



Project acronym:	PrEstoCloud
Project full name:	Proactive Cloud Resources Management at the Edge for efficient Real-Time Big Data Processing
Grant agreement number:	732339

D6.1 Architecture of the PrEstoCloud platform

Deliverable Editor:	Nenad Stojanovic, Nissatech
Other contributors:	SOFTWARE AG, ActiveEON ,ICCS ,CNRS, Ubitech, JSI
Deliverable Reviewers:	Giannis Ledakis (Ubitech) George Kioumourtzis (Aditess)
Deliverable due date:	31/03/2018
Submission date:	30/06/2018
Distribution level:	Public
Version:	1.0

This document is part of a research project funded
by the Horizon 2020 Framework Programme of the
European Union



Change Log

Version	Date	Amended by	Changes
0.1	16/03/2018	Nenad Stojanovic	Initial structure based on D2.3
0.2	01/05/2018	Ubitech	Table of Contents
0.3	10/05/2018	All	Initial content, interfaces
0.4	21/05/2018	All	Updated content, interfaces
0.5	28/05/2018	All	Updated content, interfaces Use cases input
0.6	28/05/2018	All	Updated content, interfaces, architecture
0.7	05/06/2018	All	Updated content, interfaces, architecture
0.7.1	30/05/2018	Ubitech	Section 5 content
0.75	08/06/2018	All	Updated content, interfaces, architecture
0.8	12/06/2018	All	Updated content, interfaces, architecture
0.81	14/06/2018	All	Updated content, interfaces, architecture
0.84	16/06/2018	All	Updated content, interfaces, architecture
0.87	19/06/2018	All	Updated content, interfaces, architecture
0.89	22/06/2018	All	Updated content, interfaces, architecture
0.9	25/06/2018	Nissatech	Harmonization
0.95	28/06/2018	Ubitech, Aditess	Review
1.0	30/06/2018	Nissatech	Ready for submission

Table of Contents

Change Log.....	2
Table of Contents.....	3
List of Figures.....	5
List of Abbreviations.....	6
1. Executive Summary.....	7
2. Introduction.....	8
2.1 Scope.....	8
2.2 Relation to PrEstoCloud Tasks.....	8
2.3 Structure.....	8
3. PrEstoCloud Integrated Framework Architecture.....	9
3.1 The PrEstoCloud Platform.....	9
3.2 Detailed Description of the Components.....	10
3.3 Detailed Description of the Interfaces.....	11
3.3.1 Interfaces of the Meta-management Layer Components.....	11
3.3.2 Interfaces of the Control Layer Components.....	21
3.3.3 Interfaces of the Cloud-Edge Communication Layer.....	33
3.3.4 Communication through the Broker.....	37
4. Requirements Refinement.....	45
5. Technical Integration and Planning.....	46
5.1 Introduction.....	46
5.2 Integration at Deployment Level.....	47
5.2.1 Using Docker Compose for Integration.....	47
5.2.2 Sandbox-based approach.....	48
5.3 Integration at Interface Level.....	48
5.4 Code Level Integration.....	48
5.5. Knowledge level integration.....	49
5.6 Integration Planning.....	49
5.6.1 Multi-Iteration/Release Plan.....	50
6. Conclusions.....	52
7. References.....	53
Appendix A – Broker: additional instructions related to the architecture.....	54
A1. The structure of the topics.....	54
A2. Broker Usage Guide.....	56
Appendix B – Monitoring the Edge: On/Offloading Client.....	66
Appendix C – Technical integration in Use cases.....	69

LiveU Use Case.....	69
CVS Use Case.....	70
Aditess Use Case.....	71

List of Figures

Figure 1: PrEstoCloud Initial Conceptual Architecture (D2.3)-----	9
Figure 2: PrEstoCloud Integrated Framework Architecture-----	9
Figure 3: Meta-Management Layer Component diagram-----	20
Figure 4: Meta-Management Layer Sequence diagram-----	21
Figure 5: Control Layer Component diagram-----	32
Figure 6: Cloud-Edge Layer Component and Sequence diagram-----	33
Figure 7: Sequence diagram for (i) Registration of edge node and (ii) New deployment or reconfiguration of the application.-----	37
Figure 8: Update in the list of requirements-----	45
Figure 9: PrEstoCloud project group in GitLab-----	49
Figure 10: PrEstoCloud Milestones as part of the development and integration plan	50
Figure 11: Meaning of icons used to show the functionality of On/Offloading Agent	66
Figure 12: Both aggregating and processing data are performed in the container running on the edge node-----	66
Figure 13: Presence of mobile phone or tablet-----	67
Figure 14: Stop the processing data on the edge node and start the processing data on the cloud performed by the On/Offloading Client-----	67
Figure 15: Stop both aggregating and processing data on the edge node and start them on the cloud performed by the On/Offloading Client-----	68
Figure 16: Mapping of the LiveU use case to the conceptual architecture-----	69
Figure 17: Mapping of the CVS use case to the conceptual architecture-----	70
Figure 18: Mapping of the Aditess use case to the conceptual architecture-----	72

List of Abbreviations

The following table presents the acronyms used in the deliverable.

<i>Abbreviation</i>	<i>Description</i>
AMQP	Advanced Message Queuing Protocol
CPU	Central Processing Unit
DSL	Domain Specific Languages
eNB	Evolved Node B
GPU	Graphics Processing Unit
HDA	Highly Distributed Applications
MEC	Mobile Edge Computing
PNP	ProActive Network Protocol
REST	Representational State Transfer
SCP	Secure copy protocol
TOSCA	Topology and Orchestration Specification for Cloud Applications
UAV	Unmanned Aerial Vehicle
UML	Unified Modelling Language
VM	Virtual Machine
XMI	XML Metadata Interchange
XML	Extensible Markup Language

1.Executive Summary

This deliverable reports on the work performed under the task 6.2 (PrEstoCloud detailed architecture design). The main objective was to design the detailed architecture of the PrEstoCloud platform.

This document is the continuation of the work reported in the deliverable D2.3, where the initial architecture and the interfaces between the components were defined. This work was focused on the analysis and refinement of every possible interaction in the system architecture. Special attention was given to the communication with the Broker (message-oriented middleware).

The architecture represents a distributed event driven architecture, enabling modern Edge-Cloud processing pipelines. The main goal of the deliverable was to document the details of the functionalities and the communication between all components, driven by the scenarios defined in the deliverable D2.3.

One of the main challenges was the definition of a proper structure of the topics that will be used as the structure for the exchange of the data over the broker (in a pub-sub oriented way). This task was especially complex due to a need for defining a minimal but complete set of topics that will cover different monitoring (from the edge and cloud infrastructure) and data processing requirements.

The main outcome is the detailed architecture that serves as the basis for the development of the integrated system. All interfaces are clearly defined and documented. Moreover, this document should be seen as an evolution of D2.3, whereas the information related to the description of the components are provided in D2.3 in a complete form.

The requirements are refined based on the progress in the development of the planned technology. One of the most important open issues is the compliance with GDPR.

The work was performed in a very close and intensive communication between all technical and use case partners, intending to clarify the smallest details in the proposed interfaces. There were around ten iterations of the architecture until all requirements were satisfied.

The results of this work have been already used in the integration activities in the scope of WP6.

2. Introduction

2.1 Scope

This document is the continuation of the work reported in the deliverable D2.3, where the initial architecture and the interfaces between the components were presented. The work was focused on the analysis and refinement of every possible interaction in the system architecture. Special attention was given to the communication with the Broker (message-oriented middleware)

The architecture represents a distributed event driven architecture, enabling modern Edge-Cloud processing pipelines. The main goal of the deliverable was to document the details of the functionalities and the communication between all components, driven by the scenarios defined in the deliverable D2.3.

This document should be seen as an evolution of D2.3, whereas the information related to the description of the components is provided in D2.3 in a complete form. Moreover, particular components are elaborated in corresponding deliverables, which reported the work in WP3-5.

The main outcome is the detailed architecture that serves as the basis for the development of the integrated system. All interfaces are clearly defined and documented.

Beside D2.3, this deliverable is strongly influenced by the other activities related to the development of the system and use cases, esp.:

- D2.2 that provides the list of requirements for the development of the system
- D7.2 that provides pilots and validation plan.

We also analysed the validity of the requirements based on the progress in the development of the particular components.

This work will be used (and continued) in the work in the context of the work package WP6, which is related to the integration activities.

2.2 Relation to PrEstoCloud Tasks

This deliverable reports on the work performed under the task 6.2 (PrEstoCloud detailed architecture design), which main objective was to design the detailed architecture of the PrEstoCloud platform.

2.3 Structure

The document is structured in the following way:

- In Section 3 we provide the update of the architecture and detailed the interfaces between components.
- In Section 4 we provide an update of the requirements based on the technological development.
- Section 5 describes our approach for the technical integration and some details related to the integration planning.
- Section 6 contains concluding remarks.

The deliverable includes several appendices to provide more details about some components and the use cases.

3. PrEstoCloud Integrated Framework Architecture

3.1 The PrEstoCloud Platform

In this subsection we present the updated architecture, which will serve as the basis for the development of the integrated prototype. In deliverable D2.3 we provided the initial architecture, illustrated in Figure 1 in order to enable understanding the changes made in the scope of this deliverable (cf. Figure 2).

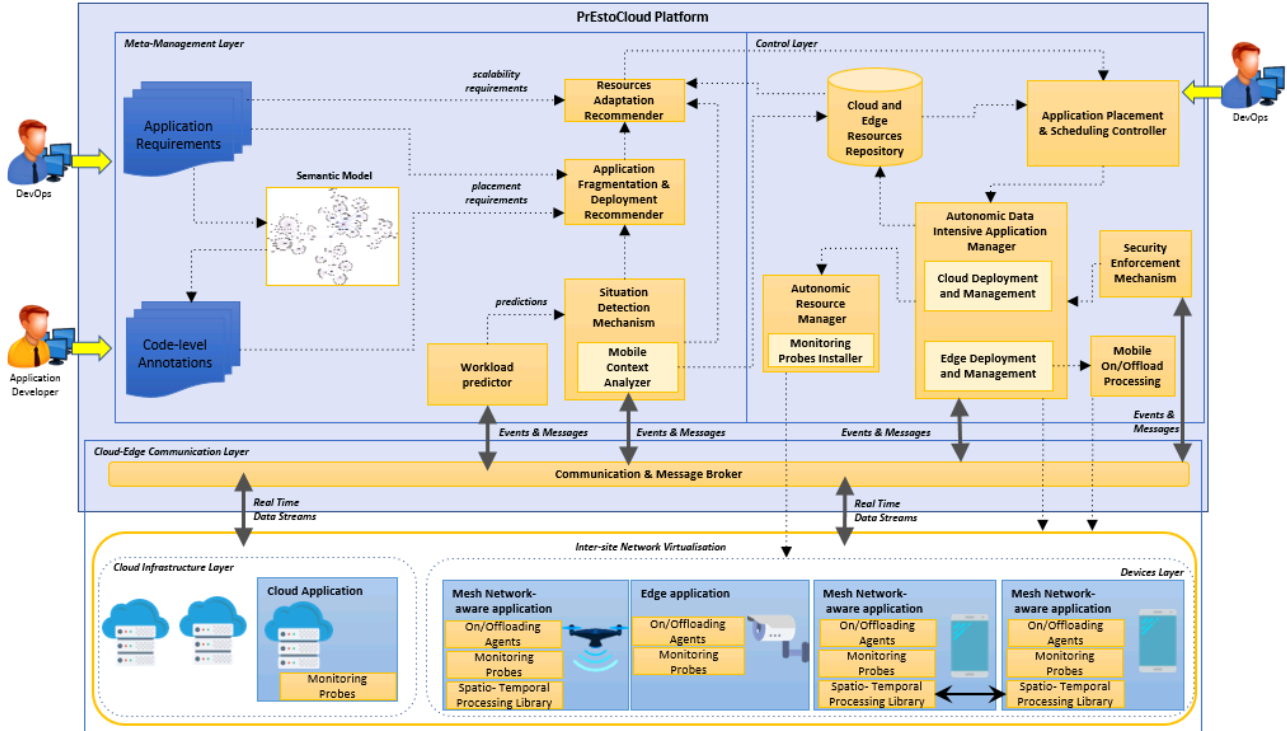


Figure 1: PrEstoCloud Initial Conceptual Architecture (D2.3)

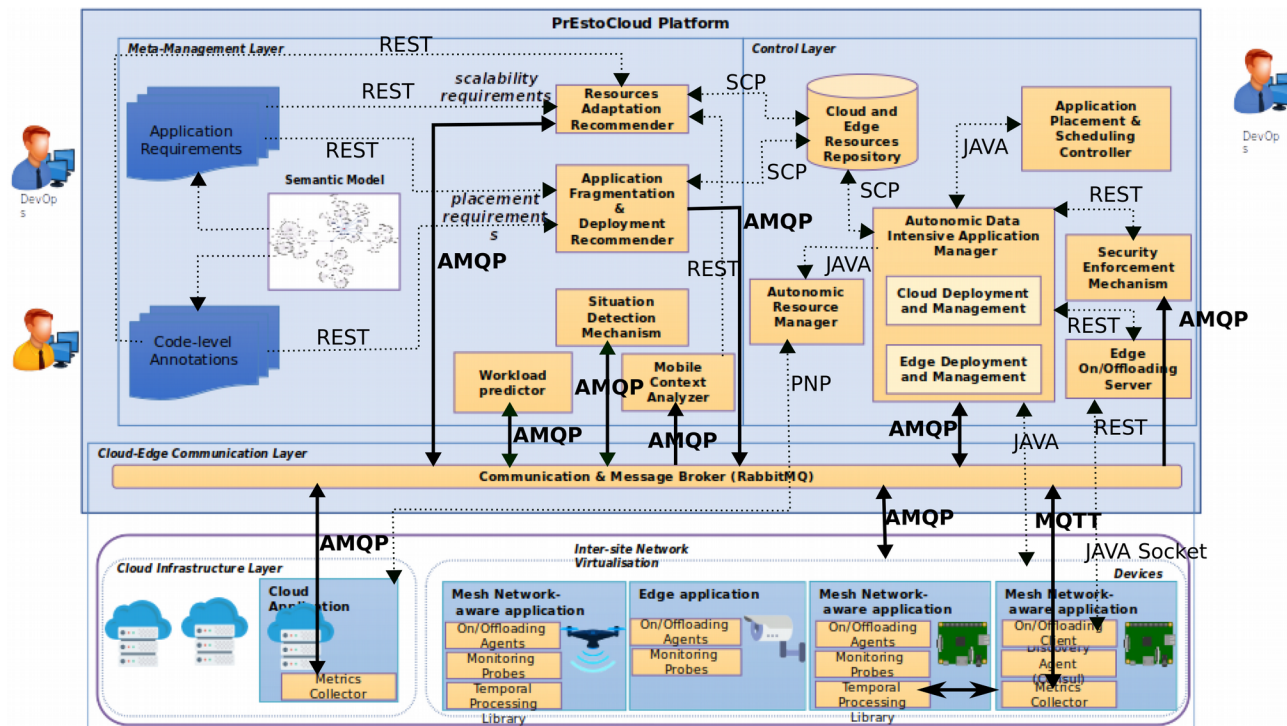


Figure 2: PrEstoCloud Integrated Framework Architecture

By comparing two architectures in details, it becomes clear that the initial architecture was complete regarding the components (position and nature), but that the interactions between components are refined and expanded in order to enable better understanding of the data flow and to support a better preparation of the integration tasks. Indeed, there was only one slight change in the placement of one component (Mobile Context Analyzer is not anymore placed within Situation Detection Mechanism).

This shows that our initial analysis performed in the scope of D2.3 resulted in a very sound architecture that furthermore indicates a well-understanding of the system and the role of particular components from an early phase in the system development. This is the reason for not repeating the description of the components in this deliverable (complete information is provided in D2.3).

Despite this positive experience, there was a huge effort required by all partners in order to provide a full integration picture, i.e. to analyse, understand, define and agree on every interaction in the system. There were around ten iterations of the interaction interfaces until all requirements were satisfied.

We argue that the detailed architecture presented in Figure 2 represents a very sound basis for an efficient development of the integrated system. Indeed, the architecture has been already used in the preparation of M18 prototype.

3.2 Detailed Description of the Components

As already mentioned, this information is provided in D2.3 “Requirements for the PrEstoCloud platform” in a complete form. In order to enable an easier understanding of the rest of this section, in the following table we provide the list of components.

Component	Responsible Partner	Layer
Workload Predictor	NISSATECH	MetaManagement
Mobile Context Analyzer	ICCS	MetaManagement
Situation Detection Mechanism	ICCS	MetaManagement
Application Fragmentation & Deployment Recommender	ICCS	MetaManagement
Resources Adaptation Recommender	ICCS	MetaManagement
Autonomic Data Intensive Application Manager	ACTIVEEON	Control
Autonomic Resource Manager	ACTIVEEON	Control
Application Placement & Scheduling Controller	CNRS	Control
Security Enforcement Mechanism	UBITECH	Control
Edge On/Offloading Server	JSI	Control
Communication and Message Broker	NISSATECH	Cloud-Edge

		Communication
Spatio-Temporal Processing Library	UBITECH	Cloud-Edge Communication
Inter-Site Network Virtualization	CNRS	Cloud-Edge Communication
On/Offloading Agents	JSI	Cloud-Edge Communication

3.3 Detailed Description of the Interfaces

This section gathers information about the interfaces required for the implementation of the integrated solution by defining the communication between the components developed in WP2-3-4-5.

The following subsections describe these interfaces (organized per activity) by detailing the following information:

- **Description:** describes the purpose of the interface.
- **Component providing the interface:** describes the component that is offering the described interface.
- **Consumer components:** describes the components that are using the described interface.
- **Type of interface:** REST, XML-RPC, GUI, Java API etc.
- **Input data:** describes data that is required by the described interface (e.g.: Methods or Endpoints, values and parameters of the interface)
- **Output data:** describes the data that is returned by the described interface (e.g.: the returned data of methods or REST call)
- **Constraints:** Any security or authentication related topics regarding this interface, specifically the need to use a secure transfer protocol. Also, any other constraints (e.g. specific prerequisites, data-types, encoding, transfer rates) which apply to the interface.
- **State:** Synchronous/Asynchronous, Stream
- **Responsibilities:** Partner that is responsible for the implementation and usage of the interface

3.3.1 Interfaces of the Meta-management Layer Components

SubscribeToEvents and ProvidePredictions Interface (Workload Predictor and Communication & Message Broker interface)

SubscribeToEvents and Provide Predictions

Description	This interface allows Workload Predictor to receive monitoring data from cloud and edge devices, and provide other components with workload predictions for specific device.
--------------------	--

Component providing the interface Communication & Message Broker

Consumer components or External Entities Workload Predictor

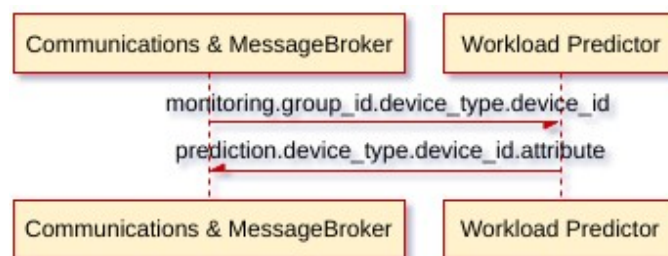
Type of Interface AMQP

State Asynchronous

Input data / Output Data	Methods or endpoints of the interface	Parameters of the method	Return Values of the method
	monitoring.<group_id>.<device_type>.<device_id>	-	JSON-formatted string with values of attributes and timestamp
	prediction.<device_type>.<device_id>.<attribute>	JSON-formatted string for any predicted attribute relevant for a specific application	-

Constraints -

UML Component/Sequence Diagram



Responsibilities Nissatech

SubscribeToEvents and PublishSituations Interface (Situation Detection Mechanism - Communication & Message Broker)

SubscribeToEvents and PublishSituations Interface

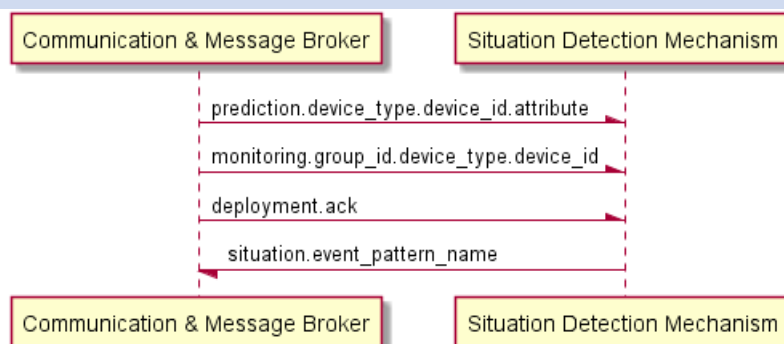
Description	The interface allows the Situation Detection Mechanism to receive monitoring events from cloud and edge devices and publishes situations that may trigger reconfigurations.		
Component providing the interface	Communications & Message Broker		
Consumer components or External Entities	Situation Detection Mechanism		
Type of Interface	AMQP		
State	Asynchronous		
Input data / Output Data	Methods or endpoints of the interface	Parameters of the method	Return Values of the method
	monitoring.<group_id>.<device_type>.<device_id>	-	JSON-formatted string with values of attributes and timestamp
	deployment.ack	-	String (tosca_id)
	prediction.<device_type>.<device_id>.<attribute>	-	JSON-formatted string for any predicted attribute relevant for a specific application
	Other Cloud/Edge resource-Level metrics	-	The indicative list of events provided above can be extended according to PrEstoCloud

adopters
needs
(e.g. Disk
read/writ
e, IPv6
traffic,
etc)

Application-Level metrics	Relevant to application monitoring data (e.g. Response Time, MB of Transcoded Video per second etc.)
situation.<event_pattern_name>	JSON-formatted string with value and timestamp

Constraints	Monitoring sensors (e.g. Netdata Monitoring and Metrics Collector) for producing relevant events under the event topics mentioned above should be deployed (in advance) on Cloud and Edge resources
--------------------	---

UML Component/Sequence Diagram



Responsibilities	ICCS, Nissatech
-------------------------	-----------------

SubscribeToEvents Interface (Mobile Context Analyzer - Communication & Message Broker)

SubscribeToEvents Interface

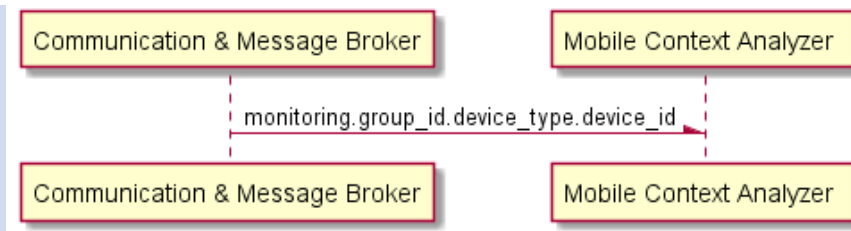
Description	The interface allows the Mobile Context Analyzer to retrieve the monitoring data from cloud and edge devices in order to derive their context.		
Component providing the interface	Communication & Message Broker		
Consumer components or External Entities	Mobile Context Analyzer		
Type of Interface	AMQP		
State	Asynchronous		
Input data / Output Data	Methods or endpoints of the interface	Parameters of the method	Return Values of the method
	monitoring.<group_id>.<device_type>.<device_id>	-	JSON-formatted string with values of attributes and timestamp
	Other Cloud/Edge resource-Level metrics	-	The indicative list of events provided above can be extended according to PrEstoCloud adopters needs (e.g. Disk read/write, IPv6 traffic, etc)
	Application-Level metrics	-	Relevant to application monitoring data (e.g.

Response Time, MB of Transcoded Video per second etc.)

Constraints

The Communication & Message Broker should make sure that even in cases where the connectivity is temporarily lost, the monitoring data is queued and delivered once the connectivity is restored.

UML Component/Sequence Diagram



Responsibilities

ICCS, Nissatech

ProvideEdgeContext & RequestEdgeContext Interface (Mobile Context Analyzer - Resources Adaptation Recommender)

ProvideEdgeContext Interface

Description The interface allows the Resources Adaptation Recommender to retrieve the derived context of edge devices when this is needed.

Component providing the interface Mobile Context Analyzer

Consumer components or External Entities Resources Adaptation Recommender

Type of Interface REST

State Synchronous

Input data / Output Data	Methods endpoints of the interface	or the Parameters of the method	Return Values of the method
	edgedevice/context/battery	UAV_id	JSON-formatted string that reports on the battery percentage expected in the next 5 minutes. Status codes: 200: Successfully transmitted

	the context data 400: Unsuccessfully tried to send the context data
edgedevice/context/attribute	JSON-formatted string for any other contextual attribute relevant for a specific application (e.g. density of edge devices in a certain area)
-	Status codes: 200: Successfully transmitted the context data 400: Unsuccessfully tried to send the context data
Constraints	Relevant monitoring data should have already delivered to the Mobile Context Analyzer
UML Sequence Diagram	<pre> sequenceDiagram participant MCA as Mobile Context Analyzer participant RAR as Resource Adaptation Recommender MCA->>RAR: edgedevice/context/attribute RAR-->>MCA: 200 (success) or 400 (failure) </pre>
Responsibilities	ICCS

AnnounceNewDeploymentRecom Interface (Communication & Message Broker - Application Fragmentation & Deployment Recommender)

AnnounceNewDeploymentRecom Interface

Description	The interface allows the Application Fragmentation & Deployment Recommender to notify the Application Placement & Scheduling Controller that a new type-level TOSCA archive has been stored to the repository.
Component providing the interface	Communication & Message Broker
Consumer components or External Entities	Application Fragmentation & Deployment Recommender
Type of Interface	AMQP
State	Asynchronous

Input data / Output Data	Methods or endpoints of the interface	Parameters of the method	Return Values of the method
--------------------------	---------------------------------------	--------------------------	-----------------------------

deployment.req	JSON-formatted string with values of toasca_id and old_tosca_id	-
----------------	---	---

Constraints

-

UML Sequence Diagram**Responsibilities**

ICCS, Nissatech

Requirements & Code Annotations Interface (Resources Adaptation Recommender, Application Fragmentation & Deployment Recommender - User Interface)

Requirements & Code Annotations Interface

Description	The interface allows the User to send Code annotations and/or annotation files which will guide the partitioning of the application and the deployment of the fragments.
--------------------	--

Component providing the interface	Application Fragmentation & Deployment Recommender, Resources Adaptation Recommender
--	--

Consumer components or External Entities	User Interface
---	----------------

Type of Interface	REST
--------------------------	------

State	Synchronous
--------------	-------------

Input data / Output Data	Methods or endpoints of the interface	Parameters of the method	Return Values of the method
	presto_events/policy_file_uploaded	policy file	Status codes: 200: Successfully transmitted the policy file 400: Unsuccessfully tried to send the policy file

presto_events/
requirements_uploaded

Code_annotati
ons,
annotations_fil
e

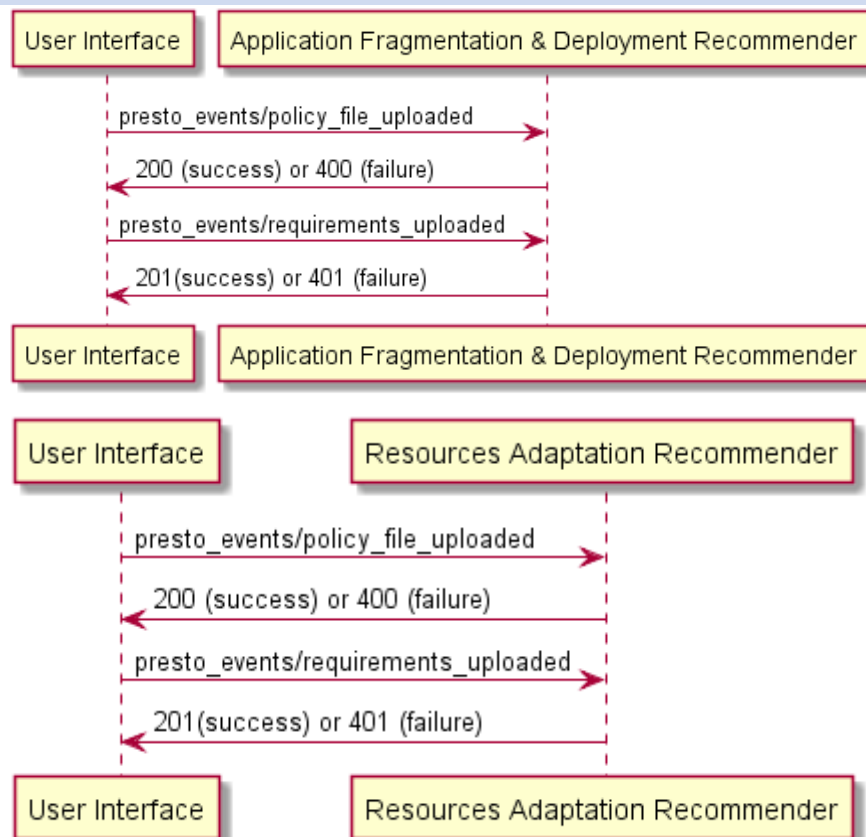
Status codes:

201: Successfully
transmitted the
annotations file /
Code annotations

401: Unsuccessfully
tried to send the
annotations file /
Code annotations

Constraints -

UML Sequence Diagram



Responsibilities

ICCS

SubscribeToEvents & AnnounceReconfigRecom Interface (Resources Adaptation Recommender - Communication & Message Broker)

SubscribeToEvents & AnnounceReconfigRecom Interface

Description

The interface allows the Resources Adaptation Recommender to notify the Application Placement & Scheduling Controller for reconfiguration opportunities (when a new type-level TOSCA archive is uploaded) and receive information for situations requiring adaptation.

Component providing the interface

Communication & Message Broker

Consumer components or External Entities	Resources Adaptation Recommender				
Type Interface	of AMQP				
State	Asynchronous				
Input data / Output Data	Methods or endpoints of the interface	Parameters of the method	Return Values of the method		
	deployment.req	JSON-formatted string with values of toasca_id and old_tosca_id	-		
	situation.<event_pattern_name>	-	JSON-formatted string with value and timestamp		
	prediction.<device_type>.<device_id>.<attribute>	-	JSON-formatted string for any predicted attribute relevant for a specific application		
Constraints	-				
UML Component/Sequence Diagram	<pre>sequenceDiagram participant CMB as Communication & Message Broker participant RA as Resources Adaptation Recommender CMB->>RA: prediction.device_type.device_id.attribute CMB->>RA: situation.event_pattern_name RA->>CMB: deployment.req</pre>				
Responsibilities	ICCS, Nissatech				

Diagrams

Following the details of the interfaces, we provide, below, an updated component diagram (in comparison to the one provided D2.3) that depicts the interfaces used in the PrEstoCloud Meta-Management Layer. We note that the ProvideReconfigTypeLevelTosca and ProvideTypeLevelTosca interfaces refer to the communication to the Cloud and Edge Resources Repository of the Control layer.

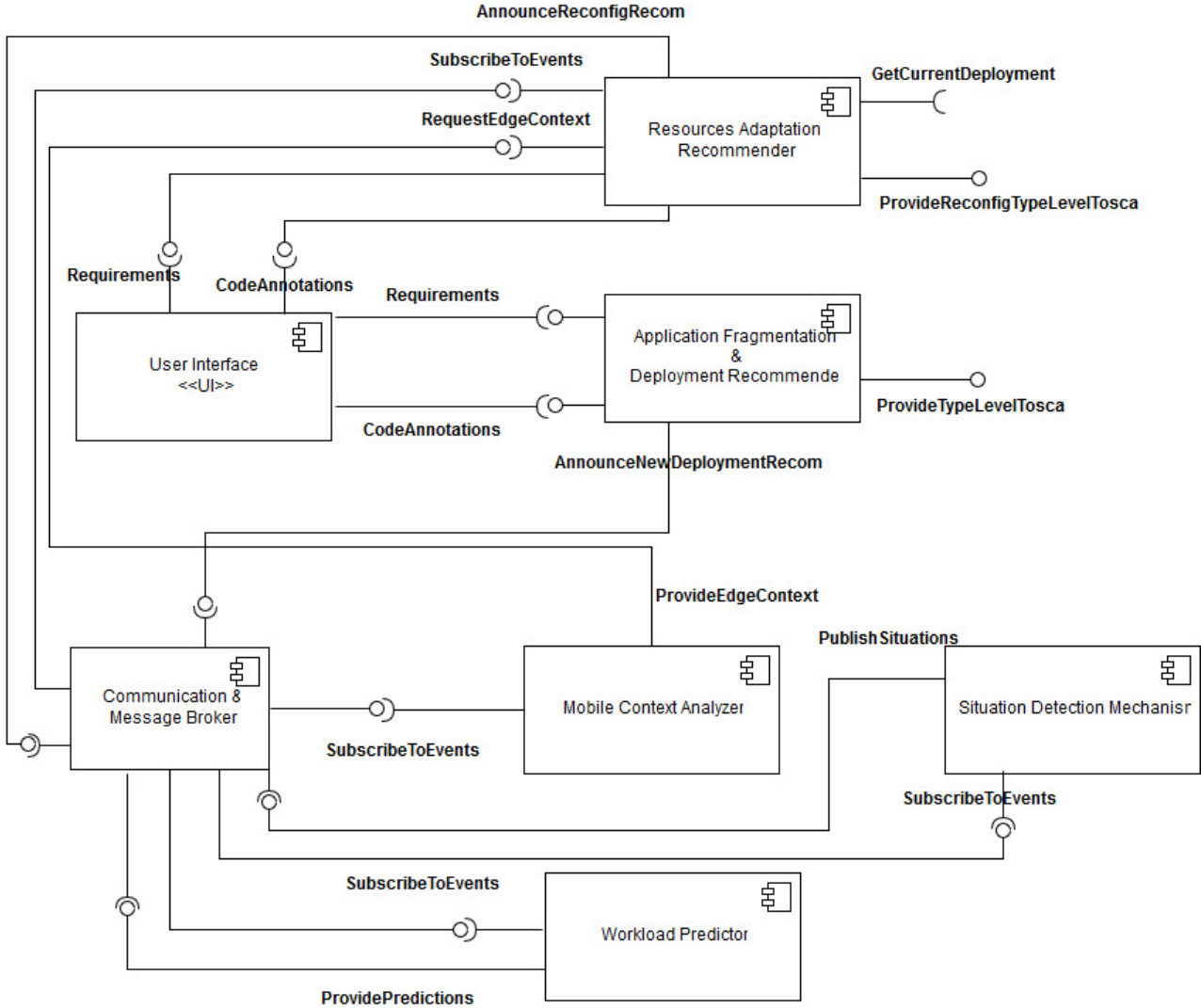


Figure 3: Meta-Management Layer Component diagram

In analogous manner, we have updated the UML sequence diagram provided D2.3 that depicts the communication sequence and message exchange in the PrEstoCloud Meta-Management Layer.

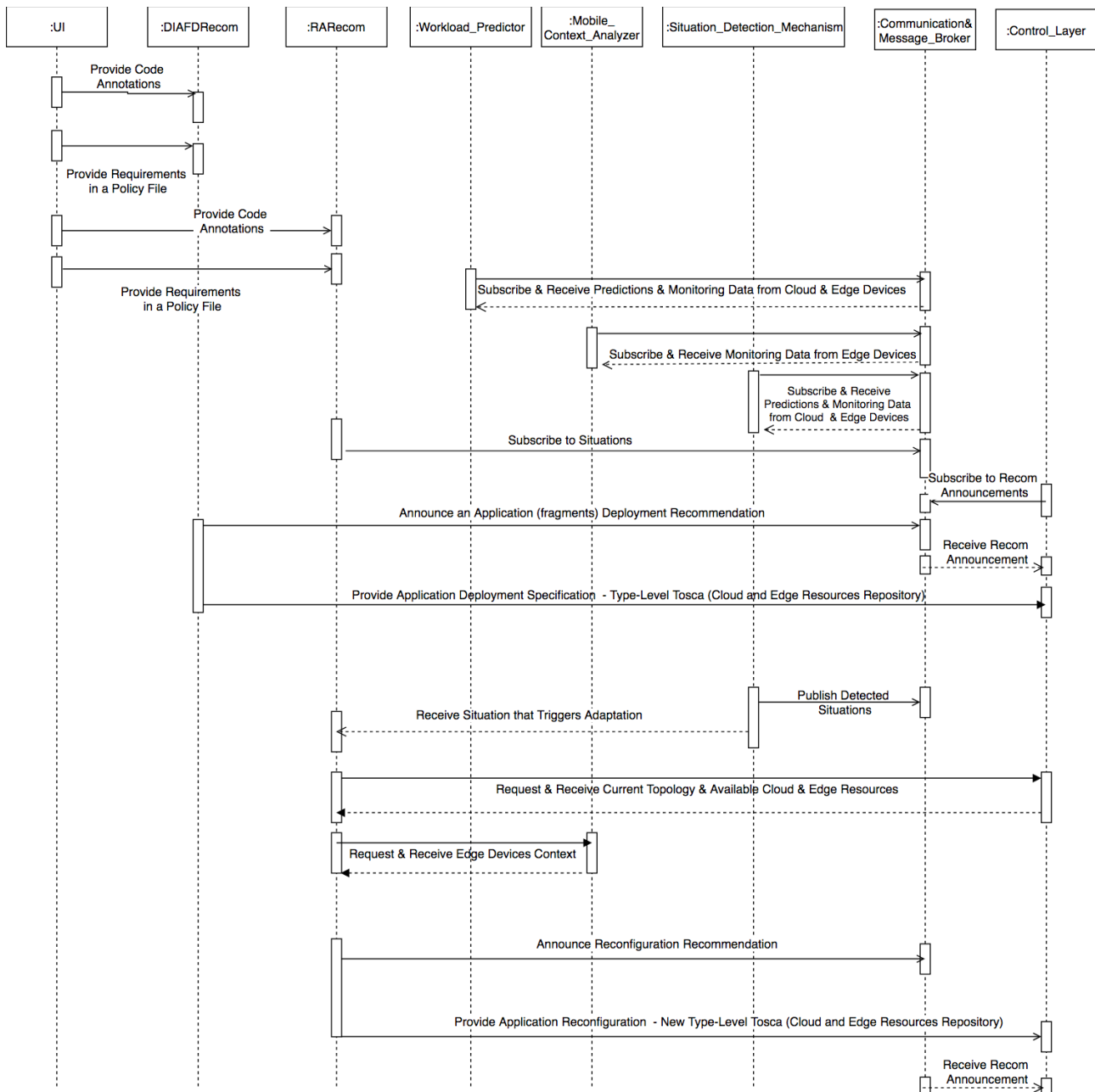


Figure 4: Meta-Management Layer Sequence diagram

3.3.2 Interfaces of the Control Layer Components

NFV Deployment Interface (Autonomic Data Intensive Application Manager - Security Enforcement Mechanism)

NFV Deployment interface

Description The interface allows the Security Enforcement Mechanism to interacts with the Autonomic Data Intensive Application Manager. The interface is used to deploy specific resources (firewall, NIDS, etc.) into PrEstoCloud network overlay in order to take additional security measures.

Component providing the Autonomic Data Intensive Application Manager

interface

Consumer components or External Entities	Security Enforcement Mechanism		
Type of Interface	REST		
State	Synchronous		
Input data / Output Data	Methods endpoints of the interface	Parameters of the method	Return Values of the method
	/ submit/deploySecurityVM	Cloud, region, and type of the security VM to acquire.	IP of the deployed security VM
Constraints	VM templates that contain the appropriate VNF functionalities must exist already		
UML Sequence Diagram	<pre> sequenceDiagram participant SEM as Security Enforcement Mechanism participant ADIAM as Autonomic Data Intensive Application Manager SEM->>ADIAM: deploySecurityVM ADIAM-->>SEM: VM's IP address </pre>		
Responsibilities	ActiveEon, UBITECH		

NewDeploymentorReconfiguration Interface (Edge On/Offloading Server - Autonomic Data Intensive Application Manager)

NewDeploymentorReconfiguration Interface

Description	The interface allows the Autonomic Data Intensive Application Manager to send requests which are new deployment or reconfiguration instructions issued for applications running on edge resources.		
Component providing the interface	Edge On/Offloading Server		
Consumer components or External Entities	Autonomic Data Intensive Application Manager		
Type of Interface	RESTful API (Application Program Interface)		

State	Synchronous			
Input data / Output Data	Methods endpoints of the interface	or Parameters of the method	of the	Return Values of the method
	Java Socket API	Type of request (new deployment or reconfiguration), service name (container name), number of container instances, place of deployment, application-specific inputs, etc.		500: Code of success 501: Code of failure
Constraints	None			
UML Sequence Diagram				
Responsibilities	JSI			

NodeRegistration Interface (Edge On/Offloading Server - On/Offloading Client)

NodeRegistration Interface

Description	The interface allows On/Offloading Clients to send requests for the registration of edge nodes.			
Component providing the interface	Edge On/Offloading Server			
Consumer components or External Entities	On/Offloading Client			
Type of Interface	RESTful API (Application Program Interface)			
State	Synchronous			
Input data / Output Data	Methods endpoints of the interface	or Parameters of the method	of the	Return Values of the method
	Java Socket API	Operating System, CPU Number, Architecture such as x86_64, Total		100: Code of success

memory, Boot Time, Total Disk, etc.	101: Code of failure
-------------------------------------	----------------------

Constraints	None
--------------------	------

UML Sequence Diagram

Responsibilities	JSI
-------------------------	-----

SubscribeDeploymentRequests Interface (Communication & Message Broker - Autonomic Data Intensive Application Manager)

SubscribeDeploymentRequests Interface

Description	The interface allows the Autonomic Data Intensive Application Manager to be notified when a new deployment/reconfiguration must be performed.
--------------------	---

Component providing the interface	Communication & Message Broker
--	--------------------------------

Consumer components or External Entities	Autonomic Data Intensive Application Manager
---	--

Type of Interface	AMQP
--------------------------	------

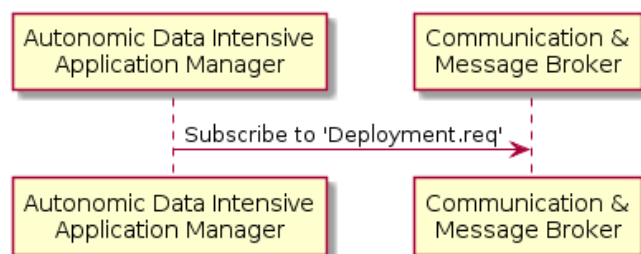
State	Asynchronous
--------------	--------------

Input data / Output Data	Methods or endpoints of the interface	Parameters of the method	Return Values of the method
---------------------------------	--	---------------------------------	------------------------------------

deployment.req	-	JSON-formatted string with value of toasca_id and old_tosca_id
----------------	---	--

Constraints	None
--------------------	------

UML Sequence Diagram



Responsibilities	ActiveEon
-------------------------	-----------

AnnounceTerminatedDeployment Interface (Communication & Message Broker - Autonomic Data Intensive Application Manager)

AnnounceTerminatedDeployment Interface

Description The interface allows the Autonomic Data Intensive Application Manager to notify the Meta-Management layer that the deployment status is now terminated and that therefore a complete TOSCA document has been stored to the repository.

Component providing the interface Communication & Message Broker

Consumer components or External Entities Autonomic Data Intensive Application Manager

Type of Interface AMQP

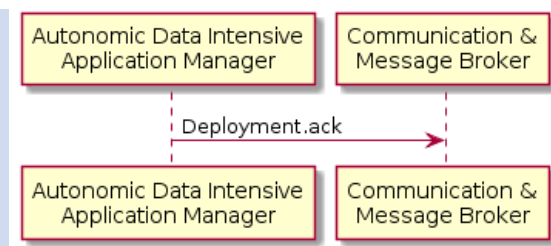
State Asynchronous

Input data / Output Data **Methods or Parameters** of the **Return Values** of the method
endpoints of the interface

deployment.ack String (tosca_id) -

Constraints The TOSCA archive must be uploaded on the repository.

UML Sequence Diagram



Responsibilities ActiveEon

ManagingCloudNodes Interface (Autonomic Resource Manager - Autonomic Data Intensive Application Manager)

ManagingCloudNodes Interface

Description The interface allows the Autonomic Data Intensive Application Manager to interact with the Autonomic Resource Manager. The interface is used to configure and trigger the creation/deletion of cloud resources and to retrieve the status of deploying & acquired resources.

Component Autonomic Resource Manager

providing the interface

Consumer components or External Entities	Autonomic Data Intensive Application Manager				
Type of Interface	Java connector				
State	Synchronous				
Input data / Output Data	Methods or endpoints of the interface	Parameters of the method	Return Values of the method		
	acquireNodes	Cloud, region, and type of the VM/node to acquire.	Identifiers of the deploying nodes		
	removeNodes	Identifiers of the nodes to remove.	Identifiers of the nodes actually removed		
Constraints	None				
UML Sequence Diagram	<pre>sequenceDiagram participant ADI as Autonomic Data Intensive Application Manager participant AR as Autonomic Resource Manager ADI->>AR: acquireNodes AR-->>ADI: nodes identifiers ADI->>AR: removeNodes AR-->>ADI: nodes identifiers </pre>				
Responsibilities	ActiveEon				

VMsRegistration Interface (Autonomic Resource Manager - Cloud Application)

VMsRegistration Interface

Description	The interface allows the deployed cloud resources (VMs) to register themselves with the Autonomic Resource Manager. The interface is used to acquire new cloud resources (through the ProActive agent installed in the VMs) in order to make them accessible by the Autonomic Data Intensive Application Manager (remote tasks execution).
Component	Autonomic Resource Manager

providing the interface

Consumer components or External Entities	Deployed cloud resources (VM)		
Type of Interface	PNP (ProActive Network Protocol) endpoint.		
State	Synchronous		
Input data / Output Data	Methods endpoints of the interface	Parameters of the method	Return Values of the method
	registerNode	PNP url of the node to register	Success: Open bidirectional communication Failure: Send error message and close communication
Constraints	None		
UML Sequence Diagram	<div style="display: flex; justify-content: space-around;"> <div style="text-align: center;"> Success: <pre> sequenceDiagram participant VM1 as Cloud resource (VM) participant ARManager as Autonomic Resource Manager participant VM2 as Cloud resource (VM) participant ARManager2 as Autonomic Resource Manager VM1->>ARManager: registerNode ARManager-->>VM1: node registered VM1->>ARManager2: PNP communication </pre> </div> <div style="text-align: center;"> Failure: <pre> sequenceDiagram participant VM1 as Cloud resource (VM) participant ARManager as Autonomic Resource Manager participant VM2 as Cloud resource (VM) participant ARManager2 as Autonomic Resource Manager VM1->>ARManager: registerNode ARManager-->>VM1: error message </pre> </div> </div>		
Responsibilities	ActiveEon		

PlacementManagment (Application Placement & scheduling controller - Autonomic Data Intensive Application Manager)

PlacementManagment Interface

Description	The interface allows the Autonomic Data Intensive Application Manager to interact with the Application Placement & scheduling controller. The interface is used to specify the model and constraints from the parsed TOSCA files, start the constraints solver, and retrieve the reconfiguration actions to execute in order to reach the desired placement of resources.
Component providing the interface	Application Placement & scheduling controller
Consumer components or External	Autonomic Data Intensive Application Manager

Entities

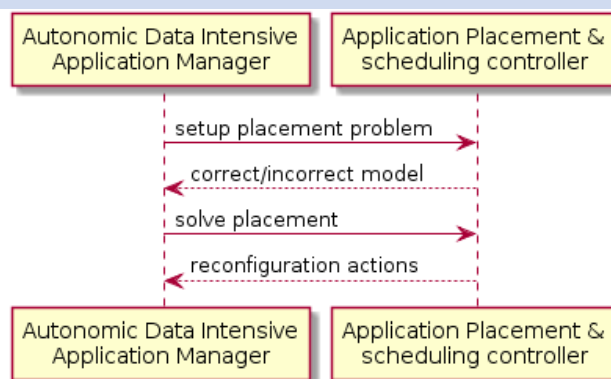
Type of Interface Java connector

State Synchronous

Input data / Output Data **Methods or endpoints of the interface** **Parameters of the method** **Return Values of the method**

setup	Sets of resources and constraints as described in the TOSCA files	A boolean that indicates the correctness of the model
solve	An objective (extracted from TOSCA metadata) represented by a variable that must be maximized or minimized	Sequence of actions to perform in order to reach the desired placement of resources.

Constraints None

UML Sequence Diagram

Responsibilities ActiveEon, CNRS

ProvideAndGetTypeLevelTosca Interface (Cloud & Edge Resources Repository - Autonomic Data Intensive Application Manager, Resources Adaptation Recommender, Application Fragmentation & Deployment Recommender

ProvideAndGetTypeLevelTosca Interface

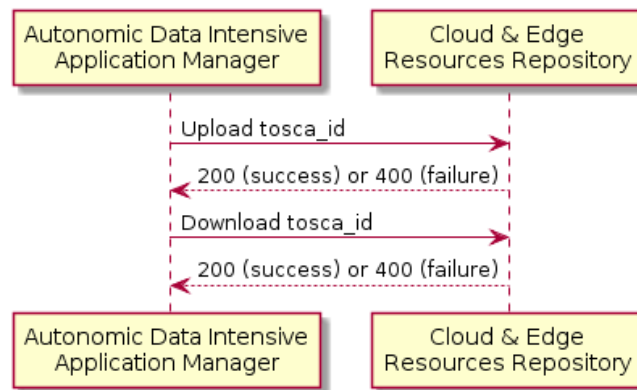
Description The interface is used to share TOSCA files and archives between components of both the Meta-Management Layer and the Control Layer.

Component providing the interface Cloud & Edge Resources Repository

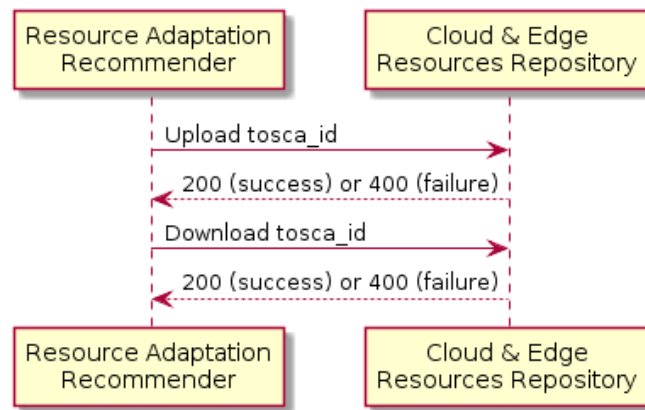
Consumer components or External Entities	Autonomic Data Intensive Application Manager Resources Adaptation Recommender Application Fragmentation & Deployment Recommender		
Type Interface	of	Shared storage based on SSH. The components will use SCP (“secure copy”) to download/upload files from/to the repository through encrypted network connections.	
State	Synchronous		
Input data / Output Data	Methods endpoints of the interface	or Parameters of the method	Return Values of the method
	Download	TOSCA archive name and version to retrieve.	Success: Transfert acknowledgment. Failure: Cause of transfert error.
	upload	TOSCA archive name and version to store.	Success: Transfert acknowledgment. Failure: Cause of transfert error.

Constraints	None
--------------------	------

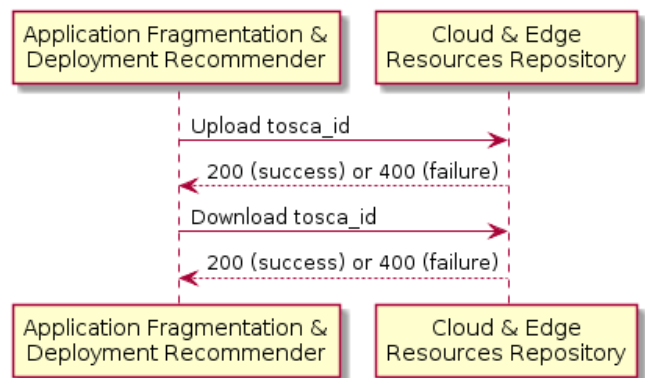
UML Sequence Diagram Autonomic Data Intensive Application Manager:



Resource Adaptation Recommender:



Application Fragmentation & Deployment Recommender:



Responsibilities

ActiveEon, ICCS

SubscribeforSecurityRelatedMonitoringEvents Interface (Security Enforcement Mechanism - Communication & Message Broker)

SubscribeforSecurityRelatedMonitoringEvents Interface

Description

The interface allows the Security Enforcement Mechanism to retrieve monitoring data from cloud and edge devices in order to identify possible situations (e.g.: a DDOS attack) that suggest the enforcement of a security mitigation action.

Component providing the interface

Communication & Message Broker

Consumer components or External Entities

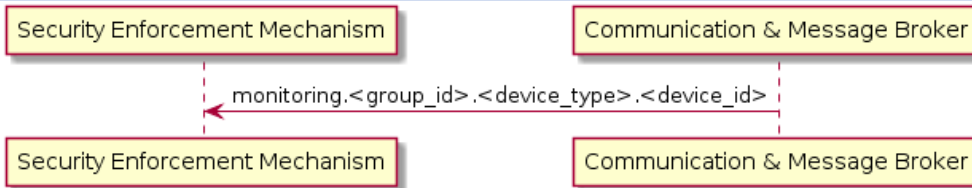
Security Enforcement Mechanism

Type of Interface

AMQP

State

Asynchronous

Input data / Output Data	Methods or endpoints of the interface	Parameters of the method	Return Values of the method
	monitoring.<group_id>.<device_type>.<device_id>	-	JSON-formatted string with Values of attributes and timestamp
Constraints	The Communication & Message Broker should make sure that even in cases where the connectivity is temporarily lost, the monitoring data is queued and delivered once the connectivity is restored.		
UML Component/Sequence Diagram	 <pre> sequenceDiagram participant CMB as Communication & Message Broker participant SEM as Security Enforcement Mechanism CMB->>SEM: monitoring.<group_id>.<device_type>.<device_id> </pre>		
Responsibilities	Ubitech, Nissatech		

Passive Monitoring Interface (Communication & Message Broker - Autonomic Data Intensive Application Manager)

Passive Monitoring Interface

Description	This interface allows the Inter-site Network Virtualization component to publish (passive) monitoring information to the control layer		
Component providing the interface	Communication & Message Broker		
Consumer components or External Entities	Autonomic Data Intensive Application Manager		
Type of Interface	AMQP		
State	Asynchronous		
Input data / Output Data	Methods or endpoints of the interface	Parameters of the method	Return Values of the method

monitoring.<group_id>.<device_type>.
<device_id>

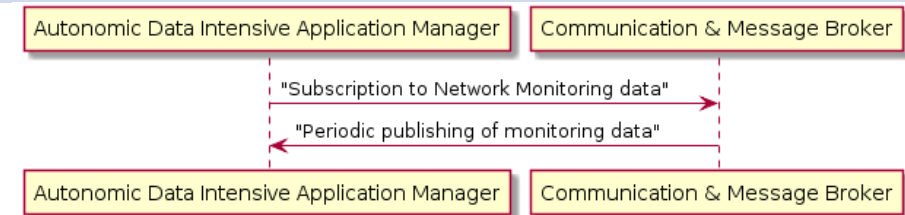
-

JSON-
formatte
d string
with
values of
attribute
s and
timesta
mp

Constraints

None

UML Component/Seq uence Diagram



Responsibilities

CNRS

Diagrams

Following the details of the interfaces, we provide, below, an updated component diagram (in comparison to the one provided D2.3) that depicts the interfaces used in the PrEstoCloud Control Layer.

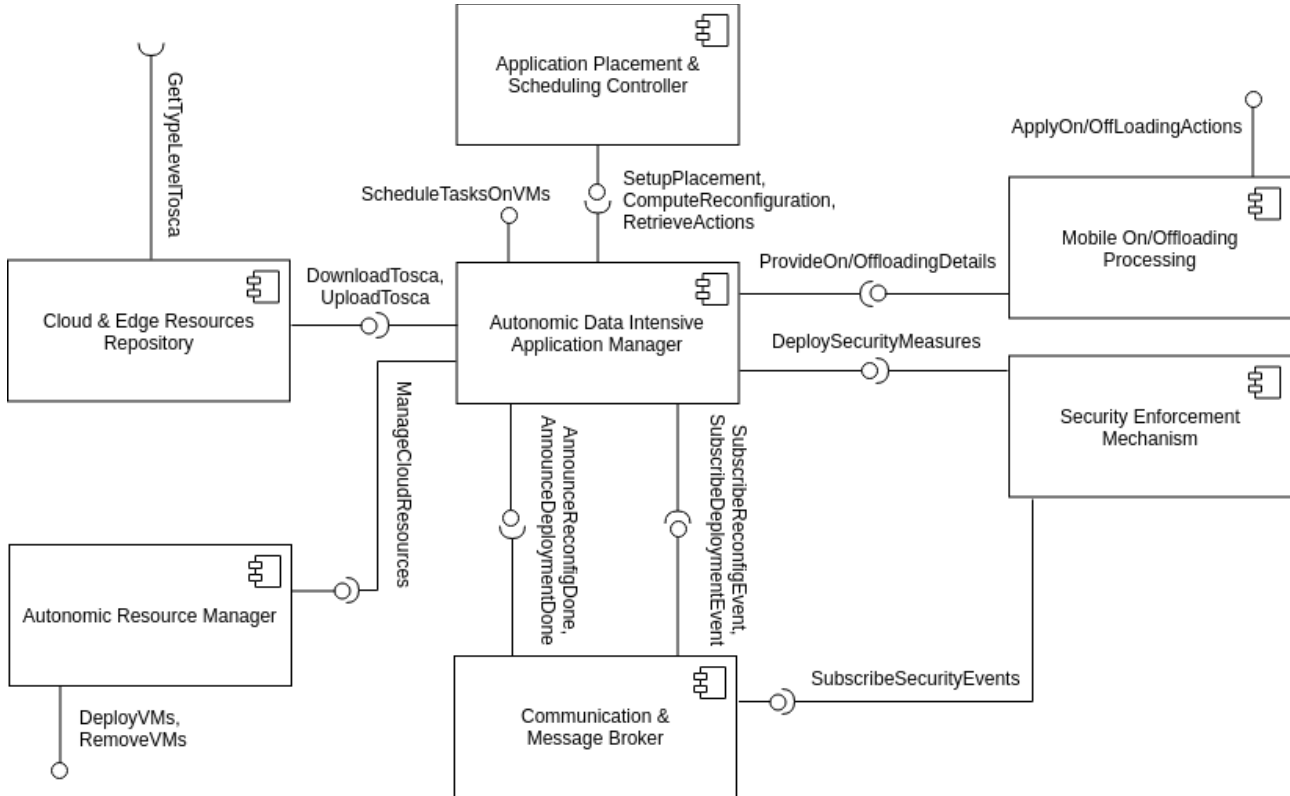


Figure 5: Control Layer Component diagram

In analogous manner, we have updated the UML sequence diagram provided D2.3 that depicts the communication sequence and message exchange in the PrEstoCloud Meta-Management Layer.

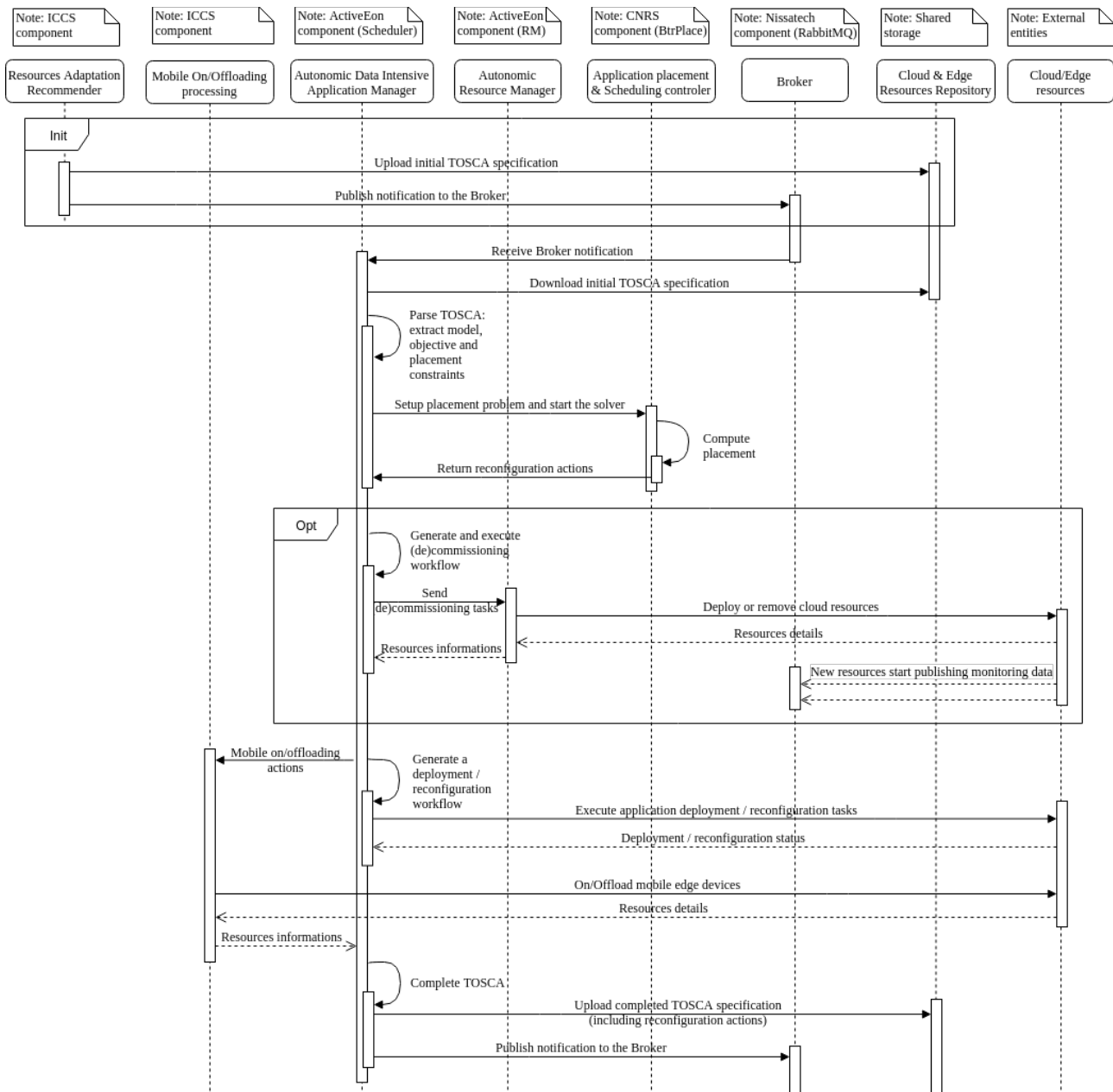


Figure 6: Cloud-Edge Layer Component and Sequence diagram

3.3.3 Interfaces of the Cloud-Edge Communication Layer

Deployment Interface (Inter-site Network Virtualization - Autonomic Data Intensive Application Manager)

Deployment Interface

Description

This interface allows the Autonomic Data Intensive Application Manager to receive the necessary network parameters for configuring a deployment site, and for deploying the VPN-Gateway subcomponent

Component

Inter-site Network Virtualization

providing the interface

Consumer components or External Entities	Autonomic Data Intensive Application Manager			
Type of Interface	Custom workflow within the Autonomic Data Intensive Application Manager			
State	Synchronous			
Input data / Output Data	Methods endpoints of the interface	or Parameters of the method	Return Values of the method	
	Cloud Configuration	IaaS platform	Configuration to be applied to the IaaS platform so as to create an isolated VNet: IP address range, specific configuration commands and parameters for IaaS platform API, topology construct	
	Gateway Initialization		ANSIBLE instructions to properly setup the VPN Gateway VM	
Constraints	None			
UML Sequence Diagram	<pre>sequenceDiagram participant ADI as Autonomic Data Intensive Application Manager participant ISNV as Inter-site Network Virtualization participant IaaS as IaaS Platform API ADI->>ISNV: "Cloud Configuration" ADI->>IaaS: "Apply configuration and comission VM" ADI->>ISNV: "Gateway Initialization"</pre>			
Responsibilities	CNRS, ActiveEon			

Passive Monitoring Interface (Communication & Message Broker - Inter-site Network Virtualization)

Passive Monitoring Interface

Description	This interface allows the Inter-site Network Virtualization component to publish (passive) monitoring information to the control layer			
Component providing the interface	Communication & Message Broker			
Consumer components or External Entities	Inter-site Network Virtualization			
Type of	AMQP			

Interface

State	Asynchronous		
Input data / Output Data	Methods or endpoints of the interface	Parameters of the method	Return Values of the method
	monitoring.<group_id>.<device_type>.<device_id>	JSON-formatted string with values of attributes and timestamp	
Constraints	None		
UML Sequence Diagram	<pre> sequenceDiagram participant IN as Inter-site Network Virtualization participant CM as Communication & Message Broker IN-->>CM: Periodic publishing of monitoring data </pre>		
Responsibilities	CNRS		

ContainerControl Interface(On/Offloading Client - Edge On/Offloading Server)**ContainerControl Interface**

Description	The interface allows the Edge On/Offloading Server to send requests (e.g. start or stop requests) in order to launch or terminate container instances.		
Component providing the interface	On/Offloading Client		
Consumer components or External Entities	Edge On/Offloading Server		
Type of Interface	API (Application Program Interface)		
State	Synchronous		
Input data / Output Data	Methods or endpoints of the interface	Parameters of the method	Return Values of the method
	Java Socket API	start or stop request for container, IP to run	200: Code of success in start a

the container, number of ports to be defined, application-specific inputs, etc.

container instance
201: Code of failure in start a container instance

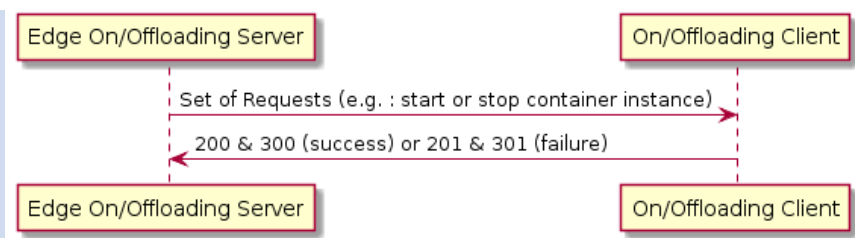
300: Code of success in stop a container instance

301: Code of failure in stop a container instance

Constraints

The application may have different inputs depending on the service type. These inputs are defined by application developers determined as parameters of the API. Moreover, the order of parameters of the method is important.

UML Sequence Diagram



Responsibilities

JSI

Appendix B contains more details about this part.

Diagrams

The following figure shows the sequence diagram for two procedures: (i) Registration of edge node and (ii) New deployment or reconfiguration of the application. These two procedures were explained before through the APIs exposed by both the Edge On/Offloading Server and the On/Offloading Client. The registration of edge node starts from the On/Offloading Client, and the new deployment or reconfiguration of the application starts from the Autonomic Data Intensive Application Manager.

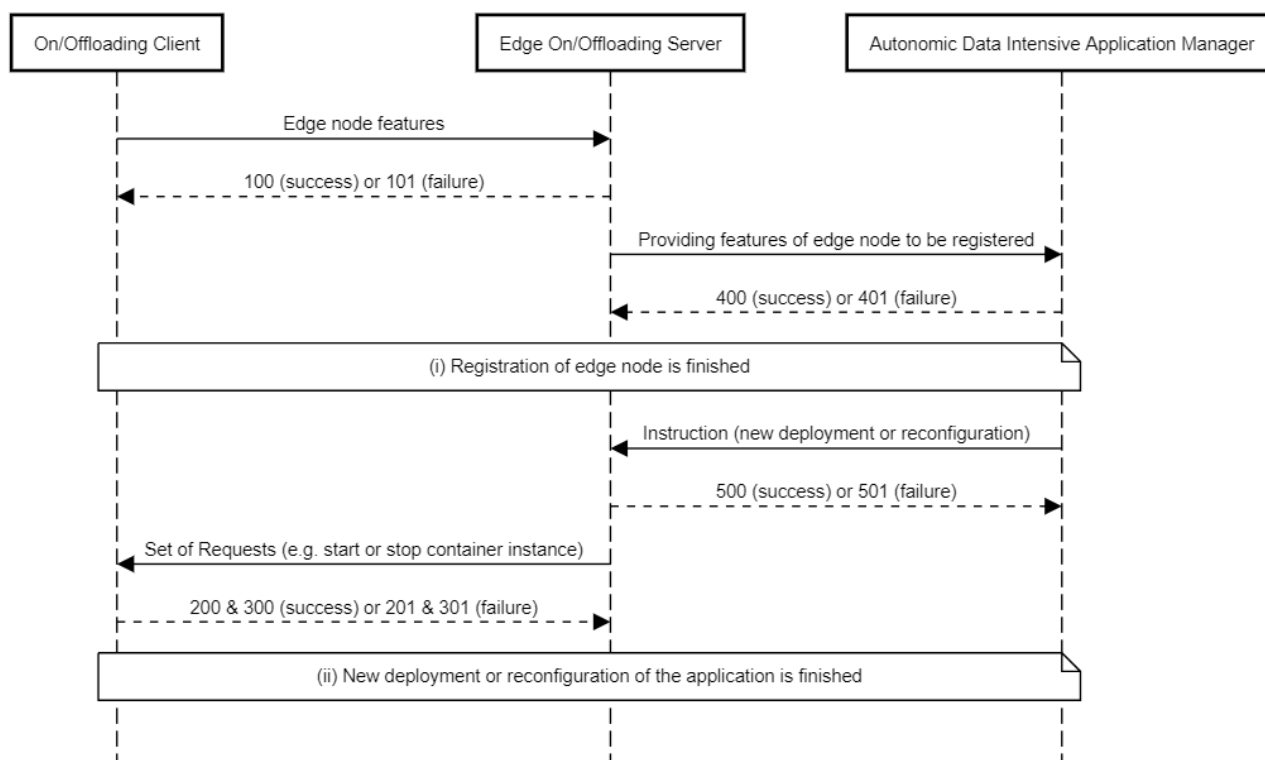


Figure 7: Sequence diagram for (i) Registration of edge node and (ii) New deployment or reconfiguration of the application.

3.3.4 Communication through the Broker

The nature and role of the Broker is described in D3.1 Communication Broker: Iteration 1.

Appendix A contains more details about the Broker (not presented in D3.1), which are outcomes from an intensive work on an easier install and configuration of the broker.

For each component that will communicate through the Broker we need:

- Topic (name)
- Publisher and subscriber
- Used protocols
- JSON (or similar) description of the format
- Description of the messages to be sent
- Any information/constraints related to the data transfer, like frequency of sending (msec) or content (KB/event)

Below we show is an indicative list of topics that may be extended as the integration of PrestoCloud components progresses. Also, parameters and units in payload can be changed.

monitoring.<group_id>.<device_type>.<device_id>

Topic Name	monitoring.<group_id>.<device_type>.<device_id>	
Description	This is a topic which describes the NetData monitoring of a cloud/edge node.	
Component publishing under this topic and protocol	Inter-Site Network Virtualization, Cloud VM's	AMQP
	Edge devices	MQTT
Components subscribing for this topic and protocol	Workload Predictor, Situation Detection Mechanism, Mobile Context Analyzer, Security Enforcement Mechanism, Autonomic Data Intensive Application Manager	
Payload structure	disk-reads: <value> disk-writes: <value> network-received: <value> network-sent: <value> cpu-system: <value> cpu-user: <value> cpu-iowait: <value> ram-free: <value> timestamp: <value>	
Example	disk-reads: 0.2 disk-writes: 0.2 network-received: 1 network-sent: 1 cpu-system: 10 cpu-user: 30 cpu-iowait: 2 ram-free: 2 timestamp: 1527023456	

prediction.<device_type>.<device_id>.<attribute>

Topic Name **prediction.<device_type>.<device_id>.disk-reads**

Description	This is a topic which describes the prediction of average read MB/s from disk in that moment from a cloud/edge node.	
Component publishing under this topic and protocol	Workload Predictor	AMQP
Components subscribing for this topic and protocol	Situation Detection Mechanism, Resources Adaptation Recommender	
Payload structure	disk-reads: <value> timestamp: <value>	
Example	disk-reads: 0.2 timestamp: 1527023456	

Topic Name	prediction.<device_type>.<device_id>.disk-writes	
Description	This is a topic which describes the prediction of average written MB/s to disk in that moment on a cloud/edge node.	
Component publishing under this topic and protocol	Workload Predictor	AMQP
Components subscribing for this topic and protocol	Situation Detection Mechanism, Resources Adaptation Recommender	
Payload structure	disk-writes: <value> timestamp: <value>	
Example	disk-writes: 0.2 timestamp: 1527023456	

Topic Name	prediction.<device_type>.<device_id>.network-received	
Description	This is a topic which describes the prediction of received Mbit/s on the network in that moment on a cloud/edge node.	
Component publishing under this topic and protocol	Workload Predictor	AMQP

protocol

Components subscribing for this topic and protocol	Situation Detection Mechanism, Resources Adaptation Recommender	AMQP
Payload structure	network-received: <value> timestamp: <value>	
Example	network-received: 1 timestamp: 1527023456	

Topic Name	prediction.<device_type>.<device_id>.network-sent	
Description	This is a topic which describes the prediction of sent Mbit/s to the network in that moment from a cloud/edge node.	
Component publishing under this topic and protocol	Workload Predictor	AMQP
Components subscribing for this topic and protocol	Situation Detection Mechanism, Resources Adaptation Recommender	AMQP
Payload structure	network-sent: <value> timestamp: <value>	
Example	network-sent: 1 timestamp: 1527023456	

Topic Name	prediction.<device_type>.<device_id>.cpu-system	
Description	This is a topic which describes the prediction of percentage usage of CPU by system in that moment on a cloud/edge node.	
Component publishing under this topic and protocol	Workload Predictor	AMQP
Components subscribing for this topic and protocol	Situation Detection Mechanism, Resources Adaptation Recommender	AMQP

Payload structure cpu-system: <value>
timestamp: <value>

Example cpu-system: 10
timestamp: 1527023456

Topic Name **prediction.<device_type>.<device_id>.cpu-user**

Description This is a topic which describes the prediction of percentage usage of CPU by user in that moment on a cloud/edge node.

Component publishing under this topic and protocol Workload Predictor AMQP

Components subscribing for this topic and protocol Situation Detection Mechanism, Resources Adaptation Recommender AMQP

Payload structure cpu-user: <value>
timestamp: <value>

Example cpu-user: 30
timestamp: 1527023456

Topic Name **prediction.<device_type>.<device_id>.cpu-iowait**

Description This is a topic which describes the prediction of percentage usage of CPU by I/O activities in that moment on a cloud/edge node.

Component publishing under this topic and protocol Workload Predictor AMQP

Components subscribing for this topic and protocol Situation Detection Mechanism, Resources Adaptation Recommender AMQP

Payload structure cpu-iowait: <value>
timestamp: <value>

Example cpu-iowait: 2
timestamp: 1527023456

Topic Name	prediction.<device_type>.<device_id>.ram-free	
Description	This is a topic which describes the prediction of free RAM (GB) in that moment on a cloud/edge node.	
Component publishing under this topic and protocol	Workload Predictor	AMQP
Components subscribing for this topic and protocol	Situation Detection Mechanism, Resources Adaptation Recommender	
Payload structure	ram-free: <value> timestamp: <value>	
Example	ram-free: 2 timestamp: 1527023456	

situation.<event_pattern_name topic>

Topic Name	situation.high_cpu	
Description	This is a situation that denotes high CPU usage (e.g. >80%) observed on a cloud or edge resource	
Component publishing under this topic and protocol	Situation Detection Mechanism	AMQP
Components subscribing for this topic and protocol	Resource Adaptation Recommender	
Payload structure	device_id: <value> metric_value: <value> timestamp: <value>	
Example	device_id: UAV_12345678 metric_value: 82 timestamp: 1526997880	

Topic Name	situation.high_ram	
Description	This is a situation that denotes high RAM usage (e.g. >80%) observed on a cloud or edge resource	
Component publishing under this topic and protocol	Situation Detection Mechanism	AMQP
Components subscribing for this topic and protocol	Resource Adaptation Recommender	AMQP
Payload structure	device_id: <value> metric_value: <value> timestamp: <value>	
Example	device_id: UAV_12345678 metric_value: 90 timestamp: 1527003160	

Topic Name	situation.low_network	
Description	This is a situation that denotes high Network usage (e.g. <10%) observed on a cloud or edge resource	
Component publishing under this topic and protocol	Situation Detection Mechanism	AMQP
Components subscribing for this topic and protocol	Resource Adaptation Recommender	AMQP
Payload structure	device_id: <value> metric_value: <value> timestamp: <value>	
Example	device_id: UAV_12453853 metric_value: 02 timestamp: 1527004160	

deployment.req

Topic Name	deployment.req
Description	This is a topic which informs the Autonomic Placement & Scheduling Controller or any other interested component that a new type-level TOSCA file has been uploaded to the repository and can be used to implement the instance-level TOSCA.
Component publishing under this topic and protocol	Application Fragmentation & Deployment AMQP Recommender, Resources Adaptation Recommender
Components subscribing for this topic and protocol	Autonomic Data Intensive Application Manager AMQP
Payload structure	tosca_id: <value> old_tosca_id: <value> timestamp: <value>
Example	tosca_id: type_level-0000002.yaml old_tosca_id: type_level-0000001.yaml timestamp: 1527009252

deployment.ack

Topic Name	deployment.ack
Description	This is a topic which informs the Resources Adaptation Recommender or any other interested component that the deployment has been successfully carried out according to the instance-level TOSCA stored in the repository.
Component publishing under this topic and protocol	Autonomic Data Intensive Application Manager AMQP
Components subscribing for this topic and protocol	Situation Detection Mechanism AMQP
Payload structure	deployment_id: <value> timestamp: <value>

Example

```
deployment_id: instance_level-0000001.yaml  
timestamp: 1527009252
```


4. Requirements Refinement

In deliverable D2.3 we provided a detailed mapping between the requirements identified in D2.2 and the conceptual architecture.

Since the list and the position of components in the updated architecture is the same, there was not a huge need for an update of the initial mapping.

However, there were a few changes related to the requirements:

1. Since the component Workload predictor has become the central component for processing data from the Broker, it will be related also to processing monitoring data and its management (requirement FR-4).
2. In addition, Workload predictor can be used for the set-up of the analytics (real-time and batch processing, requirement FR-65).
3. An issue that must be resolved is GDPR (FR-13), so that we changed the priority for this requirement in the “Must have”.

These changes have also an impact on the prioritization of the requirements. The following figure illustrates the most important changes.

FR-2	Ability to set and attach metadata to resources based on a common description model	Should have	Yes
FR-4	Ability to centralize the monitoring in a common view or place	Must have	Yes
FR-10	Uniform resource interaction (harmonized API for different cloud providers)	Should have	Yes
FR-12	Platform enables the establishment of secure inter-site channels	Should have	Yes
FR-13	The ability to comply with the new GDPR (General Data Protection Regulation) regarding the protection of personal data and personal sensitive data.	Must have	No
FR-15	Ability to modify network rules to maintain regulatory compliance	Should have	Yes
FR-28	Ability to receive recommendations on initial data placement	Should have	Yes
FR-30	Ability to enact (e.g. call an API) the implementation of the initial data placement	Should have	Yes
FR-32	Ability to express runtime data migration constraints	Should have	Yes
FR-34	Ability to receive recommendations on data migration	Should have	Yes
FR-36	Ability to enact (e.g. call an API) the implementation of data migration	Should have	Yes
FR-48	Ability to express runtime constraints at the level of a) individual Data Intensive Applications, b) entire Data Intensive Service Graph	Should have	No
FR-52	Retrieve desired set of metadata from common PrEstoCloud model	Should have	No
FR-54	Ability to manage (export/import) custom workflows	Should have	Yes
FR-64	Ability to adjust fragmentation and deployment	Should have	Yes
FR-65	Ability to configure methods for data processing (real-time, batch)	Must have	Yes
FR-68	Ability to express and compose a directed acyclic graph (hereinafter data intensive service graph - DISG) that consists of data-intensive-apps (hereinafter DIA) that collaborate each other	Should have	No

Figure 8: Update in the list of requirements

Further technical development will take into account these changes.

The most important is the compliance with GDPR. In the deliverable D7.11 we have started already the analysis of the use cases from the GDPR point of view and these findings will be used in the technical development.

5. Technical Integration and Planning

5.1 Introduction

In order to implement the use cases of the project, separate installations will be provided for each pilot. This approach is followed in order to optimize the platform setup to each use case needs, to enhance security, and to reduce network traffic that a centralized PrEstoCloud installation would add thus improving the latency and QoS of the pilot services.

For this reason, we provide a high-level mapping of the components of the architecture that will be deployed and integrated in each of the use cases. This mapping has been provided in Appendix C – Technical integration in use cases. In the same time, in order to make the initial integration and to be able to test the platform, we have also created a sandbox infrastructure that is used for all component and integration tests so far. The same infrastructure and setup of the platform can allow us to create prototypes of the platform that can be used for demonstration purposes and also as initial setup for the use cases.

As explained in deliverable D2.3, PrEstoCloud’s conceptual architecture follows, on a higher abstraction level, the self adaptivity pattern: **Monitor Analyze Plan Execute** Knowledge [IBM05] (MAPE-K model; so called “architectural blueprint for autonomic computing”, introduced by IBM). And as it was shown in the architecture diagram of Figure 1 and in the explained integration approach, PrEstoCloud Platform consists of different components responsible for separate jobs. Using the MAPE-K mapping, PrEstoCloud includes components for a) monitoring the infrastructure and application, b) process and analyse the monitored data, c) components that are responsible for planning and making decisions on the adaptation and d) components responsible for the execution. Finally, e) specific parts of the platform are holding the knowledge in terms. All these components are integrated both using direct communications between components (Star architecture) and asynchronous, loosely-coupled integration using a common message broker when it is needed.

In order to actually make the integration in a platform like PrEstoCloud that is composed by many interconnecting components that are provided by different partners, analysing only the interfaces among components is not enough, but further agreement on technical and non-technical decisions has to be made. In this section we will present how the integration actually takes place, at deployment, interfaces, code and even knowledge perspectives.

Table 1. PrEstoCloud integration mechanisms

Integration Perspectives on PrEstoCloud	
At Deployment Level	Configuration of components’ deployment using Docker compose Dedicated container registries using GitLab Sandbox for integration testing
At Interfaces Level	Documentation of Interfaces
At Code Level	Dedicated code repositories using GitLab
At Knowledge Level	Usage of Skype and Work Package

specific lists

Dedicated folder for collaboration on the shared repository of consortium and GitLab for issues management

5.2 Integration at Deployment Level

For the technical integration in PrEstoCloud we need many different components to be deployed and communicate using dedicated interfaces or the common message broker. For achieving this we are using Docker Compose¹ files. This approach is followed in order to allow easy installation and replication of the whole PrEstoCloud platform, by providing a central way of defining and configuring services, while also benefiting from the portability aspects of the Operating System virtualization (containers).

5.2.1 Using Docker Compose for Integration

Docker Compose is a tool for defining and running multi-container Docker applications, based on a YAML file that is used to configure application's services. Then, with a single command, all services can be created and started using the configuration file. Following the approach doesn't mean that all components of PrEstoCloud should be containerized and deployed using Docker. Some services may still be provided from a centralized point or through dedicated physical or virtual machines. What is managed through Docker compose configuration file (and by respecting some development guidelines) is the way that all components are possible to be deployed and communicate.

In order to make the whole integration flow to work based on Docker Compose in an autonomous and continuous way for PrEstoCloud, we will try to create Docker based container images for the components developed, whenever this possible. In comparison to virtual machine that needs to include infrastructure configuration and the whole OS, the containers image is a lightweight, stand-alone, executable package that includes everything needed to run a piece of software, including the code, a runtime, libraries, environment variables, and configuration files. A container is a runtime instance of an image—what the image becomes in memory when actually executed. In comparison to a Virtual Machine (VM) that is completely isolated, a container is partially isolated from the host environment, as it uses the kernel calls and commands of the host OS, but accessing host files and ports is possible only if configured to do so.

Images are created using Docker and are possible to be configured using Dockerfile. A Dockerfile contains instructions on how to create the desired image based on pre-existing images. More information regarding the creation process is provided in section that follows.

The pre-existing images can be stored and retrieved from image repositories called Docker registries. Such a Docker Registry is used for PrEstoCloud and is a stateless, highly scalable server-side application that allows storing and distributing Docker images. It works similar to Git, as collaborators can login and then push or pull the images that they or other partners are creating.

The configuration of the multiple available components of the platform that are communicating through the interfaces can be helped with Docker compose files,

¹ <https://docs.docker.com/compose>

through the usage of Environmental Variables for configuration. Most important parameters that are usually needed for this action are service urls, ports or any other information needed for the acknowledgment and configuration between services. With environmental variables existing in each Docker compose based deployment, the service developer shall utilize the available variables at code level in order to avoid the hardcoding of parameters that make applications difficult to deploy and, in the end, develop and test a project with many different partners in remote locations, like PrEstoCloud.

First step for collaborative working was that each partner needs to join the PrEstoCloud project page in GitLab² that will be used for the hosting of private code and container repositories, as well tools for the support of CI activities. For hosting integration related material of the whole framework, as docker-compose.yml files or the needed docker images³, a project called framework has been created.

5.2.2 Sandbox-based approach

In order to make the initial integration and to be able to test the platform, we have also created a sandbox infrastructure that is used for all component and integration tests so far. The same infrastructure and setup of the platform can allow us to create prototypes of the platform that can be used for demonstration purposes and also as initial setup for the use cases.

5.3 Integration at Interface Level

For the integration of the components and the better coordination of the development of the interfaces, lengthy discussions have been made. Each partner is responsible to provide the appropriate documentation for the interface usage and this in order to allow the consumer of the interface to use it properly and inform when changes are introduced in order to edit/adapt to changes.

The current version of the interfaces needed for the platform integration has been agreed upon discussions of the partners (at whole consortium and mostly at bilateral level) and are documented in this deliverable This mapping has been provided in Error: Reference source not found.

Asynchronous Operations

For interfaces that need asynchronous operation mode for their communication, the common message broker is used. The publish/subscribe (pub/sub) messaging pattern is realized using destinations known as topics. Publishers send messages to the topic and subscribers register to receive messages from the topic. Any messages sent to the topic are automatically delivered to all subscribers. For more details the reader can refer on section 3.3.4.

5.4 Code Level Integration

In the cases that multiple partners need to work on the same components, code level integration is supported with a code repository that is available for all partners that need to work together or to store their component's code safely. The source code repositories are available at: <https://gitlab.com/prestocloud-project>.

² <https://gitlab.com/prestocloud-project>

³ registry.gitlab.com/prestocloud-project/framework

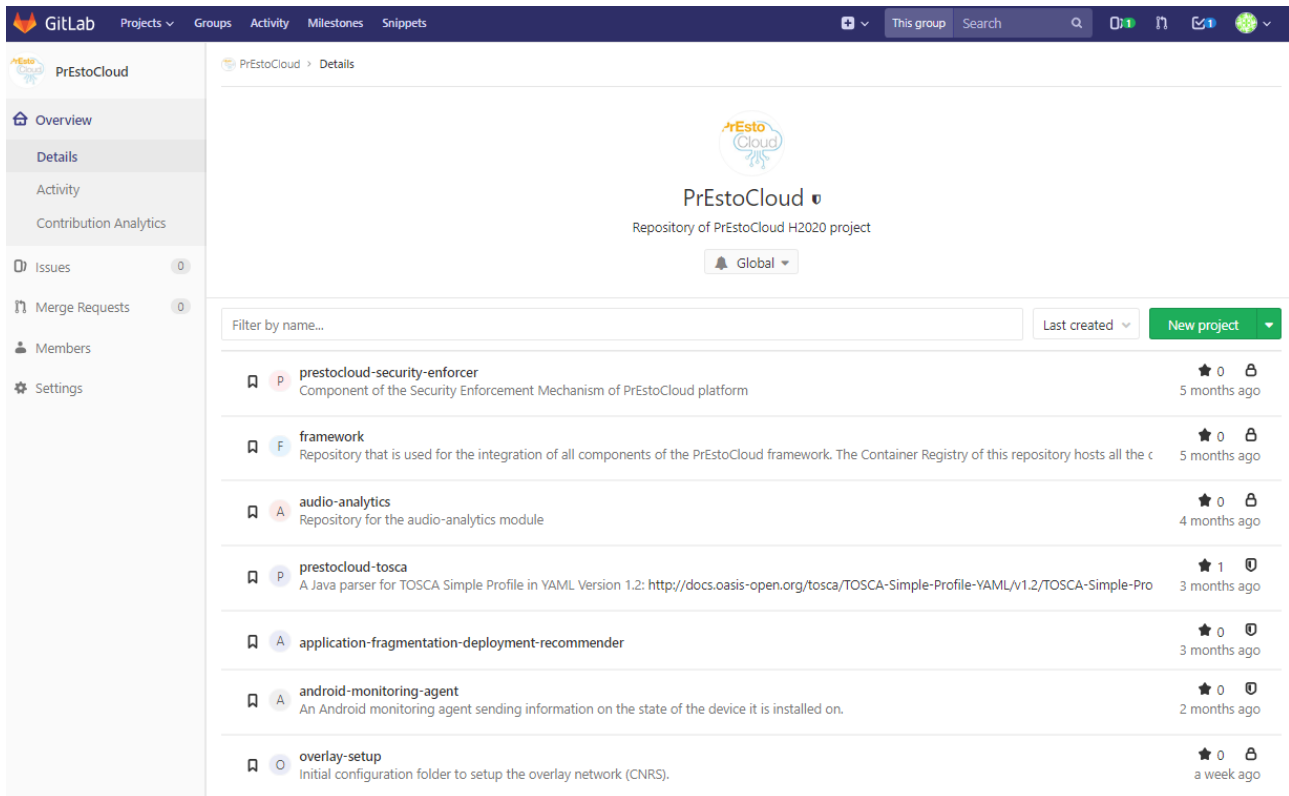


Figure 9: PrEstoCloud project group in GitLab

5.5. Knowledge level integration

The last part to cover for the technical integration is the need for a collaboration mechanism that allows partners that are working in distributed manner to collaborate. This important parameter is often performed without setting strict rules, but in the case of PrEstoCloud we tried to collect the shared knowledge in order to ease the development and integration. Towards these directions the following steps have been performed. Initially a skype group for PrEstoCloud has been created, and dedicated channels for each work package have been created. Then, in order to achieve integration planning goals, a document collecting all the details of the technical integration, together with instructions and examples has been created and uploaded to the common repository of the project. This document will be constantly updated when needed. Finally, with GitLab it is also possible to create wiki pages for each of the component in order to allow partners developing a component to provide the needed information and instructions in a structured format. In the GitLab of the project issues, enhancements and suggestions are stored by the partners, even per component, thus allow the proper tracking of the development progress and status.

5.6 Integration Planning

Following the methodology presented above, internal artefacts produced by the technical work packages (WP3-WP5) will be continuously integrated. In the same time, we plan three major releases of the PrEstoCloud integrated platform for M20, M31 and M36. The aim is to integrate the components into PrEstoCloud platform and instantiate it for each use cases. Feedback and lessons learnt will be fed back to this integration workpackage from WP7. As also described in deliverable D7.2, in PrEstoCloud we will perform two cycles of technical and user evaluations during the project period, so the development of the platform and its evaluation will be performed in parallel. The results of the validation and the evaluation need to be fed back into the development cycle, improving the user experience and detecting open issues. Therefore, after the

evaluation will be performed, the resulting issues need to be discussed by the consortium in order to be updated accordingly.

The first major release of PrEstoCloud will include the integration of the basic components developed in the technical work packages. The goal is to present basic capabilities integrated until M20. During the second iteration, the integrated version of all the stable versions of the PrEstoCloud components developed in the technical work packages will be released (M31). Feedback from the pilot-based evaluation (WP7 at M26 and M34) will be also taken into account. Based on the last evaluation iteration (M34), further enhancements or modifications will be provided in the final release of the PrEstoCloud platform (M36). However, smaller iterations with changes on the architectural design and the development will be used through the project duration.

5.6.1 Multi-Iteration/Release Plan

Based on the plan of having to three releases of PrEstoCloud framework, the consortium had to make decisions regarding the specific functionalities that will be supported on each release, in order to coordinate the development, iteration and testing process. Meanwhile the integration between components has started to allow the iterative implementation of the platform and to assure the delivery of the first version in M20, in order to collect the feedback from the use cases by M26. **It should be emphasized that even earlier that this first version,** by M18 a first set of component and functionalities will be demonstrated to the use case partners in order to acquire preliminary feedback. Then, on M31 an updated version of the platform will be provided for extensive testing until M34. The final version of the integrated platform will be delivered on M36 and will include all the individual functionalities per components and extend the integration of the first version to support the functionalities provided in the final version of the components.

To properly support this during development, we have created on the project GitLab dedicated milestones in order to be able to plan and track the advancements on all developed components and the integrated platform.

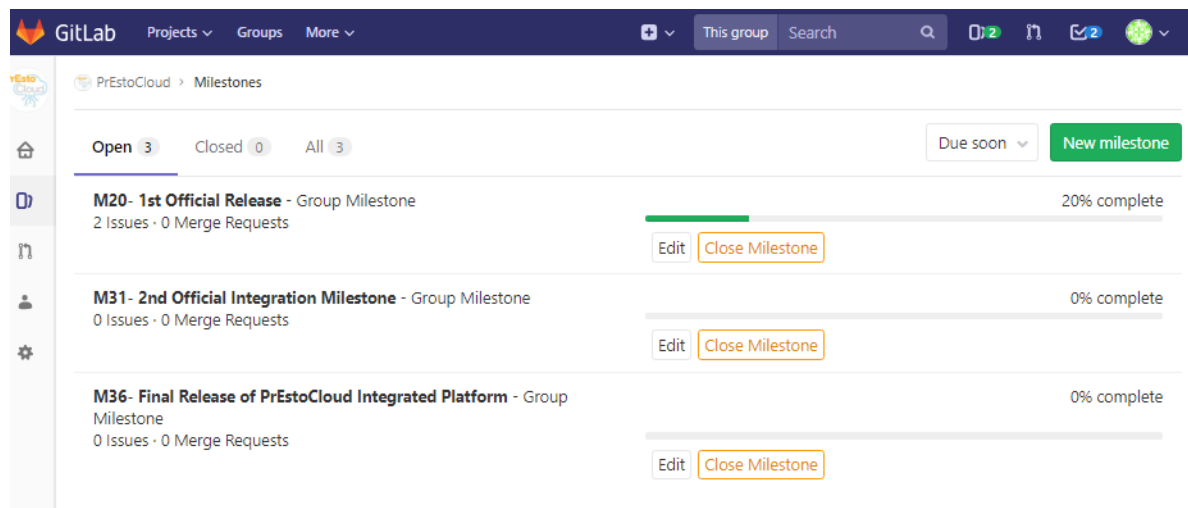


Figure 10: PrEstoCloud Milestones as part of the development and integration plan

1st Platform Release

The first release of PrEstoCloud is due on M20. The goal is to complete the first cycle of integration. The actual deadline is dictated by the deliverable D6.2 that will document the platform status and provide installation and usage instructions. Also, in Deliverable D6.2 a planning for the status of the components for the next releases will

also be provided. As already stated, the first release will focus on the integration of the basic components of the platform.

2nd Platform Release

As the development efforts for the workpackages WP3 to WP5 conclude by M30, the second platform release will be provided on M31. It will include the full implementation of the components in order to allow full testing and evaluation at the perspective of the use cases until M34.

Final Version of the Platform

The final version of PrEstoCloud Framework will be delivered at the end of the project, by M36. This version will be fully integrated and the documentation of this final version of PrEstoCloud framework will be delivered, as part of deliverable D6.4. In this final version of the platform, the feedback from the second evaluation period will be incorporated in order to improve the platform.

6. Conclusions

This deliverable is the continuation of the work reported in the deliverable D2.3, where the initial architecture and the interfaces between the components were presented. The work was focused on the analysis and refinement of every possible interaction in the system architecture. Special attention was given to the communication with the Broker (message-oriented middleware).

The architecture represents a distributed event driven architecture, enabling modern Edge-Cloud processing pipelines. The main goal of the deliverable was to document the details of the functionalities and the communication between all components, driven by the scenarios defined in the deliverable D2.3.

One of the main efforts was in defining a proper structure of the topics that will be used as the structure for the exchange of the data over the broker (in a pub-sub oriented way). This task was especially complex due to a need for defining a minimal but complete set of topics that will cover different monitoring (from the edge and cloud infrastructure) and data processing requirements.

The main outcome is the detailed architecture that serves as the basis for the development of the integrated system. All interfaces are clearly defined and documented. Moreover, this document should be seen as an evolution of D2.3, whereas the information related to the description of the components is provided in D2.3 in a complete form.

The requirements are refined based on the progress in the development of the planned technology. One of the most important open issues is the compliance with GDPR.

This work will be used in the context of the work package WP6, which is related to the integration activities.

7. References

Bernardi, S., Merseguer, J., Petriu, D., (2013). *Model-driven Dependability Assessment of Software Systems*. Springer, ISBN 978-3-642-39512-3.

ETSI, (2016). ETSI GS MEC 003: Mobile Edge Computing (MEC); Framework and Reference Architecture V1.1.1, available online at: http://www.etsi.org/deliver/etsi_gs/MEC/001_099/003/01.01.01_60/gs_MEC003v010101p.pdf.

Gómez, A., Merseguer, J., Di Nitto, E., Tamburri, D., A., (2016). *Towards a uml profile for data intensive applications*. In Proceedings of QUDOS'16, pages 18–23, New York, NY, USA, 2016. ACM.

PrEstoCloud, (2017a). *D7.1 - As-Is and To-Be Scenarios*. Confidential deliverable, available only for consortium members and Commission Services.

PrEstoCloud, (2017b). *D2.3 - Requirements for the PrEstoCloud platform*. Public deliverable, available at: <http://prestocloud-project.eu/new/deliverables/>. _

Appendix A - Broker: additional instructions related to the architecture

Since the Broker is a very important component for an efficient realization of the proposed architecture, we provide in this appendix some additional information about:

- the organization / structure of the topics exchanged through the Broker;
- the guidelines how the Broker can be used.

A1. The structure of the topics

The following tables provide details how the interaction of other components is supported through the Broker.

Component 1 (source)	Component 2 (destination)	Topic	Message format	Constraints (throughput/size)	Description	Protocol
Inter-Site Network Virtualization, Cloud VM's Edge devices	Communication & Message Broker	monitoring.<group_id>.<device_type>.<device_id>	{ [<attribute>:<value> ...], timestamp: <value> }		Monitoring data from NetData like JSON-formatted string	AMQP
		monitoring/<group_id>/<device_type>/<device_id>	{ [<attribute>:<value> ...], timestamp: <value> }			MQTT
Communication & Message Broker	Workload & predictor	monitoring.<group_id>.<device_type>.<device_id>	{ [<attribute>:<value> ...], timestamp: <value> }	Defined in the tool configuration, depends on the use-case	All monitoring data	AMQP
Communication & Message Broker	Situation Detection Mechanism, Mobile Context Analyzer	monitoring.<group_id>.<device_type>.<device_id>	{ [<attribute>:<value> ...], timestamp: <value> }	-Per s/min/h -upon request -KBs/event	Receive regularly pushed events & events on request from edge devices for certain parameters	AMQP
Communication & Message Broker	Security Enforcement Mechanism, Autonomic Data Intensive Application Manager	monitoring.<group_id>.<device_type>.<device_id>	{ [<attribute>:<value> ...], timestamp: <value> }	Whenever published		AMQP

Component 1 (source)	Component 2 (destination)	Topic	Message format	Constraints (throughput/size)	Description	Protocol
Workload Predictor	Communication & Message Broker	prediction.<device_type>.<device_id>.<attribute>	{ prediction_value:<value>, timestamp: <value> }	When prediction is done	Attribute predictions	AMQP
Communication & Message Broker	Situation Detection Mechanism, Resources Adaptation Recommender	prediction.<device_type>.<device_id>.<attribute>	{ prediction_value:<value>, timestamp: <value> }	Whenever published		AMQP
Situation Detection Mechanism	Communication & Message Broker	situation.<event_pattern_name>	{ device_id:<value>, metric_value: <value>, timestamp: <value> }	Upon detection	Detected situations	AMQP
Communication & Message Broker	Resources Adaptation Recommender	situation.<event_pattern_name>	{ device_id:<value>, metric_value: <value>, timestamp: <value> }		Detected situations from Situation Detection Mechanism	AMQP
Resources Adaptation Recommender, Application Fragmentation Deployment	Communication & Message Broker	deployment.req	{ tosca_id:<value>, old_tosca_id:<value>, timestamp: <value> }	When new type-level TOSCA is stored to Cloud and Edge resource Repository	Event containing id for retrieving new TOSCA from Repository	AMQP

Component 1 (source)	Component 2 (destination)	Topic	Message format	Constraints (throughput/size)	Description	Protocol
Communication & Message Broker	Autonomic Data Intensive Application Manager	deployment.req	{ tosca_id:<value>, old_tosca_id:<value>, timestamp: <value> }	When type-level TOSCA is stored	Event containing id for retrieving new TOSCA from Repository	AMQP
Autonomic Data Intensive Application Manager	Communication & Message Broker	deployment.ack	{ tosca_id:<value>, timestamp: <value> }	When deployment is finished	Deployment is finished	AMQP
Communication & Message Broker	Situation Detection Mechanism	deployment.ack	{ tosca_id:<value>, timestamp: <value> }	When deployment is finished	Deployment is finished	AMQP

A2. Broker Usage Guide

This section provides an explanation of the communication JAVA libraries, for publishing and subscribing to broker with AMQP and MQTT protocols. Libraries are available on our public maven, and for using them add repository:

```
<repository>
  <id>archiva.public</id>
  <name>Nissatech Public Repository</name>
  <url>http://maven.nissatech.com/repository/public/</url>
</repository>
```

All libraries log actions with log4j. And exceptions are logged with stack trace.

Notes

If both consumers connect to a broker and then the broker goes down, when it gets up again, consumers will automatically reconnect. But if consumers try for first time to connect to the broker and the broker is down, a user will get an exception that the connection is failed. So the user needs to try subscribing again after some period of waiting. Because of a lot of parameters for consuming, the user needs to create one instance of a class for every subscribing to the topic. The user also needs to manage to save the topic and the queue name so if a consumer app goes down, after reconnecting it will use same queue and get all messages published in that period of time.

Publisher can try to publish to the broker, when the broker is up or down, and buffer message if it's not possible to connect to broker or publish it. After a first success connection it will publish all buffered messages. The user can manage to save all buffered messages if the produce app goes down.

AMQP MESSAGE CONSUMER LIBRARY CLASS

For consuming messages with AMQP protocol you can use our maven library:

<https://maven.nissatech.com/#artifact/com.nissatech.presto/amqp-consumer/6.0>

We used RabbitMQ amqp-client 5.3.0 for creating consumer class.

Basic usage of consumer:

```

import com.rabbitmq.client.Envelope;
import consumer.AMQPConsumer;
...
java.util.function.BiConsumer<String, String> consumerFunction = this::doSomething;
AMQPConsumer consumer = new AMQPConsumer(broker_IP_address, topic, consumerFunction);
...
//consumer.useSSL(keyStorePath, keyPassphrase, trustStorePath, trustPassphrase);
try {
    consumer.subscribe();
} catch (AlreadySubscribedException | IOException e) {
    System.out.println(e.getMessage());
} catch (SubscribingFailedException e) {
    waitXminAndRetry(3);
}
...
//doSomething is the method that gets executed on every received message
private void doSomething(String message, String topic) {
    ...
}

```

Consumer has two constructors: (1) one with host address (IP of broker node), topic (topic to be subscribed to) and messageConsumer (biConsumer for processing message and envelope); with this constructor use **subscribe** function without arguments; (2) other constructor *doesn't set topic*, so when subscribing you need to use **subscribe** function with topic argument. Or he gets exception that topic can't be null. Constructors also initialize AMQP connection:

- Class create new connection factory, set **host**, **username** and **password** to it. For now it uses default (guest) username and password. When we add specific users to RabbitMQ, with **setUsernameAndPassword** this data can be changed.
- Default **vhost** is /, and probably there is no need for some other, but we enable for user to **setVhost**, if later we create specific vhost for specific users.
- Network recovery interval is set to 10s, with this we enabled automatic recovery so in network failure consumer try every 10s to reconnect, restore connection listeners, re-open channels, restore channel listeners. we also enabled topology recovery, which involves recovery of exchanges, queues, bindings and consumers. Also connection timeout (connection establishment timeout in milliseconds; zero for infinite) is set to 0.
- Port is automatically set to 5672. When user enable SSL on initiation of connection, port is changed to 5671, sslSocketFactory is prepared with given files and added to connection factory. If it failed to create this factory it log error and set connection without SSL.

This initialization is called when any of mentioned parameters is changed.

Subscribing check if topic is not null, check if there is no already other subscribing on this instance, try to create connection and channel, add listeners, declares exchange and queue, binds that queue to exchange and starts consuming:

- Check if topic name that user tries to connect isn't null. Log error message and return IOException. Also because of a lot of parameters for consuming and that

the messages from other topics are probably processed in other way, user needs to create one instance of class for every subscribing to different topic. So it checks if there is some connections already. Log error message and return `AlreadySubscribedException` if this is case.

- After this try to create connection and channel. If it doesn't success log error and return `SubscribingFailedException`.
- Add `ShutdownListener` to connection and channel, so they log when connection is lost. And add `RecoveryListener` to channel so user gets log when recovery is started and finished.
- Limit of unacknowledged messages on a channel (**prefetchCount**) is set to 1000. User can change this also.
- By declaring `exchange` we verify that exchange exists, or create it if needed. This method creates an exchange if it does not already exist, and if the exchange exists, verifies that it is of the correct and expected type. Default exchange is presto.cloud TOPIC exchange, and probably there is no need for some other, but we enable for user to **setNewExchange** with other name and type of exchange.
- We also declare `queue`. If **queue name** is null or empty string, we create it like "amqp-subscriber-rabbit" + nanoTime. User can also set some **queue name**, and this functionality is used for recovery of consumer application. By default queue is **durable**, non **autodelete**, non **exclusive**, and has 24 **hours** to live. All this parameters can be changed before subscribing.
- By binding queue we say to exchange that it can send data to that queue to a defined topic.
- If some exception occurred in this steps, connections will be closed. Error logged and `SubscribingFailedException` will be sent to user.
- After this we start consuming message. Message payload and topic name are processed with consume function. If an application needs to persist data, then it should ensure the data is persisted prior to returning from this function, as after returning from this function, the message is considered to have been delivered (auto acknowledge), and will not be reproducible. And consumer logs the data.
- In the end user can **unsubscribe** from broker. With unsubscribing also channel and connection are closed.

AMQP MESSAGE PRODUCER LIBRARY CLASS

For producing messages with AMQP protocol you can use our maven library:

<https://maven.nissatech.com/#artifact/com.nissatech.presto/amqp-producer/3.0>

We used RabbitMQ amqp-client 5.3.0 for creating producer class.

Basic usage of producer:

```
import producer.AMQPProducer;
...
AMQPProducer producer = new AMQPProducer(broker_IP_address);
...
//producer.useSSL(keyStorePath, keyPassphrase, trustStorePath, trustPassphrase);
try {
    producer.publish(message, topic);
}
```

```
catch (IOException e) {
    System.out.println(e.getMessage());
}
```

Producer has two constructors: (1) one with host address (IP of broker node) and topic (topic to be published to), with this constructor use **publish** function only with message argument; (2) other constructor *doesn't set topic*, so when publishing you need to use **publish** function with topic argument. Or he gets exception that topic can't be null. Constructors also initialize AMQP connection:

- Class create new connection factory, set **host**, **username** and **password** to it. For now it uses default (guest) username and password. When we add specific users to RabbitMQ, with **setUsernameAndPassword** this data can be changed.
- Default **vhost** is /, and probably there is no need for some other, but we enable for user to **setVhost**, if later we create specific vhost for specific users.
- Automatic recovery and topology recovery are disabled, because we create new connection and channel per message. Also connection timeout (connection establishment timeout in milliseconds; zero for infinite) is set to 0.
- Port is automatically set to 5672. When a user enables SSL on initiation of connection, port is changed to 5671, sslSocketFactory is prepared with given files and added to connection factory. If it failed to create this factory, it logs error and sets connection without SSL.

This initialization is called when any of mentioned parameters is changed.

Publishing checks if topic is not null, buffers message, tries to create connection and channel, add listeners, declares exchange, publishes message to defined topic, logs work and closes channel and connection:

- Check if topic name that user tries to connect isn't null. Log error message and return IOException.
- Persistence of message is automatically set to true. With one of public function user can set persistent of sent message. Or it can set persistence for all messages with **setPersistent**. True set **Delivery mode** (Should the message be persisted to disk?) to 2, and false is setting it to 1.
- Buffered message and log work.
- After this try to create connection and channel. If it doesn't success log message and error.
- By declaring exchange we verify that exchange exists, or create it if needed. This method creates an exchange if it does not already exist, and if the exchange exists, verifies that it is of the correct and expected type. Default exchange is presto.cloud TOPIC exchange, and probably there is no need for some other, but we enable for user to **setNewExchange** with other name and type of exchange.
- On first successful connection it publishes all buffered messages like FIFO, and logs work.
- If some exception occurred in this step, connections will be closed. Error will be

- logged. Because of buffering we don't need to send exception to user.
- At end channel and connection are closed.

MQTT MESSAGE CONSUMER LIBRARY CLASS

For consuming messages with MQTT protocol you can use our maven library:

<https://maven.nissatech.com/#artifact/com.nissatech.presto/mqtt-consumer/3.0>

We used eclipse.paho.client.mqttv3 1.2.0 for creating producer class.

Basic usage of consumer:

```
import consumer.MQTTConsumer;
...
java.util.function.BiConsumer<String, String> consumerFunction = this::doSomething;
MQTTConsumer consumer = new MQTTConsumer(broker_IP_address, topic, consumerFunction);
...
//consumer.useSSL(keyStorePath, keyPassphrase, trustStorePath, trustPassphrase);
try {
    consumer.subscribe();
} catch (AlreadySubscribedException | IOException e) {
    System.out.println(e.getMessage());
} catch (SubscribingFailedException e) {
    waitXminAndRetry(3);
}
...
//doSomething is the method that gets executed on every received message
private void doSomething(String message, String topic) {
    ...
}
```

Consumer has two constructors: (1) one with host address (IP of broker node), topic (topic to be subscribed to) and messageConsumer (biConsumer for processing message and envelope), with this constructor use **subscribe** function without arguments; (2) other constructor *doesn't set topic*, so when subscribing you need to use **subscribe** function with topic argument. Or he gets exception that topic can't be null. Constructors also initialize MQTT connection options:

- Class create new [MqttConnectOptions](#), set **host**, **username** and **password** to it. For now it uses default (guest) username and password. When we add specific users to RabbitMQ, with **setUsernameAndPassword** this data can be changed.
- Default **vhost** is /, and probably there is no need for some other, but we enable for user to **setVhost**, if later we create specific vhost for specific users.
- [Automatic reconnect](#) is set to true, in the event that the connection is lost, the client will attempt to reconnect to the server. It will initially wait 1 second before it attempts to reconnect, for every failed reconnect attempt, the delay will double until it is at 2 minutes at which point the delay will stay at 2 minutes. Also connection timeout (connection establishment timeout in milliseconds; zero for infinite) is set to 0.
- [Clean session](#) is set to false and both the client and server will maintain state across restarts of the client, the server and the connection. The server will treat a subscription as durable.
- [Keep alive interval](#) measured in seconds, define the maximum time interval between messages sent or received. It enables the client to detect if the server

is no longer available, without having to wait for the TCP/IP timeout. The client will ensure that at least one message travels across the network within each keep alive period. In the absence of a data-related message during the time period, the client sends a very small "ping" message, which the server will acknowledge. The default value is 60 seconds.

- Port is automatically set to 1883. When user enables SSL on preparation of connection options, port is changed to 8883, sslSocketFactory is prepared with given files and added to connection factory. If it failed to create this factory it logs error and sets connection without SSL.

This initialization is called when any of mentioned parameters is changed.

Subscribing check if topic is not null, check if there is no already other subscribing on this instance, generate queue name, try to create client, connect client to broker with set options, set callback and starts consuming:

- Check if topic name that user tries to connect isn't null. Log error message and return IOException. Also because of a lot of parameters for consuming and that messages from other topics are probably processed in other way, user needs to create one instance of class for every subscribing to different topic. So it checks if there is some connection already. Log error message and return AlreadySubscribedException if this is case.
- MQTTConsumer instance try to create new MQTT client with **queue name** and broker url. If **queue name** is null or empty string, we generate it with ClientId ["paho" + nanoTime]. RabbitMQ queue name depends on generated ClientId, and for ClientId have queue per subscription QoS level. It's like "mqtt-subscription-" + **queue name** + QoSLevel. User can also set some **queue name**, and this functionality is used for recovery of consumer application. If creating client doesn't have success, log error and return SubscribingFailedException.
- Set [callback](#) to same class. Enable an application to be notified when asynchronous events related to the client occur. Had three methods:
 - One is called when the connection to the server is lost. Log event.
 - Second is called when delivery for a message has been completed, and all acknowledgments have been received. We don't use this.
 - And third is called when a message arrives from the server. In this method we accept message and topic name with message consumer. If an application needs to persist data, then it should ensure the data is persisted prior to returning from this function, as after returning from this function, the message is considered to have been delivered (auto acknowledge), and will not be reproducible. And consumer logs the data.
- Try to connect it to broker with specified connection options.
- [Subscribe](#) to defined topic with selected level of QoS. Default state of QoS is 1.
 - Transient (QoS0) subscription use non-durable, auto-delete queues that will be deleted when the client disconnects.
 - Durable (QoS1) subscriptions use durable queues. Whether the queues are auto-deleted is controlled by the client's clean session flag. Clients with clean sessions use auto-deleted queues, others use non-auto-deleted ones.

This can be changed with [setDurable](#) and [setAutoDelete](#).

- If some exception occurred in this steps, connections will be closed. Error logged and SubscribingFailedException will be sent to user.
- In the end user can **unsubscribe** from broker. With unsubscribing user also

disconnects client and closes connection.

Difference to AMQP is that some parameters are set in rabbitmq configuration file. Like name of exchange, queue time to live, and prefetch count.

```
{exchange,      <<"presto.cloud">>},  
{subscription_ttl, 86400000},  
{prefetch,      10},
```

MQTT MESSAGE PRODUCER LIBRARY CLASS

For producing messages with MQTT protocol you can use our maven library:

<https://maven.nissatech.com/#artifact/com.nissatech.presto/mqtt-producer/6.0>

We used eclipse.paho.client.mqttv3 1.2.0 for creating producer class.

We had some problem because stable version of Paho MQTT Java client has some known issue with client disconnecting and thread exit but we workaround that (<https://github.com/eclipse/paho.mqtt.java/issues/402>).

Basic usage of publisher:

```
import producer.MQTTProducer;
...
MQTTProducer producer = new MQTTProducer(broker_IP_address);
...
//producer.useSSL(keyStorePath, keyPassphrase, trustStorePath, trustPassphrase);
try {
    producer.publish(message, topic);
catch (IOException e) {
    System.out.println(e.getMessage());
}
```

Producer has two constructors: (1) one with host address (IP of broker node) and topic (topic to be published to), with this constructor use **publish** function only with message argument; (2) other constructor *doesn't set topic*, so when publishing you need to use **publish** function with topic argument. Or he gets exception that topic can't be null. Constructors also initialize MQTT connection options:

- Class create new [MqttConnectOptions](#), set **host**, **username** and **password** to it. For now it uses default (guest) username and password. When we add specific users to RabbitMQ, with **setUsernameAndPassword** this data can be changed.
- Default **vhost** is /, and probably there is no need for some other, but we enable for user to **setVhost**, if later we create specific vhost for specific users.
- [Automatic reconnect](#) is set to false, because of buffering the messages. Also connection timeout (connection establishment timeout in milliseconds; zero for infinite) is set to 0.
- [Clean session](#) is set to true, client is not subscribing, but only publishing messages to topics, it doesn't need for server to treat a subscription as durable (to any session information be stored on the broker).
- [Keep alive interval](#) measured in seconds, define the maximum time interval between messages sent or received. It enables the client to detect if the server is no longer available, without having to wait for the TCP/IP timeout. The client will ensure that at least one message travels across the network within each keep alive period. In the absence of a data-related message during the time period, the client sends a very small "ping" message, which the server will acknowledge. The default value is 60 seconds.

- Port is automatically set to 1883. When a user enables SSL on preparation of connection options, port is changed to 8883, sslSocketFactory is prepared with given files and added to connection factory. If it failed to create this factory it log error and set connection without SSL.

This initialization is called when any of mentioned parameters is changed.

Publishing check if topic is not null, buffered message, try to create client, connect client to broker with set options, publish message to defined topic, log work and close client:

- Check if topic name that user tries to connect isn't null. Log error message and return IOException.
- Persistence of message is automatically set to true. With one of public function user can set persistent and/or retained status of sent message. Or he/she can set persistence for all messages with **setPersistent**. True set **QoS** to 1, and false is setting it to 0. By default message is not retained.
- Buffered message and log work.
- After this try to create new MQTT client with random generated ID and broker url. If it doesn't succeed, log message and error.
- Connect it to broker with specified connection options. If some exception occurred client will be closed. Error will be logged. Because of buffering we don't need to send exception to user.
- On first successful connection it publishes all buffered messages like FIFO, and logs work.
- At end it disconnects and closes client.

As it said in consumer, exchange name is set in rabbitmq config file.

```
{exchange, <<"presto.cloud">>}
```

SSL (for all libraries)

If user wants to use SSL, for now, it need to call **useSSL** function with data (paths and passphrases) about keystore and truststore before subscribing in any way. User can also check if **isSsl**, and after adding data to stores change status of SSL with **turnOnSsl** on and off. Also there are functions for getting the data about stores.

For now, TLS works with paths to keystore and truststore and broker only trust his own certificate, because of that we provide client folder for all users. With CA from on broker instances, we create and sign client cert, and create stores with next script.

```
#!/bin/bash
set -eu
#
# Prepare the client's stuff.
#
mkdir client
cd client

# Generate a private RSA key.
openssl genrsa -out key.pem 2048
```

Generate a certificate from new private key.

```
openssl req -new -key key.pem -out req.pem -outform PEM -subj /CN=$(hostname)/O=client/
-nodes
```

Sign the certificate with our CA.

```
cd ../testca-pilot
openssl ca -config openssl.cnf -in ../client/req.pem -out ../client/cert.pem -notext -batch
-extensions client_ca_extensions
```

Create a key store that will contain this certificate.

```
cd ../client
openssl pkcs12 -export -out key-store.p12 -in cert.pem -inkey key.pem -passout
pass:NissaPrEstoCloud
```

Create a trust store that will contain the certificate of our CA.

```
openssl pkcs12 -export -out trust-store.p12 -in ../testca-pilot/cacert.pem -inkey ../testca-
pilot/private/cakey.pem -passout pass:Pr3570Cloud
```

Make files visible to user

```
chmod 755 key-store.p12 trust-store.p12
```

So for usage unzip file with “PrEstoNissaCloud” password and use stores in classes like:

```
instance.useSSL("/home/nikola/Desktop/client/key-store.p12", "NissaPrEstoCloud".toCharArray(),
"/home/nikola/Desktop/client/trust-store.p12", "passXXX".toCharArray());
```

We will try to make chain of trust and implement this in way that you can create your certs and get them signed by our CA.

How to reconnect consumer to same queue if app goes down?

For every subscriber back up is created (we keep topics and queue names in backup file).

```
public void backUp(DataConsumer dataConsumer) {
    String topic = dataConsumer.getTopic();
    String queueName = dataConsumer.getQueueName();
    File dir = new File(file);
    dir.mkdir();

    try (BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(
        new FileOutputStream(file + topic))) {
        writer.write(topic + "\n");
        writer.write(queueName);
    } catch (Exception e) {
        throw new SubscriberBackupException("Could not create back up for subscriber", e);
    }
}
```

If the connection fails, the subscriber automatically reconnects to topic it was connected to before the fail.

If our application that is in charge of receiving data from the broker fails, after the new start, the back up for every subscriber is restored. This way we can be sure that data that was in queues when the crash happened is received. Without this back up, new queues would be created and data would be lost.

```
private void createSubscribersFromBackup(File dir) {
    for (File file : dir.listFiles()) {
        DataConsumer dataConsumer = consumerDataBackup.getBackup(brokerUri, file);
        dataConsumer.subscribe();
    }
}

public DataConsumer getBackup(String brokerUri, File fileName) {
    try (BufferedReader reader = new BufferedReader(new InputStreamReader(
        new FileInputStream(fileName)))) {
        String topic = reader.readLine();
        String queueName = reader.readLine();
        DataConsumer dataConsumer = new DataConsumer(brokerUri);
        dataConsumer.setQueueName(queueName);
        return dataConsumer;
    } catch (Exception e) {
        throw new SubscriberBackupException("Could not get back up for subscriber", e);
    }
}
```

How to save buffered messages form producer if app goes down?

```
import producer.BufferedAMQPMessage;

public class BufferedMessagesSaver {

    private static BufferedMessagesSaver bufferedMessagesSaver;
    ObjectMapper objectMapper;
    private Logger logger = Logger.getLogger(getClass().getName());
    String file = "buffer";

    private BufferedMessagesSaver() { }

    public static BufferedMessagesSaver getBufferedMessagesSaver() {
        if (bufferedMessagesSaver == null)
            bufferedMessagesSaver = new BufferedMessagesSaver();
        return bufferedMessagesSaver;
    }

    public void saveBufferedMessages(List<BufferedAMQPMessage> messageList) {
        try {
            objectMapper.writeValue(new File(file), messageList);
        } catch (IOException e) {
            logger.log(Level.SEVERE, "Could not back up buffered messages.");
        }
    }

    public List<BufferedAMQPMessage> readBufferedMessages() {
        List<BufferedAMQPMessage> bufferedAMQPMessages = new ArrayList<>();
        File file = new File(this.file);
        if (file.exists()) {
            try {
```

```

        bufferedAMQPMessages = objectMapper.readValue(file, new
TypeReference<List<BufferedAMQPMessage>>() {});
    } catch (IOException e) {
        logger.log(Level.SEVERE, "Could not restore buffered messages.");
    }
    file.delete();
}
return bufferedAMQPMessages;
}
}

```

Appendix B - Monitoring the Edge: On/Offloading Client

An On/Offloading Client is installed on every edge resource. On/Offloading Client is responsible for registering the edge node in the Cloud and Edge Resources Repository through the Edge On/Offloading Server. Moreover, it responds to the Edge On/Offloading Server's requests for the on/offloading tasks which can be start or stop of application between edge nodes and datacenters. Such communication between the Edge On/Offloading Server and On/Offloading Client is made via a Java Socket API.

The On/Offloading Client has been developed as a lightweight Java code for deployment on Raspberry Pi and similar devices. In the next step of the PrEstoCloud project, it will be defined if it would be possible to run the implemented On/Offloading Client on Android-based devices. However, this component will possibly have a generic architecture able to be deployed on at least all types of Linux-based devices.

The typical scenario is a system which includes sensors able to measure some parameters and send the raw data to a container-based application running on an edge node. The application consists of two parts: (i) aggregating data and (ii) processing data. In this scenario, icons represent as follows:

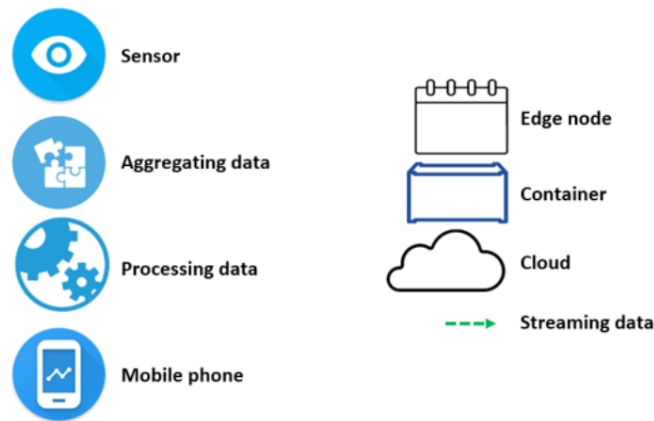


Figure 11: Meaning of icons used to show the functionality of On/Offloading Agent

In such a scenario, the part used to aggregate data receives the raw streaming data, collects and sends the aggregated data to another part which is used to process the data. In this case, both aggregating and processing data are performed in the container running on the edge node.



Figure 12: Both aggregating and processing data are performed in the container running on the edge node

For example in the logistic use case, all sensors such as accelerometer and magnetometer are connected to the edge node. In the edge node, data is aggregated and processed by the application. This application is useful in order to notify stakeholders (e.g. driver via alerts on a mobile phone or tablet) on situations where a possible accident may occur or attention is required.

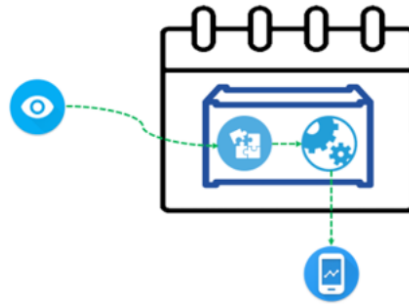


Figure 13: Presence of mobile phone or tablet

There are conditions where an edge node is not able to provide computing operations any more. For example if it is overloaded due to an increase in the number of sensors connected to the system during execution or the edge node is going to run out of storage capacity. In such conditions, the part which is used to process the data should be run on the cloud. In some cases, sensors are physically connected to the edge node, and the part used to aggregate the data needs to be run on the edge node. Therefore in this case, the edge node operates as intermediary to transmit the data to the cloud for data processing.

An On/Offloading Client is installed on every edge node. In such conditions, the On/Offloading Client responds to the Edge On/Offloading Server's requests for the on/offloading tasks. In this case, the request is terminating (stop) the processing data on the edge node or launching (start) the processing data on the cloud. In other words, if this condition happens, there will be two containers. One container is employed to aggregate data on the edge node, and another one is used to process data on the cloud. To this end three successive steps should be accomplished by the On/Offloading Client: (1) stop the current container instance running on the edge node, (2) start a new container instance on the cloud that includes the data processing part, and (3) start a new container instance on the edge node that includes the data aggregating part.

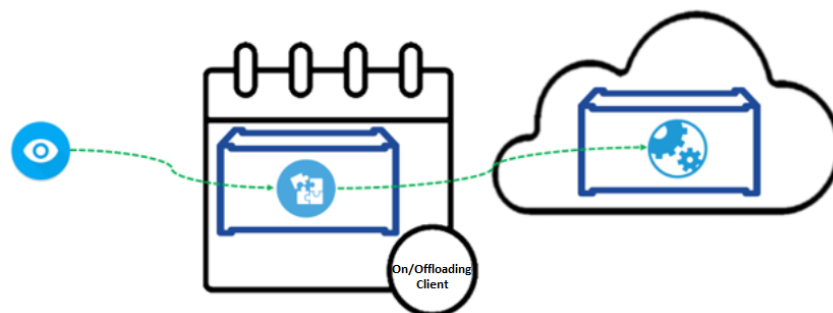


Figure 14: Stop the processing data on the edge node and start the processing data on the cloud performed by the On/Offloading Client

In some other cases, the part employed to aggregate data should also start working on the cloud in addition to the part used to process data. In such situations, On/Offloading Client responds to the Edge On/Offloading Server's request which is stop

both aggregating and processing data on the edge node and start them on the cloud. In this case, both aggregating and processing data are executed in the container running on the cloud. To this end two successive steps should be accomplished by the On/Offloading Client: (1) stop the current container instance running on the edge node, (2) start a new container instance on the cloud that includes both data aggregating part and data processing part.



Figure 15: Stop both aggregating and processing data on the edge node and start them on the cloud performed by the On/Offloading Client

Appendix C - Technical integration in Use cases

In order to illustrate the connection between the use case development and the technical architecture, in this section we provide some details about the mapping between the technical architecture and use cases.

It is done through providing a figure with depicted mappings and some indications why the integration is required through a short questionnaire.

LiveU Use Case

In the following figure, the components to be used in the use case are marked in green.

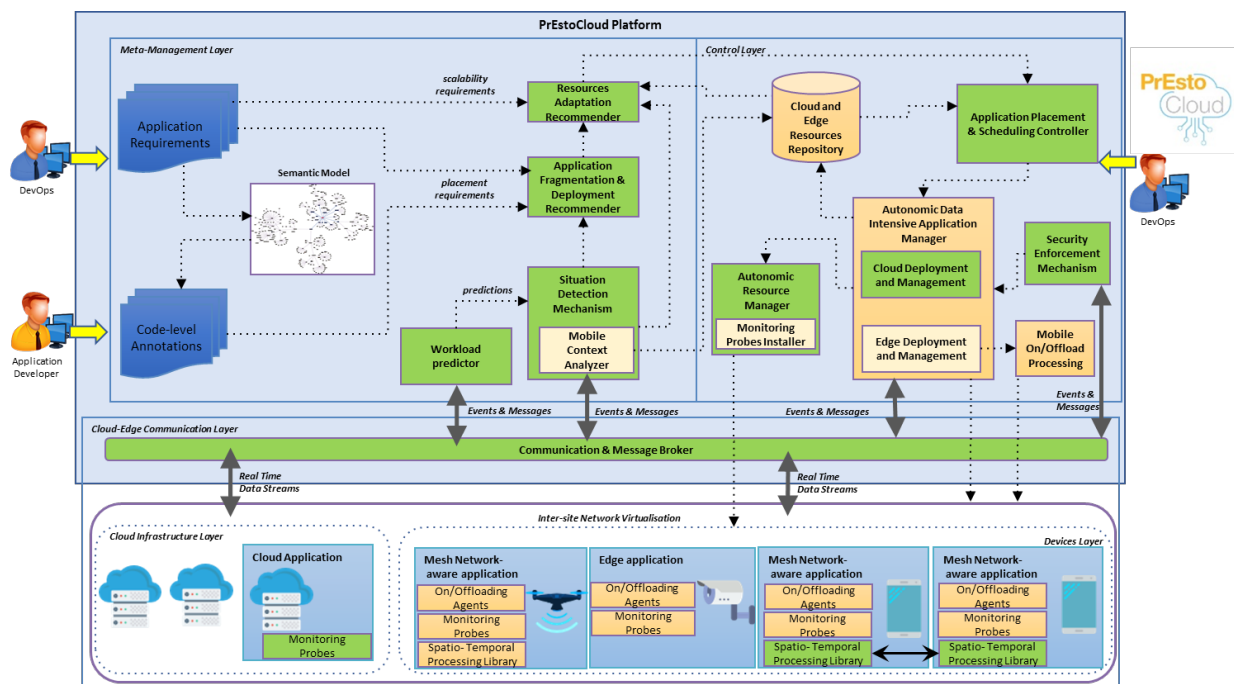


Figure 16: Mapping of the LiveU use case to the conceptual architecture

Questionnaire:

Q1: What data processing methods you believe it makes sense to be sent to the extreme edge?

Video Analytic and Storing - this will prevent sending unreliable, or un-required data from the extreme edge onward.

Link Manager - for creating an improved performance and reducing a round trip time for feedback sensitive adaptive algorithms.

Q2: Do you have some example of methods that cannot/shouldn't be move to the extreme edge? (optional)

Yes, all the methods running on the edge device shall stay there,

Q3: How the incoming data stream scales? Provide one or two examples.

The Media use case deals with unpredicted amount of streams coming from various events planned and unplanned, from professional broadcasters and from amateur consumers contributing their streams.

Q4: How your compute infrastructure will be able to scale? What you believe are realistic limits of scalability for your case (in terms of VMs, cost, devices etc)?

We have a large number of on premise devices, that can be considered as a private cloud, the scaleout to public clouds will be required when resources of private cloud will not be available, also, some of the processing such as the Link Manager and the Video Analytic shall be performed at the mobile edge to increase performance and reduce transfer costs. It is thus required to scale at mobile edge to support the streams processing.

Q5: Do you have a goal in QoS that you need to keep?

Yes, interactive services require minimum delays, in addition, video streaming in prime time, shall be high quality. Thus it is very important for us to have a stable, and continuous high performance with low delays.

CVS Use Case

Following figure illustrates the components to be used in the CVS use case.

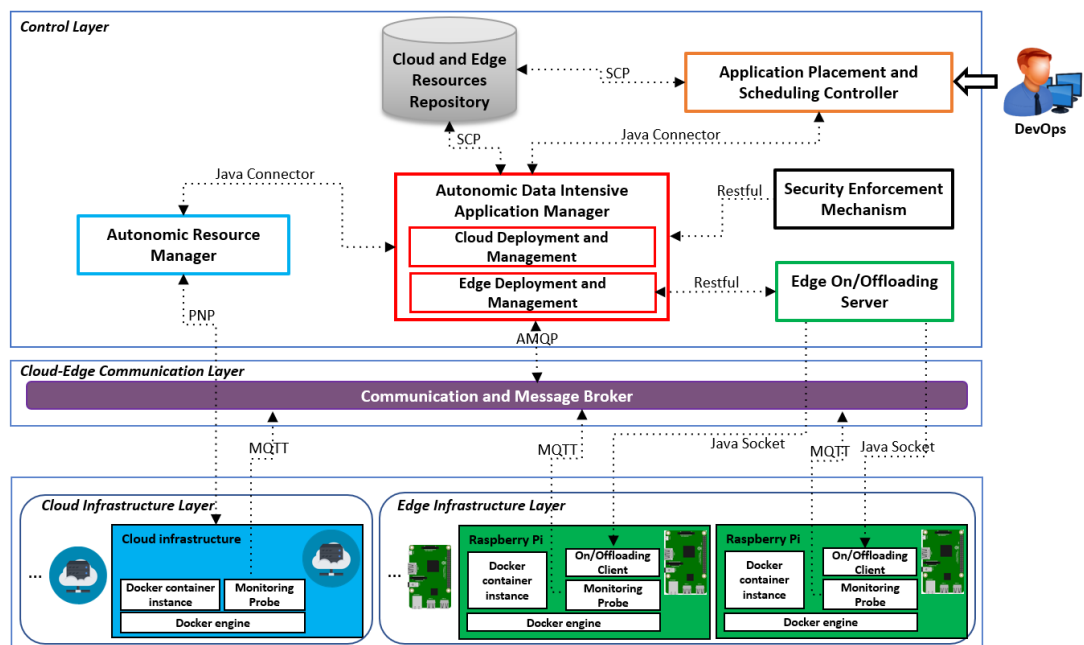


Figure 17: Mapping of the CVS use case to the conceptual architecture

Questionnaire:

Q1: What data processing methods you believe it makes sense be sent to the extreme edge?

The proposed framework is able to observe driving dynamics (e.g. acceleration, braking, turning, etc.), perform real-time data analytics at the extreme edge of the network and trigger alerts to drivers and fleet managers on situations where new decisions should be made. Therefore, data processing methods performed at the extreme edge are as follows:

- Collecting and storing the sensory data.
- fast extraction of useful information from the data streams that can be used to profile drivers' behavior.
- triggering automated alerts at run-time to help stakeholders (e.g. driver, logistic center, insurance company or vehicle owner).

Q2: Do you have some example of methods that cannot/shouldn't be move to the extreme edge? (optional)

Methods such as long-term storage of sensory data and complete offline data analytics e.g. simultaneous display of several graphs for a long period of time.

Q3: How the incoming data stream scales? Provide one or two examples.

Situations may arise in which an edge node is no longer able to provide storing or computing operations because there are no more spare storage capacity or available computing cycles for example due to increasing amount of data to be stored and processed. Therefore, the application running on the edge node should be terminated and deployed on the cloud side. In such conditions, the edge node operates as intermediary to receive the measured data from sensors and transmit the data to the cloud for data storage and processing.

Q4: How your compute infrastructure will be able to scale? What you believe are realistic limits of scalability for your case (in terms of VMs, cost, devices etc)?

Cloud resources which are applied as a pay-per-use on-demand infrastructure can employ auto-scaling mechanism when more or less resources are needed over time in conditions where the number of vehicles included in the transport logistic system varies at run-time. Therefore, a set of metrics related to cloud-based infrastructure should be measured continuously by the Monitoring Probe.

Q5: Do you have a goal in QoS that you need to keep?

Precision and recall of anomalies detected in real-time represent the number of reported alerts which are processed by the edge node and also valid (precision measure), and the number of possible alerts that the edge node actually detected (recall measure).

Aditess Use Case

The following figure illustrates the architecture realization for Aditess use case. With dark green are depicted the basic components shall be used in order to achieve the use case goals, while with light green are presented components that will be used but are considered beneficial but not obligatory.

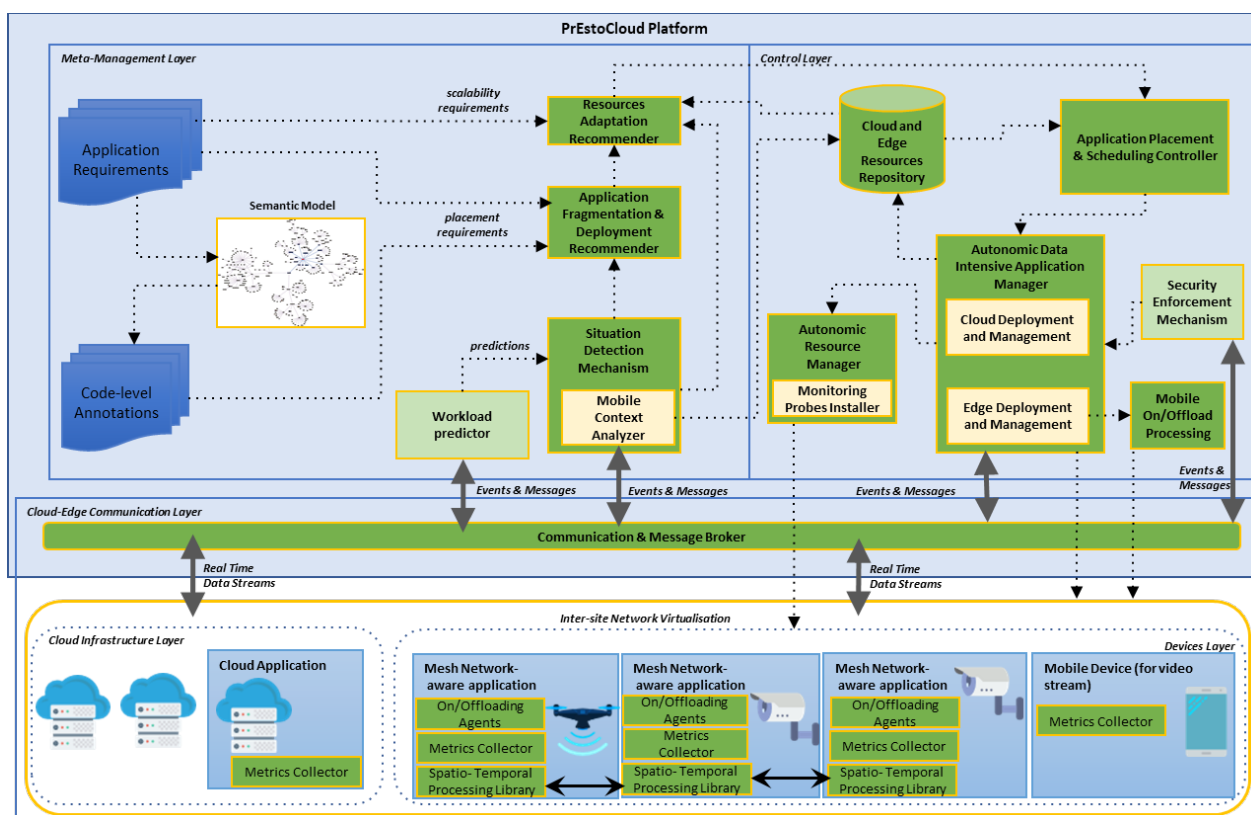


Figure 18: Mapping of the Aditess use case to the conceptual architecture

Questions:

Q1: What data processing methods you believe it makes sense to be sent to the extreme edge?

Some methods of video and audio stream analytics

Q2: Do you have some example of methods that cannot/shouldn't be move to the extreme edge? (optional)

-

Q3: How the incoming data stream scales? Provide one or two examples.

CCTVs or UAVs that were inactive are providing stream, Mobile phones also can provide input stream and this will lead to additional data input.

Q4: How your compute infrastructure will be able to scale? What you believe are realistic limits of scalability for your case (in terms of VMs, cost, devices etc)?

Scaling can be done by using public cloud resources, but before that the local (RPU) and extreme edge resources must be exploited (see presentation for example). The available resources in the RPU, extreme edge and VMs should be monitored in order to decide scale in or scale out scenarios.

Q5: Do you have a goal in QoS that you need to keep?

In the case of audio and video analytics, we need each of the fragments that are created and running at edge or cloud resource to achieve some monitorable goals. This can be the time needed to execute the method of each fragment(e.g.: to be less than 1sec).