] > 200 * 0.8:
        actions_dict['remove replicas'] = 0
    return actions_dict
```

Figure 28: Guide (#1 and #2) and constrain (#3) functions for DO.

7.6. Next Steps

Beyond BigDataStack, we plan to continue the improvement of the DO, by integrating it and testing it with FogFlow. This activity is currently under development and it will give us important insights about how generally applicable is Tutor4RL as well as the guide and constrain functions we have developed for BigDataStack.

CC BY NC ND

8. ADS Ranking & Deploy

The role of the ranking and deployment module of BigDataStack is to decide how to deploy the user's application and then operationalize that deployment via a container orchestration platform (e.g. Kubernetes). Ranking and deployment is part of the application deployment back-bone that enables a user to get their application running on a hardware cluster. Prior to ranking and deployment, the user will have defined in a conceptual manner what their application is comprised of and how the different services within that application interact, forming a BigDataStack Playbook. This conceptual definition will have then been extrapolated into multiple deployment options, representing different ways that the application/services can be mapped onto compute resources. Finally, these options will have been benchmarked, providing estimated resource usage and quality of service information for each. Ranking and deployment takes these deployment options and associated benchmarking information as input, identifies the optimal deployment configuration based on needed resources, and also handles subsequent deployment on the cluster or cloud.

As its name suggests, ranking and deployment is split into two distinct components, namely: ADS (Application and Data Services) Ranking and ADS (Application and Data Services) Deployment. ADS Ranking is responsible for taking the different deployment options and associated benchmarking information, and deciding which is the most suitable based on the user requirements and preferences. This has two uses within BigDataStack, namely: to determine what compute resources to request for a user's application when first deploying it; and to re-estimate compute resource needs in cases where a current deployment is predicted to miss one or more Service Level Objectives. ADS Ranking is also sometimes referred to as the Deployment Recommender Service, as it produces a recommended deployment configuration for the user. Meanwhile, ADS Deploy is responsible for taking the selected deployment option and using the configuration information contained within, to operationalize deployment of the user's application on the cluster or cloud infrastructure.

8.1. Changes Since D3.2

Over the last year since the previous WP3 deliverable, there have been significant changes in how user application management within BigDataStack is handled. In particular, the introduction of the Realization Engine as a new suite of services that enable the user to more effectively define, configure and manage their applications altered the previous deployment flow. Previously, upon ingestion of a BigDataStack playbook, deployment was largely automated, with that playbook being fed in a serial manner through Pattern Generation, Benchmarking, ADS-Ranking (the Deployment Recommender Service) and finally to ADS-Deploy. This is no longer the case, instead BigDataStack playbooks now go through a registration process with the Realization Engine, where it is de-constructed into modular components that can be independently managed. The user can then trigger atomic operations or aggregate operation sequences, which represent 'actions' to perform on the user application. As such, resource recommendation via ADS-Ranking and the underlying deployment process via ADS-Deploy were re-built to function within the atomic operations 'RecommendResources' and 'Apply', respectively. These operations can then be included as

part of any operation sequence, providing users significantly more customisation than was previously possible.

The other main addition was the Tier 2 implementation of ADS-Ranking, which transitions from the heuristic deployment scoring mechanisms used in Tier 1 to a new machine learned scoring function based on Learning to Rank. Associated to this, as second deployment ranking dataset was developed to enable evaluation of ADS-Ranking for machine learning based applications. Additional information regarding this can be found later in this section.

8.2. Terminology

As a result of the changes to the underlying application management process via the introduction of the Realization Engine, some of the underlying application modelling has similarly changed. This had down-stream impacts on the input formats used by ADS-Ranking and ADS-Deploy. Hence, it is worth summarizing the updated terminology that we use later and how that maps to the terminology used previously in D3.1/D3.2.

Name	Description	Relation to Previous Deliverables
BigDataStack Playbook	This is the conceptual representation of a user application that is registered with the Realization Engine. It can include one or more of the following: 1) application definition; 2) comprised object definitions; 3) metrics; 4) service level objectives; 5) operation sequences; 6) application states.	This new BigDataStack playbook is more structured and can contain significantly more information than those used in D3.1/D3.2.
BigDataStack Object	A BigDataStack Object is the internal representation of a Kubernetes/ OpenShift object (e.g. a DeploymentConfig or Service) within the Realization Engine. Such objects can either be templates or instances, where templates represent the blueprint for creating the object on the cluster, while an instance represents an actual object post-deployment.	BigDataStack Objects did not exist previously. The closest approximation under the old system is an old format BigDataStack Playbook.
Resource Template	This is a specification of a set of resources that can be associated to a BigDataStack Object, typically covering CPU, Memory and GPU requests and limits.	A Candidate Deployment Pattern (CDP) was the aggregation of a Resource Template and old format BigDataStack Playbook. Now

		Resource Templates are managed independently.
Benchmark Results	This is a set of features describing the run-time performance for a BigDataStack Object and Resource Template pair as outcome from a benchmarking run.	A Dimensioned Deployment (DD) Playbook was the aggregation of a Benchmark Result, a Resource Template and old format BigDataStack Playbook. Benchmark Results are now managed independently.
Workload	A workload describes a data in-load scenario for a user application. This is used during benchmarking to specify how heavily loaded a BigDataStack Object instance is.	This remains unchanged from previous deliverables.

8.3. Requirements

To facilitate the understanding of the design as well as the challenges addressed by this component, the requirements related to this component have been brought from D2.3, updated to reflect the above terminology changes and included into this section. Note that the requirements themselves have not changed (only the wording has been updated) and are included in here simply for the reader's convenience.

This section contains the requirements for both the ADS Ranking and ADS Deployment components, denotated as REQ-ADSR-XX and REQ-ADSD-XX, respectively.

	Id	**Level of detail**	**Type**	**Actor**	**Priority**
	REQ-ADSR-01	System	FUNC	Application Dimensioning Workbench	MAN
Name	Ingest BigDataStack Objects, Resource Templates and Benchmark Results				
Description	The Application Dimensioning Workbench sends a series of deployments (BigDataStack Object and Resource Template pairs) and Benchmark Results to the ADS Ranking component. ADS Ranking needs to collect these for subsequent scoring/ranking based on the user requirements and preferences.				
Additional Information	Communication is now handled via REST API, where the process is mediated via the Realization Engine.				

Table 8 - Requirement (1) for ADS Ranking

	Id	**Level of detail**	**Type**	**Actor**	**Priority**

	REQ-ADSR-02	System	FUNC	Dynamic Orchestrator, Application Dimensioning Workbench	MAN
Name	Deployment Suitability Feature Extraction				
Description	Once a series of deployments (BigDataStack Object and Resource Template pairs) and associated Benchmark Results has been received, the next step is to determine how each is predicted to perform based on the benchmarking information. In effect, this involves defining a series of functions that relate individual or groups of user requirements to the predicted performances produced by benchmarking. The output of this step is a vector representation for each candidate deployment, representing how that deployment is predicted to perform under different user requirements.				
Additional Information	Features produced here are dependent on the capabilities of the benchmarking system and the amount of information the user provides in terms of requirements and preferences.				

Table 9 - Requirement (2) for ADS Ranking

	Id	**Level of detail**	**Type**	**Actor**	**Priority**
	REQ-ADSR-03	System	FUNC	Dynamic Orchestrator, Application Dimensioning Workbench	MAN
Name	Deployment Scoring (Heuristic)				
Description	Given a vector representation for a deployment (BigDataStack Object and Resource Template pair), we next need to map this vector into a single score, representing how suitable that deployment will be overall (such that we can compare different deployments). This involves combining the different elements within the vector (that each represent some aspect of pattern suitability, such as cost, or predicted compute wastage). The first version of this component will use a hand-tuned linear combination.				
Additional Information	N/A				

Table 10 - Requirement (3) for ADS Ranking

	Id	**Level of detail**	**Type**	**Actor**	**Priority**
	REQ-ADSR-04	System	FUNC	Dynamic Orchestrator, Application Dimensioning Workbench	DES

CC BY NC ND

Name	Deployment Scoring (Supervised)				
Description	Given a vector representation for a deployment (BigDataStack Object and Resource Template pair), we next need to map this vector into a single score, representing how suitable that deployment will be overall (such that we can compare different deployments). This involves combining the different elements within the vector (that each represent some aspect of pattern suitability, such as cost, or predicted compute wastage). The second version of this component will learn how to combine the elements based on logging information from past deployments. Models may be non-linear in nature.				
Additional Information	Depends on REQ-ADSR-06.				

Table 11 - Requirement (4) for ADS Ranking

	Id	**Level of detail**	**Type**	**Actor**	**Priority**
	REQ-ADSR-05	System	FUNC	Dynamic Orchestrator, Application Dimensioning Workbench	MAN
Name	Deployment Selection				
Description	Once all candidate deployment patterns have been scored, the final step is to select one of those deployments to pass to ADS Deploy. In many cases this will simply involve selecting the highest scoring pattern. However, the user may have the option to select an alternative configuration at this stage.				
Additional Information	N/A				

Table 12 - Requirement (5) for ADS Ranking

	Id	**Level of detail**	**Type**	**Actor**	**Priority**
	REQ-ADSR-06	System	FUNC	Dynamic Orchestrator, Application Dimensioning Workbench	DES
Name	Supervised Model Training				
Description	To support REQ-ADSR-04, a supervised scoring model is needed. To react to changes in the deployment environment over time, this model needs to be frequently updated based on new information from current deployments. This model needs to be trained based on logging data being collected by the Triple Monitoring Framework.				
Additional	Requires logging information produced by the Triple Monitoring				

cc BY NC ND

Information	Framework and stored in the Realization Engine.

Table 13 - Requirement (6) for ADS Ranking

	Id	Level of detail	Type	Actor	Priority
	REQ-ADSR-07	System	FUNC	Dynamic Orchestrator	MAN
Name	Deployment Re-Scoring				
Description	It is envisaged that in (rare) scenarios, an ongoing application deployment will fail to meet the user's quality of service requirements. For instance, this might occur due to assumptions on data input volumes being violated. In this case, we may not be able to solve this issue without fully redeploying the user application with different resources. To support such re-deployment activities, ADS Ranking supports a re-scoring function, where a previous set of candidate deployments for a user's application can be re-scored based on updated preferences provided by the Dynamic Orchestrator, as well as data about how the previous deployment performed (and failed).				
Additional Information	N/A				

Table 14 - Requirement (7) for ADS Ranking

	Id	Level of detail	Type	Actor	Priority
	REQ-ADSR-08	System	FUNC	ADS Ranking	DES
Name	Deployment Dataset Generation				
Description	To support REQ-ADSR-06 and hence REQ-ADSR-04, significant volumes of logging data from past deployments are needed to enable effective model creation. To this end, a framework and methodology for generating this data is needed. Such logging data can be produced through either benchmarking, live deployment of the end-user applications and via simulated application deployment.				
Additional Information	Data storage for this task is handled by the Triple Monitoring Framework and Realization Engine. Data generation is supported by deployments by the application dimensioning workbench and other dedicated deployment applications.				

Table 15 - Requirement (8) for ADS Ranking

	Id	Level of detail	Type	Actor	Priority
	REQ-ADSD-01	Stakeholder	FUNC	ADS Deploy	MAN
Name	Performance Measurability				
Description	Each environment should be measurable according to a set of characteristics, that is, Key Performance Indicators (KPIs).				

cc BY NC ND

Additional Information	The KPIs considered must include: - vCPUs - Memory

Table 16 - Requirement (1) for ADS Deploy

	Id	Level of detail	Type	Actor	Priority
	REQ-ADSD-02	Stakeholder	FUNC	Application Engineer, Data Engineer	MAN
Name	Standardised Object Loading				
Description	The description of the environments and deployments (i.e., BigDataStack Objects) will follow a specification language that is intuitive and as close (similar) as possible to well-known and widely-used schemas to describe software application deployments in cloud infrastructures, such as Docker Compose or Kubernetes Deployment.				
Additional Information	N/A				

Table 17 - Requirement (2) for ADS Deploy

	Id	Level of detail	Type	Actor	Priority
	REQ-ADSD-03	System	FUNC	Application Engineer, Data Engineer	MAN
Name	Standard deployment information				
Description	When communicating with other components, as described in Section 8.2, these components will use the standard defined in REQ-RD-02.				
Additional Information	N/A				

Table 18 - Requirement (3) for ADS Deploy

	Id	Level of detail	Type	Actor	Priority
	REQ-ADSD-04	System	FUNC	ADS Ranking	MAN
Name	Application Scoring System				
Description	The ranking system evaluates each environment's deployment, which keeps track of the most suitable configuration for each application. When trying a deployment configuration for a new application, this ranking will be used to select the most suitable one.				
Additional Information	The evaluation needs to be performed following the measurements defined in REQ-RD-01.				

Table 19 - Requirement (4) for ADS Deploy

	Id	Level of detail	Type	Actor	Priority

	REQ-ADSD-05	Software	FUNC	Cluster Management	MAN
Name	Compatibility with Kubernetes				
Description	Since the technology used to run and orchestrate the applications is based on Kubernetes (OKD[11]). Thus, the ADS-Deployment component is required to be compatible with Kubernetes.				
Additional Information	The ADS-Deploy component should translate from the playbook standard defined in REQ-RD-01 into Kubernetes primitives.				

Table 20 - Requirement (5) for ADS Deploy

	Id	**Level of detail**	**Type**	**Actor**	**Priority**
	REQ-ADSD-06	System	PERF	ADS Ranking	MAN
Name	Synchronous communication				
Description	The communication with and within ADS Ranking and ADS Deploy must be done through an API REST.				
Additional Information	N/A				

Table 21 - Requirement (6) for ADS Deploy

8.4. Design Specifications

The design for ADS-Ranking and ADS-Deploy was originally specified in Section 7 of D3.1 and later updated in Section 7 of D3.2. The primary changes made in Y3 are: 1) an updated process flow to account for the introduction of the Realization Engine; 2) the introduction of the RecommendResources and Apply operations with associated updated communication interfaces for ADS-Ranking and ADS-Deploy that enables interoperability with the Realization Engine; 3) the integration with the new Realization UI (rather than the more general BigDataStack Visualisation service as was used previously); and 4) the extension of ADS-Ranking to Tier 2, enabling support for machine learned evaluation of candidate deployments. The following sub-sections describe these changes in more detail, with the exception of the changes of ADS-Ranking that is described in Section 8.5.

8.4.1. Updated Architecture

An updated architecture diagram is provided in Figure 29 below (contrasting the original architecture provided in Figure 12 of D3.1). As can be seen from Figure 29, the process flow for the usage of ADS Ranking and ADS Deploy is now as follows: First, the user (application engineer) can interact with the Realization UI to access the functionality of ADS Ranking and ADS Deploy (and indeed all of the other services connected to the Realization Engine). For illustration, let us assume that the user had an application registered that contains a BigDataStack Object 'InsuranceClassifierService'. Furthermore, let us assume that a BigDataStack Operation Sequence (action) has been registered named 'deploy' that contains

[11] OKD - https://www.okd.io/

CC BY NC ND

the three in-built operations: 'RecommendResources'; 'Instantiate'; and 'Apply'. From the Realization UI, the user can trigger the 'deploy' action. This sends a request to the Realization Engine via its API (in this case the Executions endpoint), which in turn will start processing the specified operations within the sequence in order. When the RecommendResources operation starts it will collect the needed information about the target object template (InsuranceClassifierService) and sends that information to ADS Ranking, which will in turn produce new recommended resources for that object and store that resource definition within the State Database. Once the recommended resource definition is ready, the RecommendResources operation concludes, and passes control to the next operation in the sequence, i.e. Instantiate. The Instantiate operation is responsible for generating a new BigDataStack Object Definition instance based on a BigDataStack Object Definition template. In this case, it will produce a new InsuranceClassifierService instance from the associated template, and then store that instance within the State Database. Finally, once instantiation is complete, control is passed to the Apply Operation. The Apply operation takes the InsuranceClassifierService instance along with the recommended resources definition previously produced by ADS-Ranking, and passes them to ADS Deploy, which operationalizes the creation of the associated object (and hence service) on the cloud or cluster infrastructure via the OpenShift API. Once running, adaptations to the deployment can be triggered via operation sequences in the same manner either manually or programmatically. For instance, the Dynamic Orchestrator may trigger an alteration action that involves RecommendResources as one of the operations in the sequence.

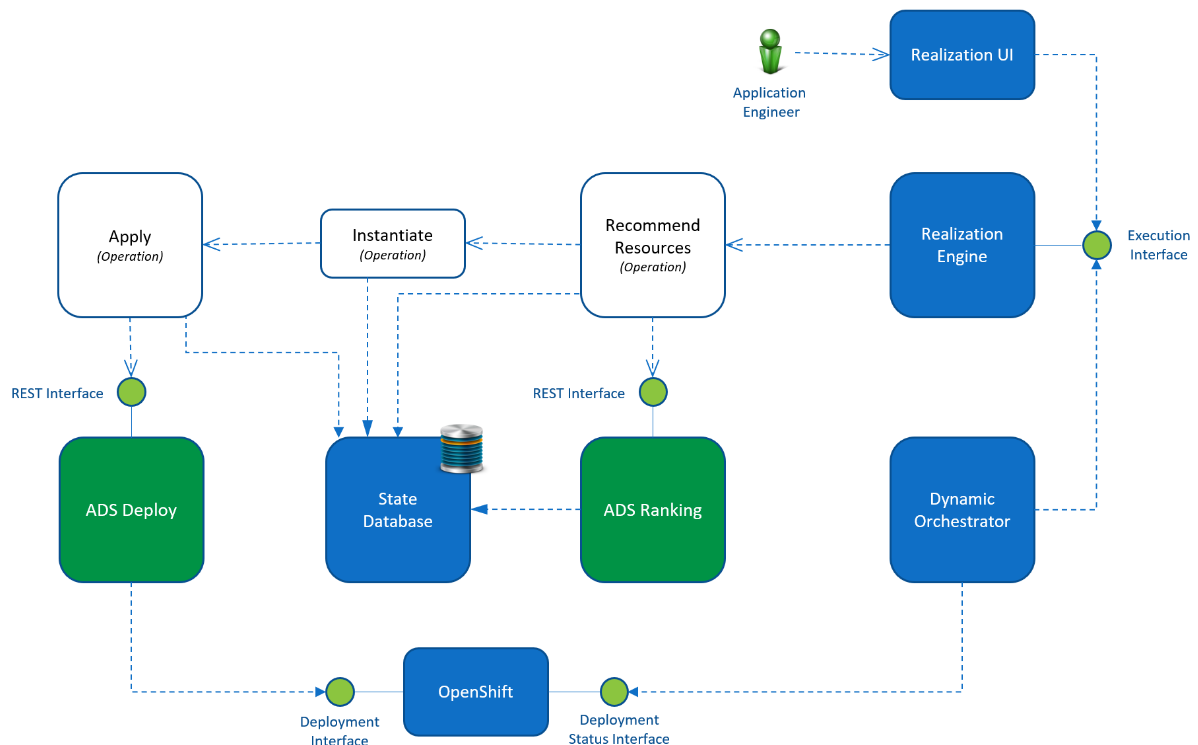


Figure 29: ADS-Ranking and ADS-Deploy Processing Architecture

bigdatastack.eu

8.4.2. Recommend Resources Operation

Previously, as described in Section 7 of D3.1, ADS-Ranking used a publisher-subscriber mechanism to ingest the user application details it needs to generate a set of recommended resources. This design decision was reasonable at the time, as the entire deployment flow was encapsulated within a stream processing pipeline, where a playbook was provided as input at one end and a deployment was the output at the other end. With the introduction of the Realization Engine, this process was converted to be user (or at least business-logic) triggered, and the various sub-components of the process were isolated such that they could be triggered independently as atomic operations.

To enable this, ADS-Ranking was altered to support ingestion of BigDataStack Objects, Resource Templates and Benchmarking Results via REST API, rather than through a subscription. In particular, all communication with ADS Ranking is now abstracted behind a pre-defined operation that is built into the Realization Engine, namely: 'RecommendResources'. When this operation is triggered for a particular BigDataStack Object, it in turn performs the following steps:

1. Retrieves the BigDataStack Object Template from the State Database
2. Retrieves the available Resource Templates for the current cluster from the State Database
3. For each Resource Template, Benchmark Results are requested from the Benchmarking component (Flexibench)
4. The resultant <BigDataStack Object, Resource Template, Benchmark Result> tuples are then sent via HTTP POST request in JSON format to an endpoint exposed by the ADS-Ranking service.
5. The operation waits until updated resource definitions are detected in the State Database.

Note that the request at Step 4 only responds with whether the request was accepted (as validated based on correct formatting), it does not directly respond with the results. Internally, ADS-Ranking is still a stream processing system (see D3.1), and hence output is asynchronous to the input. Hence, Step 5 exists to periodically check whether the updated resources are yet available.

8.4.3. Apply Operation

'Apply' is a new operation that is built into the Realization Engine to facilitate the deployment of the user application via ADS-Deploy. When Apply is triggered, it internally performs the following steps:

1. Retrieves the BigDataStack Object instance to deploy from the State Database
2. If the BigDataStack Object instance is either of type DeploymentConfig, Job or Pod, then the State Database is checked to see whether one or more Resource Templates are associated to that Object. If so, the Resource Templates will be merged into the object.

3. The resultant object is sent via REST API to ADS-Deploy.

Along with the new Apply operation, some minor changes were made to ADS-Deploy itself to support the new object modelling introduced with the Realization Engine. In particular, previously ADS-Deploy expected a Playbook as input. Instead, it now takes a BigDataStack Object instance definition, which fulfils the same purpose. It is however notable that as BigDataStack Objects are more general and support a wider range of Kubernetes/OpenShift objects, ADS-Deploy also benefits from this increased support natively (providing better support for REQ-ADSD-05 than previously).

8.4.4. Connection with the Realization UI

In D3.2 (and demonstrated at the M18 review) the user interacted with ADS Ranking, via the top-level BigDataStack user interface, which is known as the BigDataStack Visualisation Service. However, with the introduction of the Realization Engine it became clear that there was a need for a separate user interface with an increased feature-set, which would enable the user to do more than simply deploy their application. This resulted in the development of the separate Realization UI, that enables the user to also manage their applications pre- and post-deployment.

ADS Ranking is one of the in-built services within the Realization Engine, and as such the Realization Engine UI integrates some additional features for it. In particular, if the user triggers an operation sequence (action) that involves the deployment of a container (e.g. one that instantiates a BigDataStack Object of type 'DeploymentConfig' or 'Job') then the associated BigDataStack Object(s) will be checked to see if they include a complete Resource Template. If so, the operation sequence will be started as normal. If not, the operation sequence is checked to see whether it contains a RecommendResources operation targeting the object. If not, then the Realization Engine UI will prompt the user to either manually provide the missing information or trigger the RecommendResources operation to generate the missing information automatically. In this way, the Realization Engine UI leverages direct integration with ADS-Ranking to prompt the user to follow good practice when deploying cloud/cluster applications, by always specifying the resources they think they need prior to deployment.

8.5. ADS Ranking Tier 2 (Machine Learned Ranking)

In this section we describe the new machine learned model used for evaluating the suitability of different deployment options in ADS Ranking (Tier 2). The section is structured as follows. In Section 8.5.1 we provide a brief summary of related works in the field of machine learning. Section 8.5.2 describes how we formulate the problem of ranking deployment options for various service level objectives as a machine learning task. Finally, in Section 8.5.3 we discuss how to tackle the issue of multiple competing service level objectives when ranking.

8.5.1. Related Work

Previously in D3.2 we proposed to tackle the problem of determining what resources to allocate to a particular application component (a container to be more precise) as a learning

CC BY NC ND

to rank problem. Generally, ranking problems can be defined as a derivation of ordering over a list of items, where the goal is to maximize the utility of the entire list [45]. The theory is that in scenarios where multiple options will be shown to the user, it is more effective to optimise for the entire ordered list of items rather than consider each item individually. Learning to rank is the application of supervised learning technologies to such ranking problems, and has been widely used in several domains, most notably for search and natural language processing applications [10].

It is worth noting that machine learned ranking is a fundamentally different problem to either item classification or regression, where the goal is to construct a function for automatic assignment of labels or numerical values to single items, respectively (although you can use regression models for point-wise ranking as discussed later). This is because the goal of learning to rank is to maximise the utility of the entire list, hence it is the ordering of items that matters, not their individual score [10] .

A common method for distinguishing different learning to rank approaches is based on how they define a surrogate loss function over the ranked lists of items during training. In particular, the simplest (and least effective) class of learning to rank approaches use point-wise scoring of items [46]. These approaches are simply the direct application of regression-based supervised learning to the problem of ranking, where the goal is to assign a score to each item individually. These approaches have been shown to be less effective than later methods as they lack the contextual information from the rest of the ranked list. The other two classes of learning to rank algorithms, referred to as pair-wise and rank-wise approaches, both incorporate this contextual information. Pair-wise approaches calculate loss over every pair of items within the ranked list to capture the relative ordering of items [47]. Meanwhile, list-wise approaches calculate loss over the entire ranked list as a whole [48,49].

In general, list-wise approaches have been shown to be the most effective in practice. List-wise approaches can use a range of different metrics as a surrogate for loss, such as Normalized Discounted Cumulative Gain (NDCG) [9], Expected Reciprocal Rank (ERR) [50], and Mean Average Precision (MAP). These metrics score a ranking based on the number (and in cases like NDCG the quality) of relevant/suitable items near the top of the ranking, following the probability ranking principle [45].

In terms of the algorithm used to operationalize the learning process, classical learning to rank techniques can be primarily divided into either linear or tree-based learners. A linear learner will produce a model that linearly combines the feature scores for an item. Meanwhile, a tree-based learner builds a decision tree-like structure, where the branch nodes denote decisions based upon the features and each leaf node represents a final score to return. Over the last couple of years there has been a resurgence of research targeting the learning to rank problem, examining how the emergence of deep neural networks can also be applied to this problem, such as TensorFlow Ranking [51]. However, it was still unclear at this stage whether these are better than classical approaches, and the black-box nature of the underlying model is a problem when explainability is a desirable characteristic. Hence, we decided to focus on classical tree-based learners in this work, as they have been shown to be effective [52].

cc BY NC ND

8.5.2. Modelling Deployment Ranking as a Learning Task

When considering the problem of ranking deployment options for the user, the first question that needs to be asked is 'what are the items to be ranked'? In our scenario, an item corresponds to the deployment of an application component, which is comprised of a BigDataStack Object and Resource Template. When considering this as a ranking task, the goal is for a given BigDataStack Object, to produce a ranked list of Resource Templates, where suitable Resource Templates are ranked above less suitable Resource Templates. In this case, suitability is defined in terms of the service level objectives (slos) attached to the BigDataStack Object.

Having defined the task formulation, the second question that needs to be answered is 'how can items be modelled'? For any machine learning task, the items need to be represented in the form of a fixed length numerical vector. The constraint is each vector must exist within the same conceptual vector space, such that the vectors for two items are comparable. As a general rule of thumb when modelling items for any task, all distinct information about the item itself, contextual information about the ranking environment, as well as predicted indicators of suitability should be encoded within the item vector. Within the context of our ranking task, there are then four categories of information that we can potentially encode within the item vector:

- **BigDataStack Object Features**: These are the representation of the actual application component being deployed. Such features can be extracted from the BigDataStack Object, such as from its description or performance characteristics if available.
- **Benchmarks**: For each <BigDataStack Object, Resource Template> pair, we also assume that we have predictive benchmark results for that application produced by Flexibench component of WP5 or equivalent.
- **Service Level Objective Features**: These represent the service level objectives provided by the user for the BigDataStack Object.
- **Suitability Indicators**: Given a set of Benchmarks and also the Service Level Objectives provided by the user, it is often possible to produce suitability indicators that contrast the Benchmark outcomes with one or more Service Level Objectives, e.g. contrasting predicted latency vs. a latency target.

However, there are two problems with attempting to model an item using all of these categories of information. The first problem is that the benchmarks produced for an item depend on the application type. For instance, for a streaming application, benchmarking might report information about processing throughput for the stream. Meanwhile, for a batch application, the information reported might be completion time or records processed per minute. As such, applications of very different types cannot be directly compared. Practically, there is no means to work around this problem using a single model, hence the solution to this is to train a different model for each broad type of application. We discuss the features included in our test models in Section 8.5.4. The second problem is in regard to the user's service level objectives. Specifically, there are no constraints on the number of service level objectives that a user can define for a given BigDataStack Object. Recall that the item vector must be of constant length and comparable to other item vectors, hence

CC BY NC ND

simply encoding all service level objectives would result in variable-length vectors. To solve this, instead of directly reporting service level objective features, we instead define an intermediate scoring function that calculates a suitability indicator, aggregating data from across all of the service level objectives and the benchmarking results, outputting a single predictive performance number, which we discuss in the next section.

8.5.3. Aggregating Across Service Level Objectives

As discussed above, to tackle the issue with variable numbers of service level objectives, we need to devise a means to represent any combination of such objectives as a fixed length vector. To do so, we define a function that takes in a list of service level objectives along with associated benchmark data, producing a score. The aim is that this score should capture the degree to which the benchmark data indicates that the service level objectives will be met.

From a high-level perspective this function simply calculates the degree to which each service level objective is predicted to be met by the current deployment (a score), and then averages those scores across all objectives to produce a final predicted suitability score:

$$pred_sutability = \frac{1}{|SLOs|} \cdot \sum_{slo \in SLOs} 1 - loss_{[slo.metric]}(slo, B[slo.metric])$$

where *SLOs* is the set of all service level objectives, *B* is the set of benchmark results, *slo.metric* is the associated metric that the *slo* targets and $loss_{[x]}$ is a function that estimates the degree to which *B[slo.metric]* satisfies *slo* or not. Note that this internal scoring function $loss_{[x]}$ changes depending on the particular metric being evaluated. This is needed since we aim to capture the degree to which each slo is met, not simply whether it will be met. Hence, we need different scoring functions that account for the fact that different metrics have different working ranges and meanings. For example, consider a slo targeting throughput more than 100 messages per second. We might receive predicted benchmarking results indicating that throughput will be 110 messages per second, a success. But how do we quantify the degree of that success? This is the role of each $loss_{[x]}$ function.

ADS-Ranking currently supports scoring functions for the following metrics (and hence slo types):

- CPU Utilisation
- Memory Utilization
- Latency
- Throughput
- Completion Time
- Cost Per Hour
- Total Cost

For each one of these metrics we define a function that takes in a delta between the predicted value (produced by benchmarking)

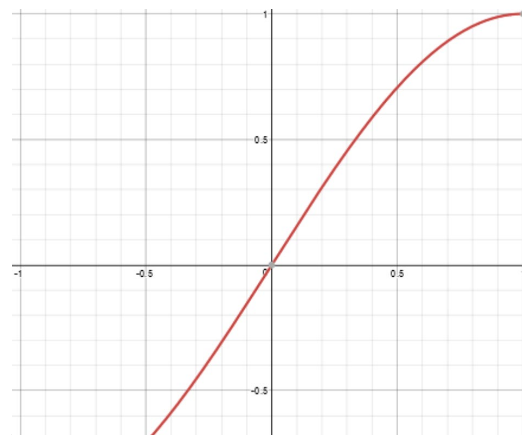


Figure 30: Cost Per Hour Delta Scoring Function

CC BY NC ND

and the target value (specified by the slo) and produces a value between 0 and 1. As this is a loss function, lower values are better. For example, we visualise the Cost Per Hour function in Figure 30, where the x axis is the cost delta in US dollars and the y-axis is the output score. As we can see, this function will return a positive score if the cost per hour is higher than the target (a failure), and a negative value if it is lower than the target (a success), where a maximum positive or negative value is achieved at 1 US dollar above or below the target respectively. Note that as we desire a loss value between 0 and 1 as output, as a final step we scale the range by adding 1 and dividing by 2 to produce the final score as a loss for this particular function.

By defining $loss_{[x]}$ functions covering the common types of service level objectives that a user might care about, we can support automated evaluation for a wide range of user applications. Internally, these functions are implemented as classes that extend a common interface, keyed by metric name. As such new functions can be added over time to increase support for new application types.

Furthermore, it is worth highlighting that the aggregate predicted suitability of a deployment can be calculated over different sub-sets of slos. In particular, each slo is labelled as either a requirement (something that the application must meet) vs. a preference (something that is desirable to meet). Hence, in practice when generating features for learning, we calculate aggregate predicted suitability for: all slos, requirements only; and preferences only. In this way, the learner is provided evidence with which it can distinguish between the different slo types.

8.5.4. Models and Feature Sets

For the purposes of supporting the Pilots within BigDataStack we develop two different models with associated feature sets, representing two common application types: 1) stream processing applications; and 2) batch model training. The pipeline for training each of these two models is the same. What distinguishes these models is the features that they work from, as both the information provided from benchmarking and the types of service level objectives that the user cares about in each case differ. The features used for each of these two models are listed below:

Feature Name	Type	Summary	Stream Processing Model	Batch Training Model
Object Description Embedding	BigDataStack Object Feature	Word embedding derived from the object description field	Yes	Yes
CPU Average	Benchmark	Average predicted CPU utilization over the container lifetime (millicores)	Yes	Yes
CPU Peak	Benchmark	Peak predicted CPU utilization over the container lifetime (millicores)	Yes	Yes

CC BY NC ND

Memory Average	Benchmark	Average predicted Memory utilization over the container lifetime (Megabytes)	Yes	Yes
Memory Peak	Benchmark	Peak predicted Memory utilization over the container lifetime (Megabytes)	Yes	Yes
Average Latency	Benchmark	Average end-point response latency (milliseconds)	Yes	No
Peak Latency	Benchmark	Maximum end-point response latency (milliseconds)	Yes	No
Average Throughput	Benchmark	Average items processed per second	Yes	No
Peak Throughput	Benchmark	Peak items processed per second	Yes	No
Completion Time	Benchmark	Total time needed to complete training (seconds)	No	Yes
Precision	Benchmark	Resultant Model Precision	No	Yes
Recall	Benchmark	Resultant Model Recall	No	Yes
NDCG	Benchmark	Resultant Model Normalised Discounted Cumulative Gain	No	Yes
Predicted Suitability All	Suitability Indicator	Output of the predicted suitability scoring function discussed above for all slos	Yes	Yes
Predicted Suitability Requirements	Suitability Indicator	Output of the predicted suitability scoring function discussed above for slo requirements	Yes	Yes
Predicted Suitability Preferences	Suitability Indicator	Output of the predicted suitability scoring function discussed above for slo preferences	Yes	Yes
Predicted Proportion of SLOs Passed	Suitability Indicator	The proportion of all slos that are predicted to be met	Yes	Yes
Predicted Proportion of Requirements Passed	Suitability Indicator	The proportion of requirement slos that are predicted to be met	Yes	Yes
Predicted Proportion of Preferences Passed	Suitability Indicator	The proportion of preference slos that are predicted to be met	Yes	Yes

cc BY NC ND

8.6. Experimentation Outcomes

In this section we evaluate the performance of ADS Ranking at identifying effective and efficient deployment configurations. As its name suggests, ADS Ranking is a ranking service at its core, i.e. it ranks a set of items provided to it, which are Resource Templates for a BigDataStack Object in our case. Some of those Resource Templates will be more suitable than others. By suitability, we refer to whether the user's requirements and preferences will be met or exceeded, if we use that Resource Template to deploy the user's application. Hence, we can measure how effective ADS Ranking is for an application by evaluating to what extent the top-ranked Resource Templates are suitable. By evaluating the effectiveness of ADS Ranking at deploying different types of application, we can determine the overall effectiveness of ADS Ranking as a whole. In this section we describe the experimental framework and setting we use to perform an evaluation of ADS Ranking in terms of datasets, methodology, metrics and baselines. We then report the performance of ADS Ranking Tier 1 (Heuristic-based) and Tier 2 (Machine Learned), along with two baselines under these datasets and metrics.

8.6.1. Dataset Structure

As discussed in D3.1 Section 8.5, the idea of producing an automatic system to estimate what resources are needed to deploy a user application is novel. Hence, there are not readily available standard datasets that we can leverage to evaluate ADS Ranking. Instead, for our evaluation we generate new datasets. In effect, a dataset for this task can be considered to be comprised of six main parts:

- **BigDataStack Objects**: The definition of application components that we are going to deploy onto the cluster infrastructure. Each BigDataStack Object describes a service within a user's application and is typically either of type DeploymentConfig, Job or Pod.
- **Service Level Objectives**: These are the quality of service factors that the user cares about in terms of hard requirements and softer preferences for a particular BigDataStack Object. These are needed as input to the evaluation function to estimate suitability of a deployment.
- **Workload**: The workload for an application represents the amount of work that the application needs to do. For a real-time streaming application, this might represent the stream of records or requests that need to be processed. Meanwhile for batch operations, this would be the statistics of the dataset or database that needs to be processed or queried.
- **Resource Templates**: For a BigDataStack Object that describes a single service, we also need a series of resource templates that describe the different ways that we might deploy that service on the cluster infrastructure in terms of resources (CPU, GPU, memory, per service).
- **Benchmark Performances**: As part of the ranking process, ADS Ranking utilizes predicted performance estimates produced by the Benchmarking (Flexibench) component of the Application Dimensioning Workbench. In effect, for each BigDataStack Object and Resource Template pair, Benchmarking provides a series of

CC BY NC ND

indicators (features) about how well the service is expected to perform if deployed using the resources described within the specified Resource Template.

- **Ground-truth Performances**: To evaluate to what extent each Resource Template is, in fact suitable for a BigDataStack Object and set of Service Level Objectives, we need to have ground truth information about how the service actually performs on the cluster infrastructure when deployed. Note that this is different to what the Benchmark Performances provide, as those are only (predictive) estimates and are subject to error.

To evaluate ADS Ranking, we report performance using two datasets, each representing a different type of application (streaming vs. batch processing) as summarized in the following two sections.

8.6.2. Dataset 1: Real-time Data Server (Streaming)

The first dataset that we develop to evaluate the quality of ADS Ranking targets the scenario of a real-time service that responds to user traffic (e.g. an insurance recommendation service). In this type of scenario, we have a stream of request traffic being sent to the service and we care about the amount of time it takes for the user to receive a response (referred to as response time or latency), along with the cost of running the service. This is a very common scenario with services that drive user-facing applications.

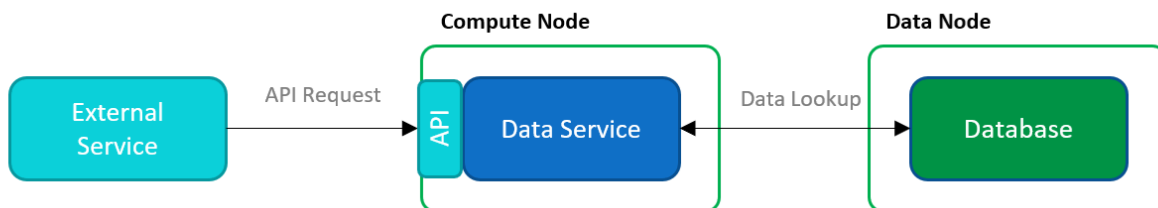


Figure 31: Realtime Data Server Architecture

Figure 31 illustrates the overall architecture of the real-time data service that this dataset models. As can be seen from Figure 31, within this type of system, there is an external service (e.g. a user's web browser) that makes an API request to the main data service. This in turn performs a data lookup into an external database located on another machine. Once the data has been retrieved, some local processing takes place, before a response is generated for the user. Within this type of system, there are a range of properties that can influence the response time that the user experiences, such as latency for the request to the database, the available bandwidth for data transfer between the service and database, as well as the complexity of response generation and compute capacity available on the data service itself. To create this dataset, we implemented a simulation framework that allows us to produce deployable variants of this system for testing, with different properties. For example, one variant might require more cpu cycles to produce each response, while another might involve moving a large record from the database.

Table 22 - Realtime Data Server Deployment Ranking Dataset Statistics

Realtime Data Server Dataset	BigDataStack Objects (Service Variants)	24
	Service Level Objectives (Unique Scenarios)	4 (x 3)
	Workloads	1
	Resource Templates	35
	Total Deployments	2,520

In particular, we created 24 variants of the service. Each of these variants have different processing properties, such as start-up time, per-record processing time, memory usage, maximum throughput and more. We then defined three quality of service levels, which we refer to as medium, high and extreme, where each quality of service level specifies the response time bounds and cost for the application (service level objectives, or slos) that are acceptable for different classes of user, as follows:

- Medium QoS:
    - Requirements: Response Time less than 200ms, Cost less than $1.9/hour
    - Preferences: Response Time less than 100ms, Cost less than $0.7/hour
- High QoS:
    - Requirements: Response Time less than 150ms, Cost less than $1.9/hour
    - Preferences: Response Time less than 70ms, Cost less than $0.7/hour
- Extreme QoS:
    - Requirement: Response Time less than 70ms, Cost less than $1.9/hour
    - Preference: Response Time less than 50ms, Cost less than $0.7/hour

Next, we generated one BigDataStack Object for each service variant and quality of service pair, resulting in 72 combinations (24 services x 3 QoS levels). For this dataset, we define a single stream processing workload (to limit the number of tests needing run), where the average input rate is 300 requests per second, with a peak input rate of 500 requests per second. We refer to the combination of a <BigDataStack Object, QoS scenario, Workload> tuple as an experimental scenario.

For each of the generated BigDataStack Objects, we then submitted them to the ADW Pattern Generation component deployed on our local testbed, which in turn produced Resource Templates for each. Based on the underlying available hardware, each BigDataStack Object has 35 possible Resource Templates to consider, hence 35 possible deployments are generated per experimental scenario, creating a total of 2,520 deployments (72 scenarios x 35 Resource Templates). At this point, we deployed each of the 2,520 combinations in turn, collecting resource usage and quality of service information. More precisely, we tracked average and peak CPU and memory usage, along with average and peak response time. In this way, we collected our ground truth performances. There was no competing for resources during these tests and so performances should be comparable between scenarios.

CC BY NC ND

Finally, to generate the benchmark performances that are used as features within ADS-Ranking, we use a local Benchmarking Simulation service that we developed, which simply takes the true ground truth performances and generates benchmark performances from them, with a randomised degree of performance error (+/- 20%) added to represent imperfect benchmarking. This allows us to evaluate ADS Ranking while avoiding systematic biases potentially introduced by Flexibench.

8.6.3. Dataset 2: Training a Deep Learning Model (Batch Processing)

The second dataset that we develop represents a second common type of job that is run on cluster infrastructure, i.e. a job that trains a machine learned model and saves the result to a datastore. For this type of job, the user typically cares about two main quality of service indicators (from the deployment perspective), namely: the time to complete the job; and the cost of training the model. This is because it is common for users to either have a fixed budget for preparing their models, and/or time constraints for completion (e.g. the new model must be available by business open on Monday morning). Cost in particular for training models can be significant with the introduction of new deep learning models that require expensive GPU infrastructure to run.

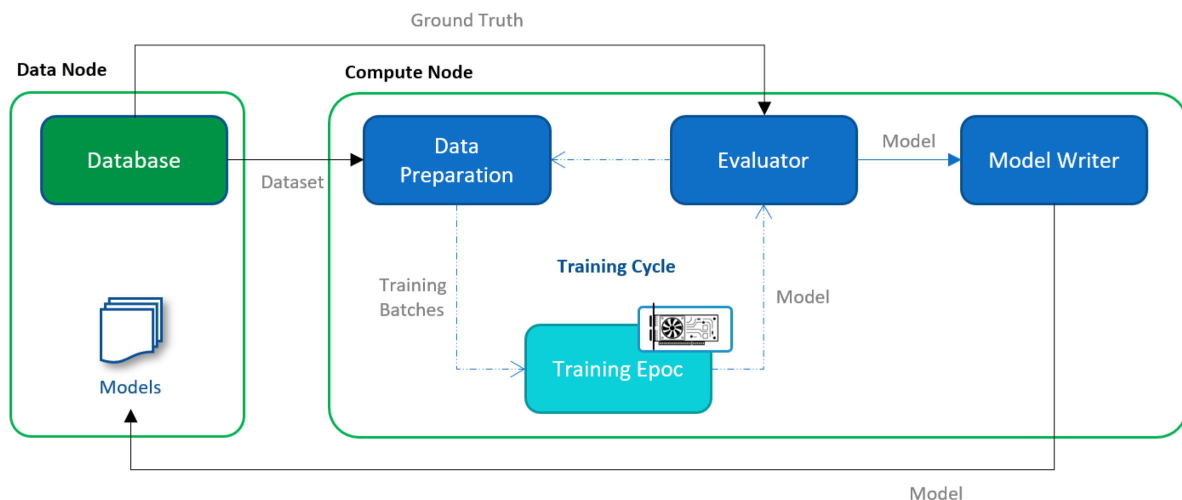


Figure 32: Deep Learning Architecture

Figure 32 shows the architecture of a standard deep learning job that this dataset models. As we can see, initially the dataset being used for training will be transferred to the compute node that will perform the learning. Next, a data preparation stage will be performed, which may involve feature generation and/or data sampling. Once the data is ready, that data will be loaded sequentially in batches into the learning process, where one pass of the data is known as a training epoc. At the end of a training epoc, the current model effectiveness is validated against a ground truth and if an exit condition is not yet reached, the next training epoc will start. Once an exit condition is met, the final model is saved to a datastore. The key factors that can affect cost and completion time for this type of model are: time to transfer the dataset from the database, the compute capacity available for data preparation, the compute available for learning (which may be CPU or GPU bound),

compute available for validation (that can be very expensive for some scenarios like product recommendation); and time for writing the final model.

Table 23 - Statistics for the Deep Learning Deployment Ranking Dataset

Deep Learning Dataset	BigDataStack Objects (Service Variants)	9
	Service Level Objectives (Unique Scenarios)	8 (x 2)
	Workloads	4
	Resource Templates	35
	Total Deployments	2,520

As for dataset 1, we construct a simulator framework to generate variants of this type of job with different properties. In this case, the framework is based on the BetaRecsys framework that we also developed and is described in more detail in D6.2. In particular, we generate 9 variants of this job, where the primary variables were the properties of the model type being trained (e.g. Triple2Vec [53] vs. VBCAR [54]) and the volume of training data used. We then defined two quality of service levels to evaluate the job under, which we refer to as `slow and cheap' and `expensive but fast'. In this case, each quality of service level specifies goals for completion time and cost, either as hard requirements or softer preferences:

Slow and Cheap:

- Requirements:
    - Completion Time less than 2.8 hours
    - Total Cost less than $5
- Preferences:
    - Completion Time less than [1.9, 1.4, 1] hours
    - Total Cost less than [$4, $3, $2]

Expensive but Fast:

- Requirements:
    - Completion Time less than 1.4 hours
    - Total Cost less than $10
- Preferences:
    - Completion Time less than [1.11, 1, 0.83] hours
    - Total Cost less than [$8, $6, $4]

Note that unlike for dataset 1, we define multiple of the same type of service level objective (i.e. completion time or cost) for the preferences here. This is to enable distinctions to be drawn between multiple deployments that all meet the higher-level preferences. To represent varying datasets that the machine learned models might be trained upon, we

cc BY NC ND

define four workloads, where we vary the number of training samples used per training epoc and the total number of epocs to use (the training exit condition):

- Workload 1: 1,000,000 samples per epoc, 120 epocs
- Workload 2: 500,000 samples per epoc, 120 epocs
- Workload 3: 1,000,000 samples per epoc, 50 epocs
- Workload 4: 500,000 samples per epoc, 50 epocs

As for dataset 1, we refer to the combination of a <BigDataStack Object, QoS scenario, Workload> tuple as an experimental scenario. For each variant, we pass the associated BigDataStack Object to the Pattern Generation component deployed on our testbed, which produces a similar set of 35 Resource Templates as for dataset 1, with the exception that each also includes a single RTX Titan graphics card, as all of the variants tested here build a deep learned model that requires a GPU. The combination of 9 BigDataStack Objects, 2 QoS scenarios, 4 Workloads and 35 Resource templates results in 2,520 deployments. We subsequently deployed these on our testbed and recorded the completion time, along with the average and peak cpu and memory usage, forming our ground truth. As with dataset 1, we generate benchmark data from this ground truth by adding a randomised degree of performance error (+/- 20%) to represent imperfect benchmarking.

8.6.4. Metrics

For each of the BigDataStack Objects (representing an application component to deploy), ADS Ranking will output a ranking of the associated Resource Templates along with scores for each. However, to determine how effective each of these rankings are, we need a means to determine the true suitability of each deployment within the rankings. During dataset creation described above, we have two pieces of information to aid in this task. First, we have the quality of service requirements and preferences set by the user. Second, our ground truth performances tell us how well each service performed when given the resources specified within each Resource Template. Hence, we need a mapping function that takes these two pieces of information and produces a suitability score, where a higher score indicates that the user's requirements and preferences were better met. Hence, we use a simple scoring function that produces a suitability score between 0 and 3, where 0 indicates that the deployment (object, workload, QoS scenario and resource template) was unsuitable and 3 indicates that all requirements and preferences were met. Scoring is performed as follows:

- If either response time or cost exceeds the user requirement, or the deployment failed (i.e. the container crashed due to a lack of resources) the deployment receives a score of 0.
- If the user requirements are met, but none of the user preferences are met, the deployment receives a score of 1.
- If the user requirements are met, and any (but not all) of the user preferences are met, the deployment receives a score of 2.

- If all requirements and preferences are met, then the deployment receives a score of 3.

We use this function to produce a ground truth suitability score/label for each deployment.

Once the deployments have been scored, we need to use these scores to evaluate the performance of ADS Ranking as a whole. To do so, we use standard ranking metrics from the information retrieval literature. In particular, we report*:*

- Success@1: This simply evaluates whether the top-ranked deployment met at least the requirements specified by the user over all BigDataStack Objects tested.
- *Precision@5*: This evaluates whether the top ranked deployments were suitable (had a score equal to or greater than 1) over all BigDataStack Objects tested.
- *Mean Average Precision (MAP)*: Average precision (at a particular rank) is the proportion of suitable (has a score equal to or greater than 1) deployments down to that rank. MAP is average precision calculated at the maximum rank (35 in this case) over the BigDataStack Objects tested. [13]
- NDCG: Discounted Cumulative Gain (DCG) is a measure of the usefulness, or gain, of an item based on its position in a ranking. Total gain is accumulated starting from the top of the result list (ranking) and moving downwards to a set rank (the number of deployments ranked in our case, i.e. 35). Gain of each result is discounted at lower ranks and can incorporate different (suitability) grades. Hence, unlike the above two metrics, this metric considers whether the preferences were met in addition to the requirements. NDCG is DCG normalized across (in our case) different application deployments to account for some deployments being easier to find suitable patterns for than others. [9]

8.6.5. Baselines

Using the above dataset and metrics, we can score ADS Ranking in terms of its effectiveness. However, such a score in isolation can be misleading, as it does not provide us information about how difficult the task is. Hence, we also need reference baselines to compare against, providing us context. As this is a new task, there are no standard baselines. Hence, we propose two new baselines here, representing simple strategies that a human might employ when selecting a Resource Template:

- **RankByCost**: This baseline simply ranks each deployment by its cost on the cluster hardware, where the cheapest deployment is ranked first. In particular, cost is calculated as the sum of the cost of the requested resources across the services defined by the BigDataStack Object, where a mapping between resources and a US dollar cost from a commercial cloud provider (Amazon Web Services EC2) is used.
- **MidTierFirst**: This second baseline represents a user selecting resources that are in the middle of the available range, as they don't know what they need. To represent this, we manually ordered the available Resource Templates by requested resources, placing those using mid-tier hardware first, followed by high-tier hardware, and finally putting the lowest-tier hardware at the bottom of the ranking.

CC BY NC ND

8.6.6. Training Procedure

For ADS-Ranking Tier 2, we need to train supervised learning to rank models for each of the two datasets. This in effect creates one model for resource prediction for use with stream processing applications, and one model for use with batch learning applications. To train these models, within each dataset we split the BigDataStack Objects (application components to test) into 5 separate folds. Following a standard cross-fold validation procedure, we then train a model using 4 folds (with 3 folds being used for training and 1 fold used for validation) and 1 fold used for testing. This process is repeated for all 5 fold configurations and performance averaged across the folds.

8.6.7. ADS Ranking Performance Results

In this section we report the performance of the ADS Ranking component when using both the heuristic ranking model (tier 1) and the supervised learning to rank model (tier 2) against the baselines summarized above for each of our two datasets (representing resource prediction scenarios for two different application types). Table 33 reports deployment ranking performance for the Real-time Data Server dataset, while Table 34 similarly reports performance under the Deep Learning dataset. For both datasets we report Success@1, Precision@5, MAP and NDCG metrics. * indicates a statistically significant increase/decrease in performance over the MidTierFirst baseline (paired t-test, $p<0.05$). A bold highlight indicates an increase in performance over the baselines.

Starting with the Real-time Data Server dataset in Table 33, we first observe that ADS-Ranking with the heuristic model is significantly better at recommending deployment configurations than the baselines tested (e.g. 0.5582 vs. 0.2793 NDCG). Moreover, the increase in performance is larger under Precision@5 and MAP (that only consider the user requirements) than under NDCG (which factors in requirements and preferences), indicating that ADS Ranking is much better at meeting at least the minimal user requirements for this application type. Second, comparing the performance of the learning-to-rank approach, we see a further increase in performance (0.5925 vs. 0.5582 NDCG), indicating that the learning-to-rank approach is more effective. On the other hand, current average performance of ADS Ranking appears to be around 0.6, which may indicate that there is still significant scope to improve ranking performance. However, upon further investigation of per-deployment performance, we observed that much of the loss in the reported performances was due to 0-scored experimental scenarios under metrics that score to maximum depth (e.g. MAP and NDCG). A 0-score here means that no valid deployment existed in the Resource Template set, i.e. this is either a failure on the part of Pattern Generation (i.e. it did not produce good Resource Templates), or the quality of service level was impossible to achieve. For this case, it is the latter, where the extreme QoS level could not be met in some cases (i.e. no deployment could successfully meet both the cost per hour and response time requirements).

CC BY NC ND

Table 24 - Deployment Ranking Performance on the Real-time Data Server dataset

Approaches	Success@1	Precision@5	MAP	NDCG
RankByCost	0.0000	0.0111	0.1407	0.2793
MidTierFirst	0.1111	0.1778	0.2260	0.3532
ADS Ranking Tier 1 (Heuristic)	**0.5278**	**0.5500***	**0.5204***	**0.5582***
ADS Ranking Tier 2 (List-wise LTR, LambdaMART)	**0.5972***	**0.5611***	**0.5887***	**0.5925***

Considering our second application type, i.e. a deep learning job, we see from Table 34 a slightly different picture. First, the performance of both the baselines (RankByCost and MidTierFirst) are both much more effective ranking strategies here, as demonstrated by performances in the mid-0.60's). This result tells us something about the dataset itself, i.e. that ranking is 'easier' than for the Real-time Data Server dataset. This is primarily because there are a larger number of deployments that meet the requirements and preferences specified in the QoS levels, meaning that it is much easier to produce a good ranking by chance. Second, comparing the performance of ADS-Ranking against the baselines, as with the Real-time Data Server dataset, we observe that ADS-Ranking using the learning-to-rank model is more effective (by a statistically significant margin, except under Success@1) than the baselines tested, although the degree of improvement is smaller. We also see that the heuristic model is not as effective here, resulting in a small but significant decrease in performance in comparison to the MidTierFirst baseline. This appears to be largely due to the Heuristic model overly favouring lower cost deployments, which can out-right fail due to out-of-memory errors in some learning scenarios.

Table 25 - Deployment Ranking Performance on the Deep Learning dataset

Approaches	Success@1	Precision@5	MAP	NDCG
RankByCost	0.6389	0.6556	0.6639	0.6734
MidTierFirst	0.6944	0.6500	0.6607	0.6736
ADS Ranking Tier 1 (Heuristic)	0.6667*	0.6361*	0.6397*	0.6530*
ADS Ranking Tier 2 (List-wise LTR, LambdaMART)	0.6944	**0.6833***	**0.6836***	**0.6896***

To conclude on the performance of ADS-Ranking, we have seen that over the two different application types (a real-time data server and a deep learning job), the supervised ADS-Ranking model based on learning-to-rank is effective at recommending deployment configurations.

CC BY NC ND

8.7. Summary

In this section we described the changes made to the ADS Ranking and ADS Deploy components during Y3, along with associated evaluation. In summary, both ADS Ranking and ADS Deploy were subject to significant updates to factor in the new Realization Engine component of BigDataStack, as well as better integrate them with that component via operations (see Section 8.4). Meanwhile, the ADS Ranking component was updated to meet the final missing requirements, i.e. to add support for supervised ranking via learning to rank (see Section 8.5). Furthermore, a new dataset was created that represents deployments for deep learning-type jobs (see Section 8.6.3). Finally, extending the evaluation that was reported in D3.2, the final version of ADS-Ranking was evaluated across both datasets, demonstrating that it is able to produce effective rankings of deployments for the user across two categories of application.

In terms of software requirements, ADS-Ranking and ADS Deploy are complete. ADS-Ranking is able to ingest information about the user application from the Realization Engine (REQ-ADSR-01), and also extract features about that application based on predicted Benchmarking Results (REQ-ADSR-02). Two scoring functions are supported, namely heuristic scoring (REQ-ADSR-03) that is unsupervised, and supervised learning-to-rank scoring (REQ-ADSR-04), and these are used to rank deployments for the user (REQ-ADSR-05). These models can also be used for re-ranking in the same manner as part of an operation sequence that performs application adaptation (REQ-ADSR-07). The supervised learning-to-rank model can be trained using metrics exposed by the BigDataStack platform (REQ-ADSR-06) via the Realization Engine. To support the training of this model, we also produced two datasets (REQ-ADSR-08). Meanwhile, for ADS-Deploy, it supports deployment using standardized data formats defined by the Realization Engine for metrics (REQ-ADSD-01), an application definition (REQ-ADSD-02) and resources (REQ-ADSD-03), which extend Kubernetes style objects (REQ-ADSD-05). It also supports deployment integration with the Realization Engine via API (REQ-ADSD-06) that handles deployment scoring (REQ-ADSD-04).

CC BY NC ND

9. Triple Monitoring & QoS Evaluation

Enabling cloud application adaptation, service level objective must be evaluated constantly according to the desired range value defined by the application owner. The platform adapts the application if the collected value violated the agreement. The collection of metrics is performed by scraping each "I" interval of time. For each scrape request, Prometheus gather many samples (data points). This strategy provides the evaluation tools enough data points for avoiding the adaptation on outliers.

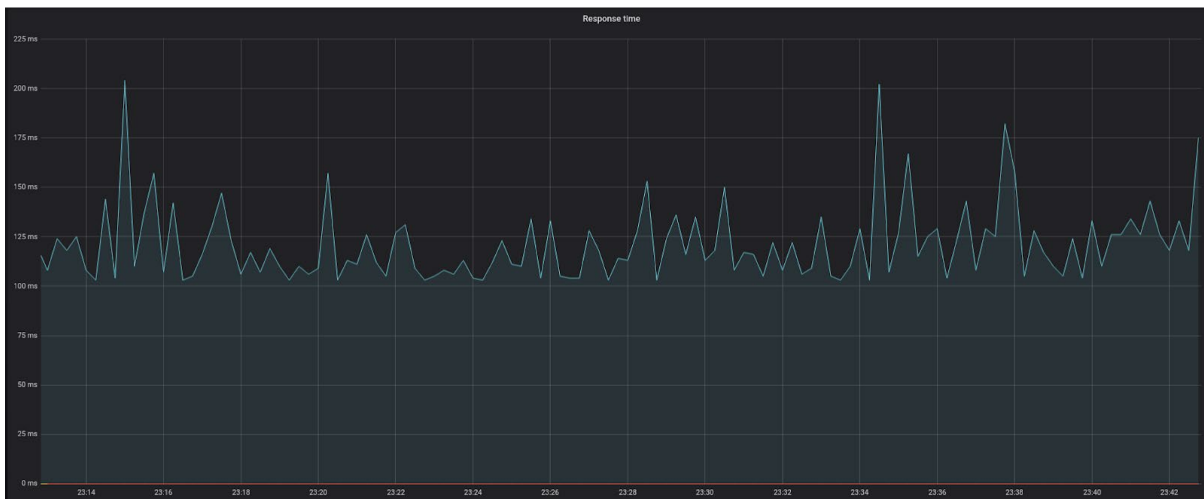


Figure 33: Evaluation by data points

On the response time of the time series shown on the figure above, we can observe that for most of the time, the value in less than 150ms. If the threshold is set to 150ms in the agreement, the QoS evaluator will raise unnecessary violation which will cost in term of resource since many operations will be take place for readapting (scaling) the application and cost to the application owner because of the increase of the resource allocated to the application.

The QoS Evaluator can guarantee the compliance of a SLO for the most part or a given period or time window. We define "for the most part" as the level of confidence we can have in the evaluation of the SLO.

There exist different ways in which we can "assess" a group of data points or measurements to determine whether they comply with the objective "for the most part". One way is to aggregate data points in groups of n and determine whether the group as a whole complies with the objective. Again, there are different aggregation functions we can use: from quantiles/percentiles to mean (average) and median; we chose the former In other words, that, **metric's value** is lower or higher than the objective for the **percentage** of measurements collected in the **time window**.

 bigdatastack.eu

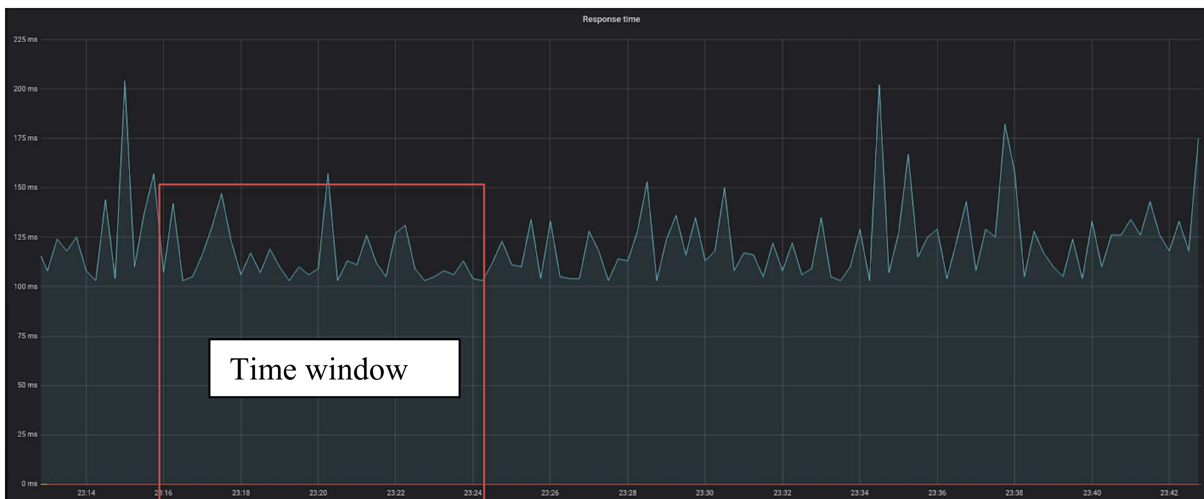


Figure 34: Evaluation by percentile

- *Response time* < *900ms* for *99%* measurements collected in *10min*

This percentage can be calculated as the percentile 99th or 0.99 quantile (also known as 99% quantile), depending on the nomenclature we want to use.

The implementation is the percentile computation is performed in a streaming mode. The manager starts the computation of the percentile when its receives a "qos" request. This request contains the name of the queue to reply to, the name of the request and a list where each element is an object composed by the name of metrics, the percentage, the name of the application producing the corresponding metric, the interval of time of the time window. This request has the following format:

{"request":"qos","queue":"qos",
"metrics":[{"application":"tester","metric":"scrape_duration_seconds","interval":10,"percentage":90}]}

The manager creates a bucket based on the interval of time specified in the request, then it computes the percentile taking into account the percentage.

The output has the following format:

{"application": "tester", "metric": "scrape_duration_seconds", "percentile": "0.016867146", "request": "qos"}.

9.1. Requirements

Requirements did not change from D3.2.

9.2. Design Specifications

The monitoring is collecting metrics from the infrastructure, applications (application specific metrics) and data (data transaction). Achieving this collection requires an extendable/scalable monitoring engine. BigDataStack infrastructure is based on OpenShift

CC BY NC ND

which has its own monitoring system, gathering information related to the nodes, pods and services etc. BigDataStack components are deployed by namespace which provides the flexibility of grouping metrics by component since they can be collected by namespace. The monitoring engine exploits service discovery provided by Prometheus to detect all Prometheus exporter (endpoint exposing metrics in Prometheus format). Some components of BigDataStack such as the CEP engine and the realization engine have internal Prometheus instances. The monitoring engine disposes of technique to extend its collection capability by adding these Prometheus instances. This functionality is provided through the operator and the capability of Thanos components.

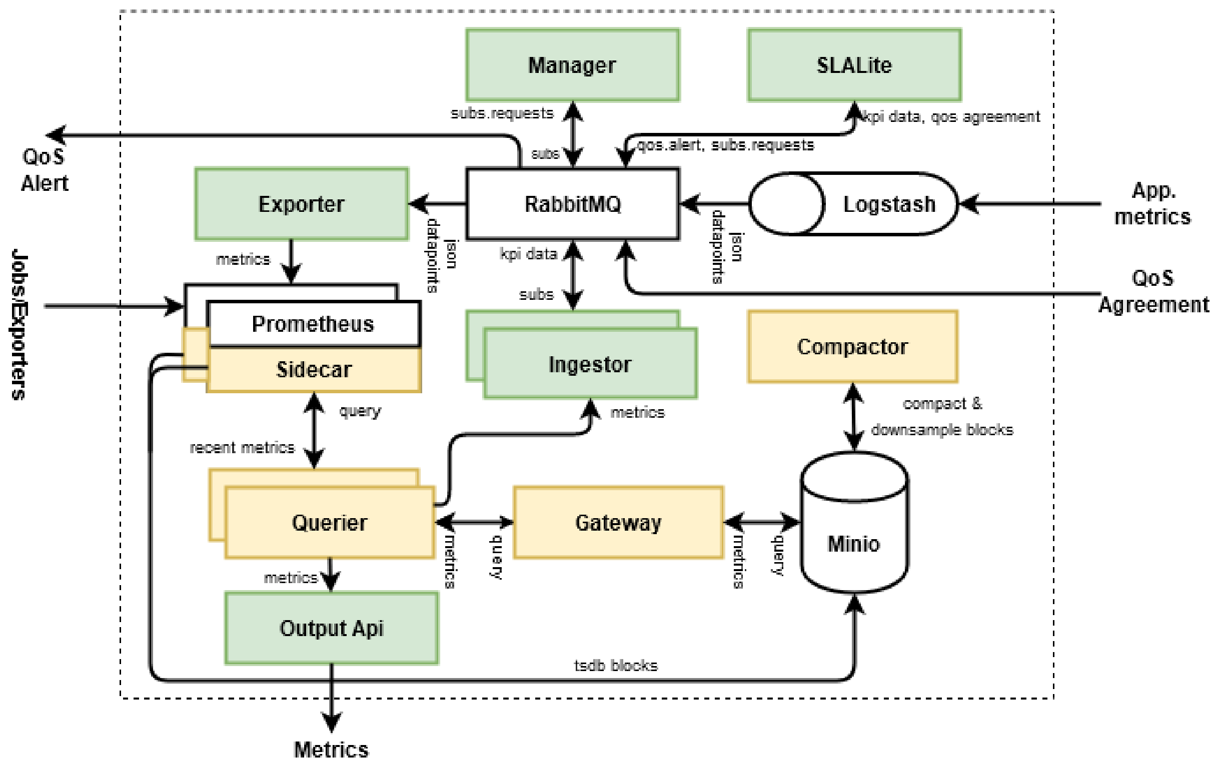


Figure 35: Architecture of the Triple Monitoring Engine

The above picture shows the latest architecture of the TME. For each Prometheus instance we assign a sidecar (Thanos component) which will be connected to the querier. The Querier implement all Prometheus HTTP API. Metrics collected by different Prometheus instances in the platform can be accessed in a single point. The Ingestor requests metrics to the Querier then, publish them to the queue specified in the subscription object. The interval of time by which metrics are published can be altered in the subscription object. To allow the collection of metrics from application that don't have the ability to embed a Prometheus exporter (based on architecture constraint). The triple monitoring engine is receiving their metrics through the Universal exporter collected to RabbitMQ. These applications can publish their metrics over HTTP to an endpoint available through Logstash. Logstash by its capability of handling huge data flux, publishes these metrics to the queue listened by the Universal exporter.

Through the Realization engine which creates the OpenShift object (application pods), applications (use case applications) running on BigDataStack receives the "application ID"

which is the unique identification of the application on BigDataStack and the "object ID" which is the name of the component. Those elements are passed to the application as environmental variable and assigned to each metric. The Triple monitoring engine combines the application id and the object ID to create a unique BigDataStack application identifier which is present in the QoS start request as application field. The manager can create the correct subscription object which will enable the Ingestor to filter the correct metric.

9.2.1. TME Scaling and Long-Term Persistence

The current architecture uses Minio as metric storage for long term retention. Each Prometheus instance connected to the monitoring engine is configured such way to retain metrics for 2 hours. TSDB blocks are moved from each Prometheus instance (volume assigned) to Minio. For storage optimization, samples are compressed (aggregated) then stored to Minio. This operation is performed as routine by the Compactor (Thanos component).

9.3. Experimentation Outcomes

Like in D3.2 "*no individual or specific experiments are conducted for this component; the Triple Monitoring engine and QoS Evaluation (QoSE) play a supportive role to the components bringing the intelligence to the DDIM capability: the ADS Ranking & Deploy and the Dynamic Orchestrator (DO).*" Therefore, for experiments where the TME & QoSE participate are engaged please refer to Sections 7 and 8.

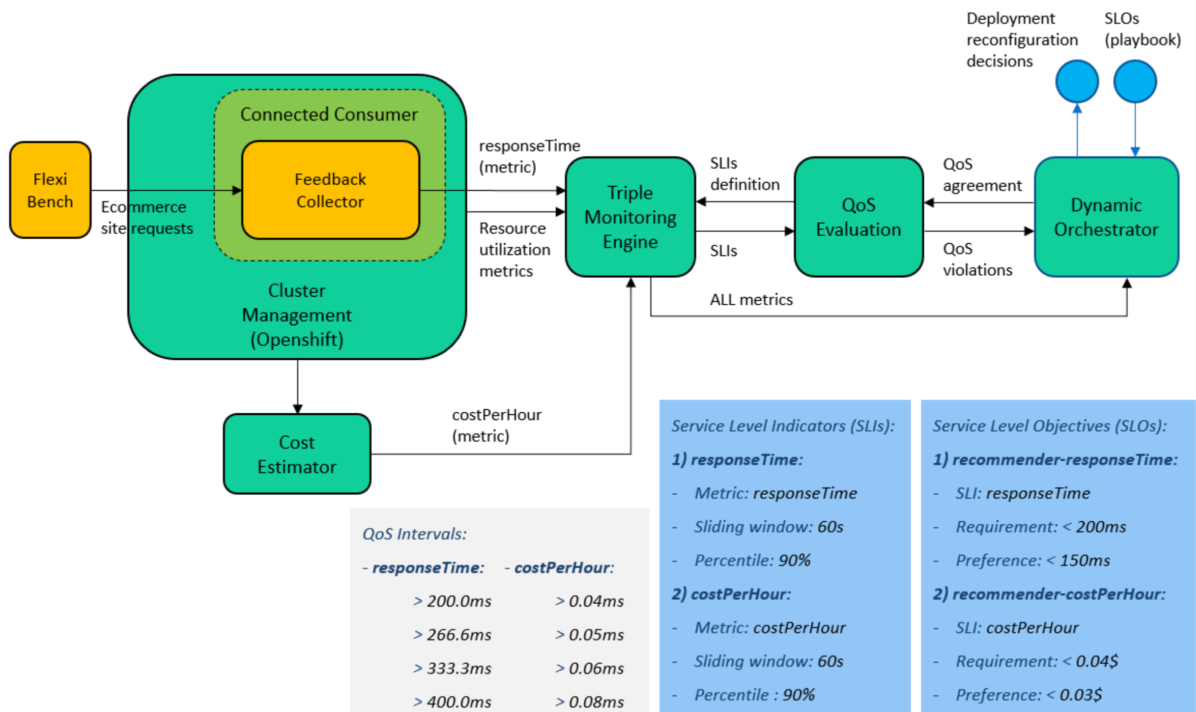


Figure 36: Example of configuration of TME & QoS Evaluation for experimental setting 5 for scenario 4.

For example, Figure 36 shows an example of configuration of the TME & QoS Evaluation for the experimentation of *real-time product recommendation analytics cost-readiness* (scenario 4). In this scenario, **the traffic of users' behavioural events at the EROSKI's ecommerce webpage towards the Feedback Collector service rises dramatically,** which poses a challenge to the Dynamic Orchestrator, that is trading off between the response time (i.e. data freshness or time to value) of the real-time analytics process and the infrastructure resources cost.

9.4. Implementation and Integration Highlights

Figure 37 shows the main interactions in the context of the TME & QoS Evaluation component. There is a close collaboration based on asynchronous message passing among the main subcomponents of that component: QoS Evaluator, RabbitMQ (message queue), Manager and Prometheus-based monitoring system.

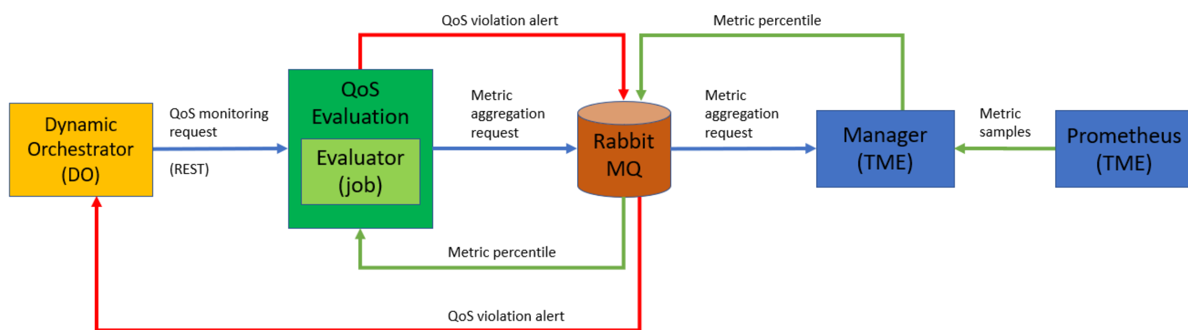


Figure 37: Triple Monitoring Engine & QoS Evaluation integrations.

Within BigDataStack, the Dynamic Orchestrator (DO) is the only consumer of the TME & QoS Evaluation service. This ensures low coupling in the architecture as well as high cohesion. This collaboration ensures that the DO gets notifications of violations of the QoS with respect to certain levels of QoS, and a specific confidence level (e.g. 95%, 99%, 99.99%, etc. See D2.3 for a full description of the *QoS Evaluation Confidence Levels* feature).

9.5. Conclusions

The design of this component, with clear distinction and assignment of responsibilities between the monitoring system (e.g. TME) and the QoS evaluation system, has provided the necessary flexibility to serve the specific needs of the Dynamic Orchestrator. In particular, the requirements for the management of multiple levels of QoS as well as confidence levels, posed a challenge that was analysed and designed in Y2, and implemented and tested in Y3.

Furthermore, more of Y3 was dedicated to the testing and evaluation of the component in real-world conditions. This pinpointed lacks the Y2 designs that needed to be solver in order to provide the solution with high scalability and availability. This results in the integration with Thanos[12], and the open source and highly available Prometheus setup with long term storage capabilities. This has allowed us to go one step further (beyond Prometheus) in the

[12] https://thanos.io/

CC BY NC ND

integration within the cloud-native foundation ecosystem. Furthermore, we plan to submit this component as a CNCF sandbox[13] project in the upcoming months.

[13] https://www.cncf.io/sandbox-projects/

CC BY NC ND

10. Information-Driven Networking

The development focus and enhancements compared with Y2 technical activities are that the respective networking mechanisms (i.e. Kuryr integrated into the OpenShift enabling to avoid the double encapsulation problem due to using two (2) different overlays, namely OpenStack SDN and OpenShift SDN on top, as well as Istio service mesh with sidecar injection enabled) were fully integrated, parameterized and validated in order to serve the application requirements derived by the demonstrators. Specifically, in Y3, we have focused in:

i. **Deployment and configuration** of Istio service mesh with sidecar injection enabled at the BigDataStack Testbed;
ii. **Deployment and configuration** of the telemetry application of Kiali Dashboard to monitor and visualize the structure of the BigDataStack service mesh and display its topology;
iii. **Set up and configuration** of interactions with the Triple Monitoring and QoS Evaluation (i.e. which works along with the Prometheus monitoring system) in order to analyse the enforcement of network policies and prioritization schemes (i.e. response time, requests per second, etc.) based on defined metrics;
iv. **Description of the deployed microservices** by means of network rules configuration (i.e. YAML files) that realize the BigDataStack applications and the interactions between them;
v. **Implementation, network policies enforcement and experimentation over a pluggable layer** enabling traffic prioritization through weighted load balancing, access control and rate limit across diverse protocols and runtimes.

10.1. Requirements

Requirements did not change from D3.2.

10.2. Design Specifications

Through the Information-Driven Networking component the Data Scientist declares her intend to be realized by the underlying system to translate either the data flows or the application requirements into specific networking primitives that achieve the desired Service-Level Objective (SLO). This objective may refer to efficiently handling various kinds of traffic – streams, batches and micro batches – get the isolation/priority of availability and bandwidth that are needed to serve the application. With the convergence of all data and services in the same network mesh, the Information-Driven Networking manages traffic according to the network utilisation, the applications requirements and the communication latency without compromising the functionality of the services. Using policy statements, either the Network Administrators or the Data Scientists can specify which kinds of service / pod need to be given weighted load priorities, at what times and on what part of their communication protocol (TCP, HTTP, etc.). By deploying and configuring Istio service mesh at the BigDataStack testbed all the data metrics are collated by Mixer and stored in Prometheus. Kiali uses the data stored in Prometheus to show the service mesh topology, metrics, traffic information and more. A common set up which concretises the data flow events and the respective logical design is presented as follows.

CC BY NC ND

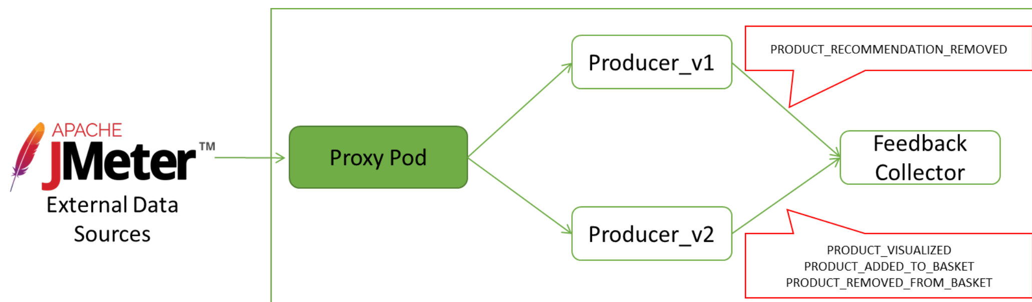


Figure 38: Proxy Pod prioritizes Weighted Load Traffic to the Producer App

In more details, the Proxy Pod based on Istio rules acts as a gateway which receives external traffic. The Proxy Pod through Istio service mesh sidecar splits the requests/traffic based on the application requirements. In Figure 38, the event of "product recommendation rejections" gets more priority compared to the events related with other user interactions to the application, because the former contributes in the re-training/re-calculation of the ML model delivering products recommendations to the end users (i.e. improve the accuracy of the model).

The deployed microservices / pods and their interaction are described in YAML files. In order to enable Istio service mesh for pods at OpenShift Platform, we add "sidecar.istio.io/inject: "true" in the YAML file. The other fields remain the same as in the default OpenShift deployment. The necessary commands are presented as follows:

```
oc apply -n istioapp proxy.yaml

oc apply -n istioapp producer.yaml

oc apply -n istioapp feedbackcollector.yaml
```

The commands below contain the Destination Rule (app subsets) and the Virtual Services (weight and routes) that define the mesh network policies in Istio:

```
oc apply -n istioapp destination_rule.yaml

oc apply -n istioapp routing_subset.yaml
```

In the following, we present an example of initializing the Proxy Pod at the BigDataStack testbed.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: feedbackcollector-proxy
spec:
  hosts:
    - "*"
  gateways:
    - feedbackcollector-gateway
  http:
    - match:
        - uri:
            exact: /feedbacks
      route:
```

cc BY NC ND

```
        - destination:
            host: proxy
            port:
              number: 8083
```

Figure 39: Initialization of the Proxy Pod which splits the events

In addition to this, to address the challenges of a specific application, its requirements and the respective policies enforcement, a set of mechanisms operating at the services layer have been deployed. At the same time, to realize the appropriate attributes in order to weight the traffic towards concrete microservices / pods, we give priority to events of interest based on their type. This functionality implements the policy enforcement endpoint inside the pod as sidecar container in the same network namespace. This approach is highly flexible and HTTP aware and facilitates to apply policies in the support of **operational goals**, such as service routing, prioritization schemes over data flows, retries, circuit-breaking, etc.

The Information-Driven Networking mechanisms also operate at the application layer. The latter gives the advantage of being universal. Our focus is to address the challenges arising from the diverse **data types** (i.e., stream, micro-batch, batch) to enforce policies to DNS, storage services (i.e., scalable storage of LeanXscale, Object Store, etc.), real-time streaming, ML model incremental training/update and a plethora of other services that do not use HTTP. The workloads in the BigDataStack environment can communicate without IP encapsulation or network address translation for bare metal performance, which enables easier troubleshooting, and better interoperability. In settings that require an overlay, the Information-Driven Networking mechanisms also support tunnelling. This approach is universal, highly efficient, and isolated from the pods and facilitates to apply policies also related with data privacy goals. In the following, we present the main service / networking configuration of controlling communications to HTTP GET/POST requests by giving an indicative network policy definition which prioritizes events accordingly.

```
# On poll producer service handles weighted traffic (i.e. with 90-10 priority).

# On feedbacks events are being prioritized towards the defined subsets / destinations.

apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: producer
spec:
  hosts:
    - producer
    - proxy
    - feedback-collector
  http:
    - match:
        - uri:
            exact: "/poll"
      route:
        - destination:
            host: producer
            subset: v1
          weight: 90
        - destination:
            host: producer
            subset: v2
```

```
        weight: 10
    - match:
        - uri:
            exact: "/feedbacks/PRODUCT_RECOMMENDATION_REMOVED"
      route:
        - destination:
            host: producer
            port:
              number: 8083
            subset: v1
    - match:
        - uri:
            exact: "/feedbacks/PRODUCT_VISUALIZED"
      route:
        - destination:
            host: producer
            port:
              number: 8083
            subset: v2
    - match:
        - uri:
            exact: "/feedbacks/PRODUCT_ADDED_TO_BASKET   "
      route:
        - destination:
            host: producer
            port:
              number: 8083
            subset: v2
    - match:
        - uri:
            exact: "/feedbacks/PRODUCT_REMOVED_FROM_BASKET"
      route:
        - destination:
            host: producer
            port:
              number: 8083
            subset: v2
```

Figure 40: An indicative network policy definition for controlling HTTP GET/POST requests

10.3. Experimentation Outcomes

The **Data Scientist** uses the **Information-Driven Networking (IDN)** tool to define metadata and means of service mesh communication in order to apply tailored controls to data intensive operations (e.g. data streams requiring concrete prioritization schemes or weighted load balancing) and applications related with data intensive tasks (e.g. prioritizing user-generated data to facilitate an ML model update/recalculation based on events of special focus) according to specific requirements, by also considering:

- The identification of the end-to-end application objectives in terms of specifying KPIs and criteria for optimal networking management and engineering (i.e. response time, requests per second, throughput, jitter);
- The definition of the constraints arising from the type of data to be processed (prioritization of specific events, liveness, readiness among services) and the requirements of the application (time criticality, accuracy improvement of ML model, security, privacy);

- The validation of the applied network controls through the assessment of the corresponding metrics exposed by Prometheus used by the Triple Monitoring and QoS Evaluation. The validation step is required in order to evaluate that the policies have been correctly enforced and that resources are distributed among consumer or producer services/applications, as requested.

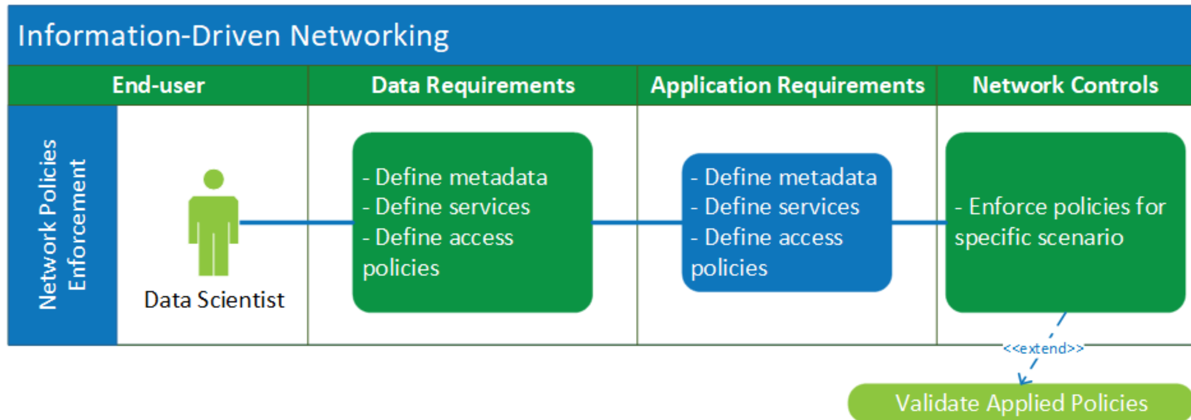
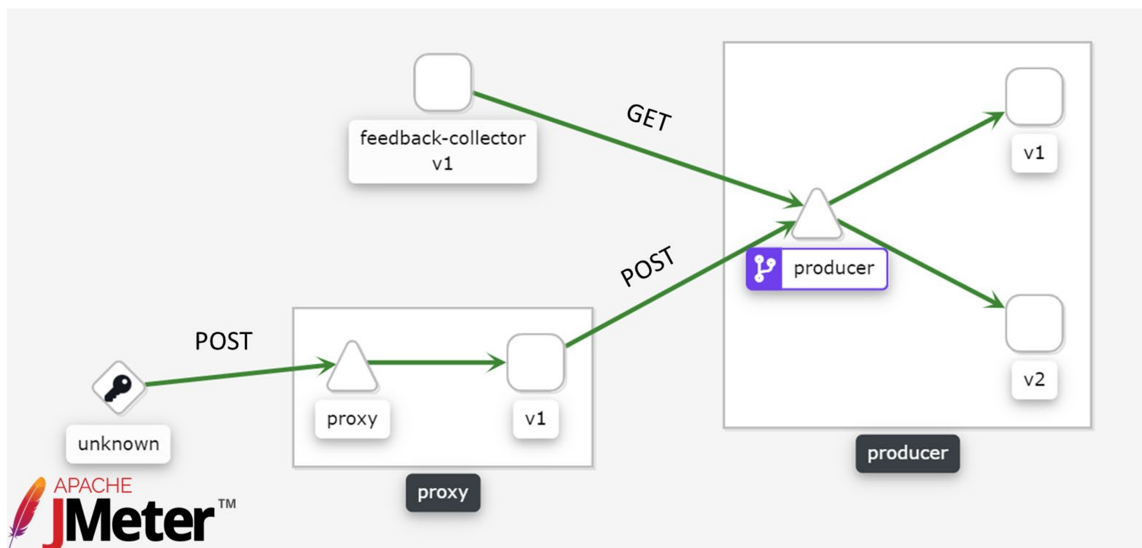


Figure 41: Mapping of Information-Driven Networking tool with BDS Use Cases

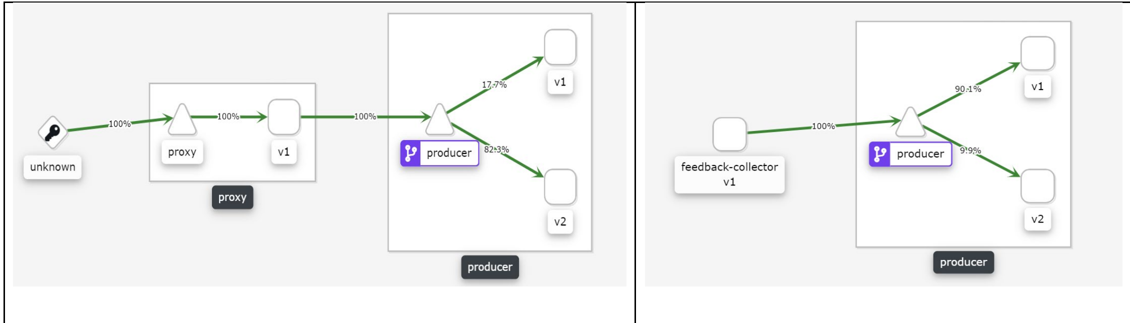
The IDN plays a supportive role to the components bringing the intelligence to the Data-Driven infrastructure Management: the ADS Ranking & Deploy and the Dynamic Orchestrator (DO). It also interacts with the Triple Monitoring and QoS Evaluation to collect metrics w.r.t. response time, requests/traffic rate per second, request volume, request duration, etc., which are relevant to the application requirements. The experimental settings of the Information-Driven Networking are broken into 3 steps, as follows:

i. **Flow of Requests**, which includes the Initialization of the respective services at the Istio/Kiali contexts. The Proxy Pod receives data from external sources based on the defined rules and splits the traffic in other serving pods (i.e. producer subset). Finally, feedback collector makes an HTTP GET request at the producer which responds based on the defines traffic (e.g. v1 = 90% and v2 = 10%).



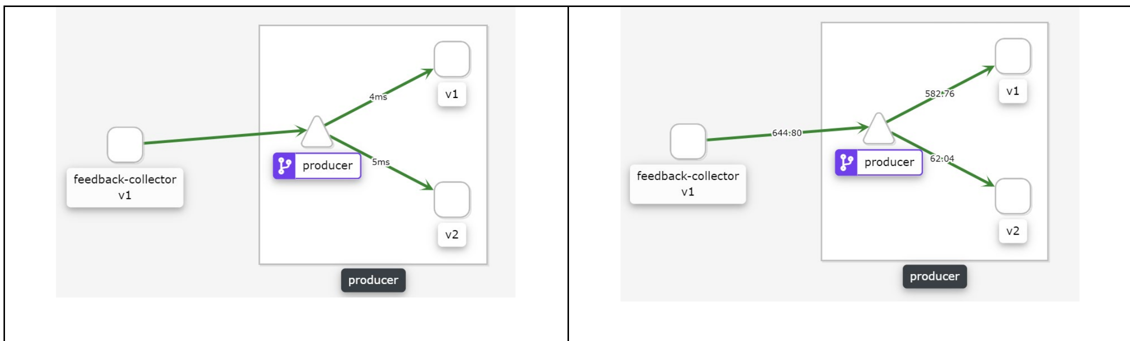
ii. **The decomposition of the respective services within the Istio service mesh**, as

presented in the below figures. Kiali communicates with Prometheus and gets metrics (i.e. % weighted requests) about how producer service interacts with input (left side) / output (right side) requests.



iii. **The Visualization of the results in Kiali** including response time (figure on the left hand side) and request per second (figure on the right hand side) for the incremental ML model updates.

iv.



The producer log records according to the prioritized events are split between EVENTS of TYPE A (i.e. PRODUCT_RECOMMENDATION_REMOVED) and EVENTS of TYPE B (i.e. the rest of customer events), as depicted in the following.

Figure 42: Producer logs according to the event type

Kiali Dashboard visualizes mesh network health between the interacting services where the producer routes the weighted loads (v1 vs. v2) to the feedback collector. It is shown that in some cases (i.e. 92.9%) the requests reach their destination while in some extreme cases (i.e. 7.1%) the Proxy Pod faces some requests loss.

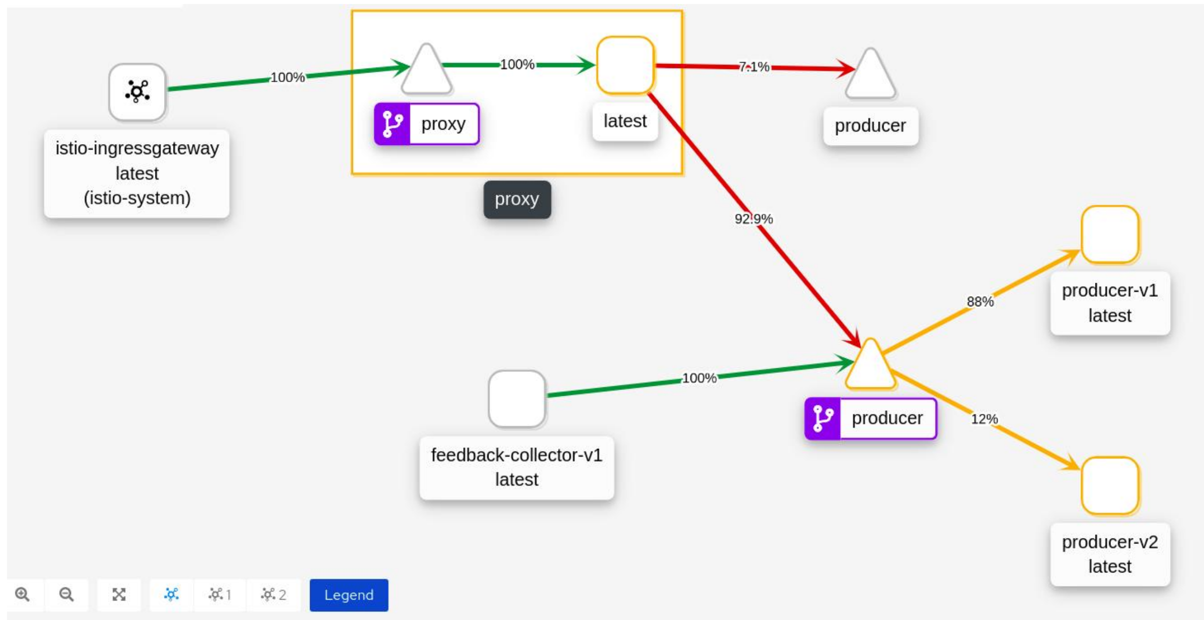


Figure 43: Service Mesh Health Check through Kiali

The Prometheus graph depicted in the following presents the total requests for the producer (i.e. the case of this destination app) where the prioritized data flow is greater (i.e. v1) than the common data flow (i.e. v2).

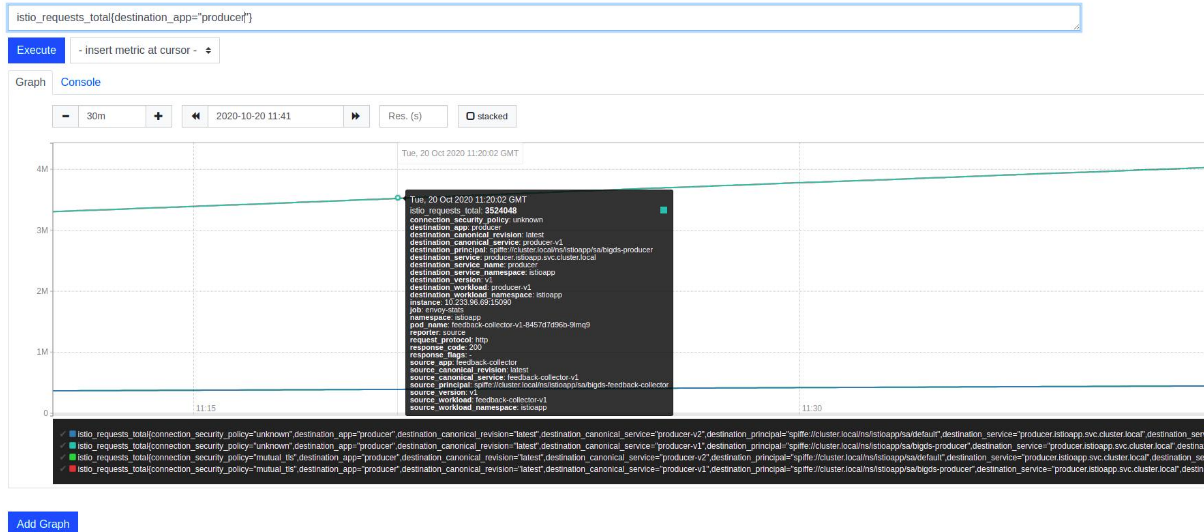


Figure 44: Total requests collected by Prometheus

10.4. Implementation and Integration Highlights

The Information-Driven Networking component combines the OpenShift Network Policies[14], services and routes to handle Ingress or Egress traffic in the cloud infrastructure at the Network, Transport and Application Level with the Istio[15] open source service mesh that

[14] https://docs.openshift.com/container-platform/4.1/networking/configuring-networkpolicy.html
[15] https://istio.io

cc BY NC ND

transparently layers the services / pods. The service mesh is used to describe the network of containerized microservices / pods that interact in the BigDataStack Testbed. As the service mesh grows in size and complexity, it achieves efficiency in service discovery, load balancing, failure recovery, metrics, and monitoring. In more details, we implemented 3 spring boot applications in the support of the respective services: proxy, producer, and feedback collector. In the frame of interactions of IDN with the Triple Monitoring & QoS, the metrics are stored in Prometheus, while Kiali uses these metrics to show the service mesh topology, metrics, traffic information and more.

In this direction, we deploy special sidecar proxies throughout the BigDataStack environment which intercept all network communication between microservices. The key capabilities include the efficient traffic management, incorporating the rules configuration and traffic routing, which controls the traffic flows and API calls between services / pods.

10.5. Conclusions

Overall, the design and implementation of this component required the deployment and configuration of the Istio service mesh, Kiali Dashboard and Prometheus Dashboard coupled with service mesh observability functionalities. Specifically, Istio enabled to create a mesh network over the BigDataStack pods for better traceability and monitoring of the deployed services with weighted load balancing and service-to-service interaction capabilities.

At the same time, Kiali facilitated to monitor traffic flows produced by the services of the mesh, visualise how they are connected as well as operations, updates, prioritized processes by means of network policies which can be enforced. In the context of BigDataStack project we worked towards the definition and set up of some complex scenarios focusing on weighted load balancing which resulted in traffic prioritization and therefore featuring data flows distribution controls over the Testbed. In this way, we manage to prioritize traffic to the workload instances which better serve the application requirements and are met by improving / updating the accuracy of the ML model. Finally, this ML model is part of the front-end application which calculates the recommendations delivered to the customers / end users.

CC BY NC ND

11. References

[1] Network Policies in Kubernetes. Available Online: https://kubernetes.io/docs/concepts/services-networking/network-policies/

[2] Project Calico. Available Online: https://www.projectcalico.org/

[3] Istio. Available Online: https://istio.io/

[4] de Vaulx, Frederic J., Eric D. Simmon, and Robert B. Bohn (2018). "Cloud computing service metrics description." Special Publication (NIST SP)-500-307. 2018.

[5] William Voorsluys, James Broberg, Srikumar Venugopal, Rajkumar Buyya, Martin Gilje Jaatun, Gansen Zhao, Chunming Rong (2009). "Cost of Virtual Machine Live Migration in Clouds: A Performance Evaluation", Cloud Computing, Springer Berlin Heidelberg, 2009, P 254-265

[6] D. Guyon, A. Orgerie, C. Morin and D. Agarwal (2017). "How Much Energy Can Green HPC Cloud Users Save?" in 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), St. Petersburg, 2017, pp. 416-420.

[7] Gulisano, V., Jimenez-Peris, R., Patino-Martinez, M., Soriente, C., & Valduriez, P. (2012). "Streamcloud: An elastic and scalable data streaming system." IEEE Transactions on Parallel and Distributed Systems, pp. 2351-2365.

[8] H. Rui et al. (2014). "Enabling cost-aware and adaptive elasticity of multi-tier cloud applications." Future Generation Computer Systems, pp. 82-98.

[9] Kalervo and Jaana. (2002). "Cumulated gain-based evaluation of IR techniques." ACM Transactions on Information Systems (TOIS), pp. 422--446.

[10] L. Tie-Yan. (2009). "Learning to rank for information retrieval." Foundations and Trends in Information Retrieval, pp. 225-331.

[11] M. Ferdman et al. (2012). "Clearing the clouds: a study of emerging scale-out workloads on modern hardware." ACM SIGPLAN Notices, pp. 37-48. ACM.

[12] Raschke, R. (2010). "Process-based view of agility: The value contribution of IT and the effects on process outcomes." International Journal of Accounting Information Systems, 11(4), pp. 297-313.

[13] Salton and McGill. (1986). "Introduction to modern information retrieval." McGraw-Hill, Inc.

[14] Sergey and Christian. (2015). "Batch normalization: Accelerating deep network training by reducing internal covariate shift." arXiv preprint.

[15] Z. Jia et al. (2013). "Characterizing data analysis workloads in data centers." IEEE International Symposium on Workload Characterization (IISWC), pp. 66-76. IEEE.

[38] Mossalam, H., Assael, Y. M., Roijers, D. M., & Whiteson, S. (2016). Multi-objective deep reinforcement learning. *arXiv preprint arXiv:1610.02707*.

[39] Mao, H., Alizadeh, M., Menache, I., & Kandula, S. (2016, November). Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks* (pp. 50-56). ACM.

[40] Mao, H., Netravali, R., & Alizadeh, M. (2017, August). Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (pp. 197-210). ACM.
[41] Bu, X., Rao, J., & Xu, C. Z. (2009, June). A reinforcement learning approach to online web systems auto-configuration. In *2009 29th IEEE International Conference on Distributed Computing Systems* (pp. 2-11). IEEE.
[42] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). Openai gym. *arXiv preprint arXiv:1606.01540*.

[43] Breakout-v0 – Openai gym. https://gym.openai.com/envs/Breakout-v0/

[44] Fadel Argerich, M., Cheng, B., & Fürst, J. (2019). Reinforcement Learning based Orchestration for Elastic Services. *arXiv preprint arXiv:1904.12676*.
[45] Norbert Fuhr. 1989. Optimum Polynomial Retrieval Functions Based on the Probability Ranking Principle. ACM Transactions on Information Systems , Vol. 7, 3 (1989), 183--204.
[46] Fredric C. Gey. 1994. Inferring Probability of Relevance Using the Method of Logistic Regression. In 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. 222--231.
[47] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. 2005. Learning to rank using gradient descent. In 22nd International Conference on Machine Learning . 89--96.
[48] Xuanhui Wang, Cheng Li, Nadav Golbandi, Michael Bendersky, and Marc Najork. 2018b. The LambdaLoss Framework for Ranking Metric Optimization. In 27th ACM International Conference on Information and Knowledge Management. 1313--1322.
[49] Fen Xia, Tie-Yan Liu, Jue Wang, Wensheng Zhang, and Hang Li. 2008. Listwise Approach to Learning to Rank: Theory and Algorithm. In 25th International Conference on Machine Learning. 1192--1199.
[50] Olivier Chapelle, Donald Metzler, Ya Zhang, and Pierre Grinspan. 2009. Expected Reciprocal Rank for Graded Relevance. In 18th ACM Conference on Information and Knowledge Management. 621--630.
[51] Rama Kumar Pasumarthi, Sebastian Bruch, Xuanhui Wang, Cheng Li, Michael Bendersky, Marc Najork, Jan Pfeifer, Nadav Golbandi, Rohan Anil and Stephan Wolf. 2019. Tf-ranking: Scalable tensorflow library for learning-to-rank. Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2970—2978.
[52] Christopher J.C. Burges. 2010. From RankNet to LambdaRank to LambdaMART: An Overview . Technical Report Technical Report MSR-TR-2010--82. Microsoft Research.
[53] Mengting Wan, Di Wang, Jie Liu, Paul Bennett, and Julian McAuley. 2018. Representing and Recommending Shopping Baskets with Complementarity, Compatibility and Loyalty. In Proceedings of the 27th ACM International Conference on Information and Knowledge Management. 1133–1142
[54] Zaiqiao Meng, Richard McCreadie, Craig Macdonald, and Iadh Ounis. 2019. Variational Bayesian Context-aware Representation for Grocery Recommendation. In Workshop on Context-Aware Recommender Systems.

cc BY NC ND