Adeptness

ADEPTNESS – Design-Operation Continuum Methods for Testing and Deployment under Unforeseen Conditions for Cyber-Physical Systems of Systems

EUROPEAN COMMISSION
Horizon 2020
H2020-ICT-01-2019
GA No. 871319

Deliverable No.	ADEPTNESS D5.5	
Deliverable Title	Workflow and toolchain documentation final release	
Deliverable Date	2023-01-31	
Deliverable Type	Report	
Dissemination level	Public	
Written by	UES	2022-12-09
Checked by	IKL, EGM	2023-01-16
Approved by	Executive board	2023-01-31
Status	First version	2023-01-27

H2020-ICT-01-2019 – 871319 – ADEPTNESS: Design-Operation Continuum Methods for Testing and Deployment under Unforeseen Conditions for Cyber-Physical Systems of Systems	
Acknowledgement	
The author(s) would like to thank the partners involved with the project for their valuable comments on previous drafts and for performing the review.	
Project partners	
1 – MGEP – Mondragon Goi Eskola Politeknikoa – ES 2 – ORO – Orona S. Coop – ES 3 – UES – Ulma Embedded Solutions S. Coop – ES 4 – SRL – Simula Research Laboratory S. Coop – NO 5 – BT – Bombardier Transportation Sweden – SE 6 – IKL – Ikerlan S. Coop – ES 7 – EGM – Easy Global Market SAS – FR 8 – MDH – Mälardalens Högskola – SE 9 – TUW – Technische Universität Wien – AT	
Disclaimer:	
This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871319.	

Document Information

Additional author(s) and contributing partners

Name	Organisation
Sergio Rodríguez Mikel Garay	UES
Aitor Arrieta Goiuria Sagardui Ibai Román Unai Muñoz	MGEP
Franck Le Gall Romain Magnani	EGM
Aitor Agirre Blanca Kremer	IKL
Shaukat Ali Liping Han	SRL
Zlatan Tucaković Meixun Qu	TUW
Wasif Afzal Amirali Piadehbasmenj	MDH

Document Change Log

Name	Date	Comments
v0.1	2022-12-09	Initial draft
v0.2	2022-12-23	Updated draft, all parterns' contributions added
v0.3	2023-01-20	Updated draft, including internal deep review
v1.0	2023-01-27	First version

Exploitable results

Exploitable results	Organisation(s) that can exploit the result
Step-by-step guidance to use microservices	Consortium partners
Available toolchain and configuration	Consortium partners

Contents

LIST OF FIGURES

1 PURPOSE OF THE DOCUMENT

1.1 Document structure

The structure of the document is as follows. In Section 3 a brief definition of specified subsystems participating in the Adeptness ecosystem. The specification of the microservices belonging to every subsystem describes the variety of collaborative components.

Section 4 focuses on the workflow sequence to deploy and run design-operation continuum engineering for CPSoS starting from test artifact generation and provisioning, followed by plans definition and concluding with the workflow exectution.

Finally, Section 5 summarizes the content of this document and provides the guidelines to extend the approach to other microservice ecosystems.

1.2 Deviations from the original Description in the Grant Agreement Annex 1 Part A

1.2.1 Description of work related to deliverable in GA Annex 1 – Part A

There are no deviations with respect to work of this deliverable.

1.2.2 Time deviations from original planning in GA Annex 1 – Part A

Deliverable date has been delayed from M34 (2022-10-31) to M37 (2023-01-31) due to a three-months amendment accepted in 2022.

1.2.3 Context deviations from the original plan in GA Annex 1 – Part A

There are no deviations from the Annex 1.

2 INTRODUCTION

A workflow is a sequence of steps involved in moving from the beginning to the end of a working process. In the context of Adeptness, such a working process is a dynamic ecosystem, in which several microservices part of the CPSoS enable a design-operation continuum engineering process.

Here, the workflow identifies four different stages:

- Test Artifact Generation, which focuses on the semi-automatic generation of validation artifacts, such as Oracles or Test Cases.
- Planning, in which the plans for deploying, monitoring and validation are defined.
- Execution, in which the tasks and communication of the microservices at runtime is exposed, enabling the design-operation continuum validation.
- Extensions, which extend the core functionality of Adeptness, and provides a closed feedback loop for CPSoS.

The availability of a guidance to setup, configure and run microservices requires several artifacts to enable the subscription and publication of the topics exchanged in the Adeptness ecosystem. There are four main domain subsystems defined in the core platform:

- Deployment subsystem
- Validation subsystem
- Monitoring subsystem
- Management subsystem

Underlying consumers of these domains are dependent on the deployment of each subsystem. Moreover, the interaction between the different domains contains an inherent sequence at configuration time, to enable microservice availability at runtime.

This document focuses on the steps required to enable the microservices deployment and availability at setup, and lifecycle management at runtime.

3 ADEPTNESS FRAMEWORK

The Adeptness ecosystem provides a collaborative framework, composed of several subsystems, each one with a specific role. The operational usage of subsystems is realized using the microservices attached to them. Each microservice is specialized in a specific task. The interaction between the microservices, both inside the subsystem and among subsystems, communicate through well-defined interfaces to achieve the common objective of a continuous validation process. The deliverable D1.2 – Interface of the microservices – specifies the API designed to support these interactions between microservices.

The Adeptness defined subsystems are:

- The Deployment subsystem, including deployment orchestrator and agents.
- The Monitoring subsystem including monitoring orchestrator and monitors..
- The Validation subsystem, including orchestrator, agents and oracles.
- The Management subsystem, comprising the Test-as-a-Service (TaaS) web application, as specified in D5.1. (Adaptation of TaaS for continuous CPSoS Testing) and the Stellio logger subsystem.
- The Test Case Generation subsystem.
- The Traceability subsystem, enabling Lifecycle Management.
- The Uncertainty subsystem, which enables uncertainty generation at design time and uncertainty detection at runtime.
- The Recovery subsystem, which enables the rules and actions required to recover the normal behaviour of the system, once a malfunction is detected at operational level.

The first four subsystems are part of the so-called Adeptness Core whereas the latter four are referred to as Adeptness Extensions.

Associated to each subsystem, there are specified one or more microservices. These are used for design and for runtime to exchange information from/to the CPSoS, specifically, in the cloud and within defined edge nodes.

In the document, the partners define the interactions between subsystems, forming a collaborative interoperability at CPSoS level, and between microservices. These interactions allow the specification of the workflow in the next Section.

In the following section, the document provides detailed information about each subsystem.

3.1 Core subsystems

3.1.1 Deployment Subsystem

The deployment subsystem is explained in deliverables D1.1 and D2.2. Section 4.4 of deliverable D1.1 contains the requirements and the first design of the subsystem. Section 4.3 of deliverable D2.2 describes exhaustively the implementation details of the subsystem. This section contains a brief description of the subsystem.

The deployment subsystem is responsible for the installation at each of the nodes of the microservices required to carry out a validation plan. The subsystem will receive the order of deployment from an automation server and is able to download the deployment plan from the repository, parse it and send the necessary messages to the deployment agents to download and execute the artefacts contained in the plan.

The defined microservices in deployment subsystem are:

- Deployment orchestrator microservice
- Deployment agent microservice

These two microservices communicate with each other via asynchronous messaging with well-defined interfaces. These messages are defined in deliverable D1.2 - Interfaces of microservices. The instances of microservices' interfaces are available in Adeptness GitLab repository for deployment orchestrator[1] and for deployment agent[2].

In the following subsections, we provide further details about defined microservices.

3.1.1.1 *Deployment orchestrator microservice*

The deployment orchestrator is an essential part of the deployment subsystem. In a scenario of continuous integration in CPSoS, different components need to be deployed on different CPS in order to not only instantiate the software components that contain the CPS functionality but also those other software artefacts that are needed for the validation of the CPSoS .

The deployment orchestrator is part of the Adeptness infrastructure framework. It is an Adeptness Python microservice that runs as a Docker container and is invoked from the automation server when executing the deployment pipeline.

The orchestrator needs a deployment agent on each of the CPS where it must deploy components. The orchestrator and the deployment agents communicate with each other through a well-defined MQTT interface. The orchestrator offers synchronous and asynchronous communication mechanisms. Synchronous communication is done through a Rest API and allows external actors, such as an automation server or a recovery service, to ask the orchestrator to execute a deployment plan. The asynchronous mechanism is performed through the MQTT protocol and is used by the orchestrator to send the deployment orders to the deployment agents and also to receive the deployment result from them. The orchestrator will also use this mechanism to publish the global deployment result for a complete deployment plan to anyone who wants to subscribe.

3.1.1.2 *Deployment agent microservice*

The deployment agent is the component that enables the remote deployment on each CPS node. It receives and processes commands from the deployment orchestrator and, based on the content of the message received, downloads and launches new services using the corresponding method. Currently, the deployment

[1] https://gitlab.com/adeptness/wp1/interfaces/deployment-orchestrator-interface

[2] https://gitlab.com/adeptness/wp1/interfaces/deployment-agent-interface

agent is able to deploy two types of applications: containerized applications based on Docker images and compressed files which may optionally contain processes to run.

Two separated deployment agents have been implemented, one in Python that can be executed natively or containerised and another one in C that runs only natively for resource constrained devices that are not capable of running the Python environment or Docker.

3.1.2 *Monitoring Subsystem*

The monitoring subsystem is explained in deliverables D1.1 and D2.2. Sections 4.5.1 and 4.5.2 of deliverable D1.1 contain the first design of the monitoring subsystem. Section 5.1 of deliverable D2.2 describes exhaustively the implementation details of the monitoring subsystem. Section 5.2 of deliverable D2.2 and sections 3.3, 3.4, 4.2 and 4.3 of deliverable D2.4 describe the storage features of the subsystem in-depth. This section contains a brief description of the subsystem.

The monitoring subsystem shall oversee monitoring all the relevant information, from the cyber physical systems, to validate the behaviour of the whole system. Each validation plan must be capable of executing a series of test cases. To carry out these tests, observations must be made in the SUT, and depending on each test case, certain system variables must be monitored. One or more monitoring agents will be deployed in each target/edge node. These monitor agents will be configured to read the proper variables depending on the monitoring plan. The monitoring orchestrator will configure these agents and will also trigger the order to the monitors to start sampling the variables and publishing them so that other microservices can make use of them.

The defined microservices and tools in monitoring subsystem are:

- Monitoring orchestrator
- Monitoring agents
- Trace library

Monitoring orchestrator and agents communicate with each other via a well-defined Rest-API. These interfaces are available in Adeptness Monitoring Interfaces GitLab repository[3].

The following subsections provide a brief description of these microservices and tools. For further details, check the associated deliverable D2.3 – Services software release for continuous monitoring in CPSoS.

3.1.2.1 *Monitoring orchestrator microservice*

The monitoring orchestrator is one of the core microservices of the Adeptness environment. It executes the monitoring plan sent by the automation server, configuring the distributed monitors. The monitoring orchestrator is itself an Adeptness microservice deployed as a Docker container in a cloud server.

The gist of the monitoring orchestrator lies in the monitoring plan it consumes. On the one hand, the orchestrator provides a REST-API to receive this plan from the automation server; on the other hand, the

[3] https://gitlab.com/adeptness/wp1/interfaces/monitoring-orchestration-interface *and* https://gitlab.com/adeptness/wp1/interfaces/monitoring-agent-interface

orchestrator uses a REST client to access the endpoint where each one of the monitor agents is offering a REST API to be configured.

3.1.2.2 *Monitoring agents microservices*

Monitoring agents are responsible for reading variables from the field bus and publishing them via MQTT so other microservices can use them. These monitoring agents are thus executed in the edge nodes. Monitored variables (sensors) are specific to the functionality of each node, their values are gathered from different sensors and field buses by the monitoring agents. Monitoring agents are configured by the monitoring orchestrator microservice to specify how to read the variables that the monitoring agent must observe. They must be able to access the physical interface with the configured connection settings (indicated in the monitoring plan) and publish the values of these variables in SenML format using MQTT.

3.1.2.3 *Trace library*

The trace library is able to trace, through MQTT, the value of source code internal variables (i.e., they are not present in physical field buses)**.** Thus, the source code must be instrumented to be able to use this functionality. These variables can be further used by the validation oracles, just the same way as any other monitored variable.

3.1.3 Validation Subsystem

The validation subsystem is responsible for performing validations of the CPSoS on different test levels, starting from SiL, HiL and up to Operation.

The subsystem is comprised of a validation orchestrator, responsible for coordinating the validation strategy, and several validation agents, coordinating the validation components in each of the edge nodes. The validation itself is carried out by oracle microservices, previously generated with the DSL2Oracle tool. External tools aid in the validation process providing access to domain-specific or legacy tools developed out of the scope of Adeptness. The validation strategies are defined on validation plans, where all the steps to be taken to perform the validation on the CPSoS are defined.

The defined microservices and tools in validation subsystem are:

- Validation orchestrator microservice
- Validation agent microservice
- Oracle microservice
- External tool microservice

3.1.3.1 *Validation orchestrator microservice*

The validation orchestrator microservice is in charge of managing the whole lifecycle of a validation plan in the context of the Adeptness ecosystem. Developed in Python, the validation orchestrator is capable of running in the major platforms where the Python interpreter or Docker is supported. All the communications are made exclusively through APIs[8] defined on the Adeptness architecture, meaning that the orchestrator is also agnostic to the use-case. The microservice exposes a REST API that enables the management of validation plans and their execution, and publishes the validation plan status and verdicts through MQTT. The validation orchestrator interacts with validation agents deployed in edge validation platforms though their REST API, and listens for their verdict and status changes through MQTT.

3.1.3.2 *Validation agent microservice*

The validation agent microservice is responsible for coordinating the execution of a validation plan and components in an evaluation platform (HiL, SiL, an installation in operation, etc.). As it happens with the validation orchestrator, the validation agent is also developed in Python, sharing the ability to be run on the majority of platforms. All the communications are also done through standard APIs defined on the Adeptness architecture.

The agent will coordinate the Oracles, and external tools required to perform an evaluation, configuring them, managing the execution and collecting the generated test results. The generated results are then aggregated and published, so that the validation orchestrator is able to collect the results.

3.1.3.3 *Oracle microservice*

The oracle microservice is responsible for evaluating operational values based on inputs provided by monitors. Test oracles defined via the DSL are integrated in the microservice and provided with capabilities to interact with the rest of the components via its REST and MQTT APIs[9]. The validation agents will coordinate with the oracle microservices for configuring them and to expect verdicts raised based on the performed evaluations.

3.1.3.4 *External tool microservice*

The external tool microservice is a convenience microservice developed with the integration of legacy, unsupported or out of scope tools into the Adeptness ecosystem. The external tool microservice exposes a basic API[10] and means of managing the execution of a sub-process or external application. This API can be extended to provide additional functionality and means to manage the tool being commanded by the external tool microservice. For each tool that wants to be integrated on the Adeptness ecosystem, an ad-hoc implementation of the external tool microservice is performed, creating external tool subtypes for each of the integrated tools.

3.1.4 Management Subsystem

3.1.4.1 *Stellio*

The Stellio context broker enables provisioning of the Adeptness plans as well as storage capabilities of the data coming from monitors and verdicts coming from validation agents. It serves as an interface between

[8] *https://gitlab.com/adeptness/wp1/interfaces/validation-orchestrator-interface*

https://gitlab.com/adeptness/wp1/interfaces/validation-agent-interface

[9] *https://gitlab.com/adeptness/wp1/interfaces/oracle-interface*

[10] *https://gitlab.com/adeptness/wp1/interfaces/external-tool-microservice-interface*

other Adeptness services (such as the Test Generator and OSCL Bridge) and the data collected from monitors and oracles. During the project course, the logging features explored two solutions: Stellio and Prometheus.

According to the analysis of the features in both solutions, the decision was to select Stellio solution as the main solution for the context description and storage in Adeptness. This has not prevented us from also exploring and implementing options to connect Prometheus to the Adeptness workflow in particular scenarios.

Stellio[12], created and maintained by EGM, is a FIWARE Generic enabler, freely accessible on GitHub[13], and released under the Apache Public License.

Beyond the implementation of the NGSI-LD standard[14], Stellio context broker is fully compliant with a "FIWARE architecture"[15]. This is realised both by the management of structural and contextualized data according to the common NGSI-LD data model, and by the integration with other components of the FIWARE ecosystem (IoT agents, data collection ...), which enabled thanks to compliance with the NGSI-LD standard. This standard guarantees the internal and external interoperability of the platform.

The principles implemented in the Stellio context broker are those of a modern, responsive architecture that can scale:

- Business micro-services, divided according to the three main families of the NGSI-LD API, subscribed to their topics of interest, and using data stores adapted to their business;
- A central message bus (Kafka) of the pub-sub type ensures the exchange of messages and events between the components of the platform in a decoupled, extensible, scalable and responsive way.

The data produced by the various external components integrated into the platform (sensors, results of verdicts, oracles...) are stored by the context broker in two different databases:

- Neo4J, for storing Information Context.
- PostgreSQL, for storing subscriptions/publications.

SenML-MQTT Bridge

Stellio SenML Bridge is intended to get data from deployed monitors, verdicts from oracles and aggregated verdicts from validation agents. A bridge has been developed and deployed to perform the transformation process from messages received in SenML format from an MQTT broker to an NGSI-LD based data format that can then be injected into the Stellio context broker.

3.1.4.2 *TaaS*

The TaaS (Test-as-a-Service) web application is responsible for setting up test campaigns and triggering the launch of the validation plans associated with the test campaigns. To do so, the TaaS web interface provides

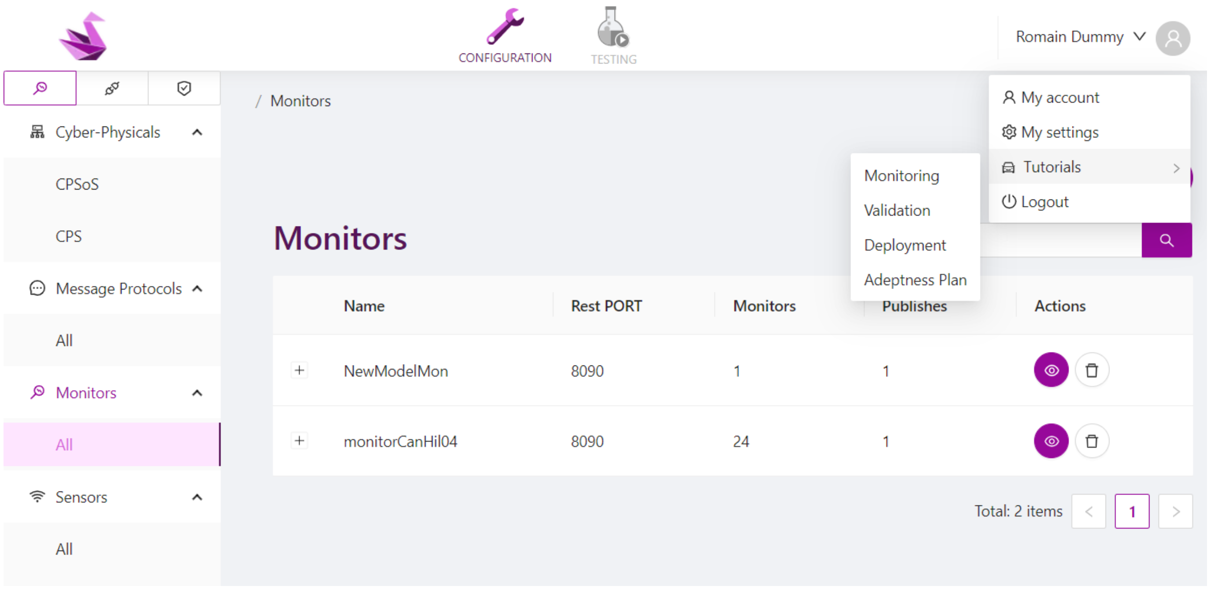
[12] https://stellio.io
[13] https://github.com/stellio-hub/stellio-context-broker
[14] https://www.etsi.org/deliver/etsi_gs/CIM/001_099/009/01.04.01_60/gs_cim009v010401p.pdf
[15] https://www.fiware.org/developers/catalogue/

tools to ease the configuration part, also called the provisioning part, where the Stellio context broker registers the needed components.

Nevertheless, even if the TaaS is here to help, the configuring part can still have many steps with a lot of forms to fill and pages to visit in different order, therefore it can be tedious for beginners to go through it without help. That is why multiple tutorials with guided steps have been developed in order to ease the provisioning for the user tester. Four tutorials have been implemented in total:

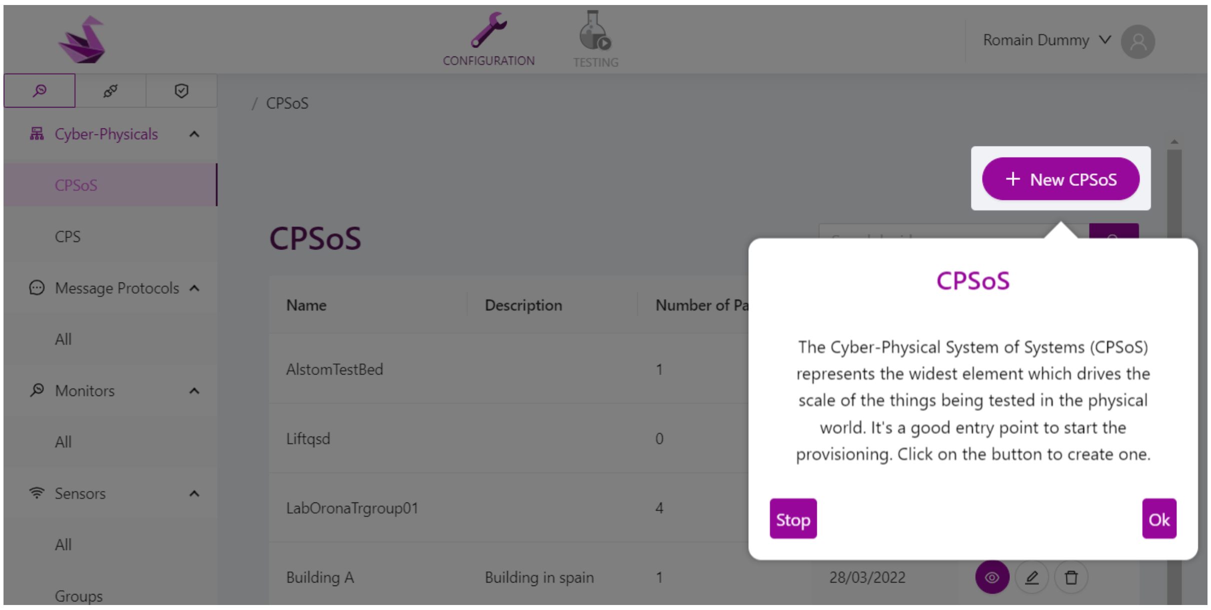
- Create a minimal viable monitoring plan
- Create a minimal viable deployment plan
- Create a minimal viable validation plan
- Create an Adeptness plan (this last one is the entry point from where to launch a test)

The idea is that by following all of them, the user can configure a simple but full Adeptness Plan from start to finish (see figures below).



Name	Rest PORT	Monitors	Publishes	Actions
NewModelMon	8090	1	1	
monitorCanHil04	8090	24	1	

Figure 1 - Access tutorials from top right user menu



CPSoS

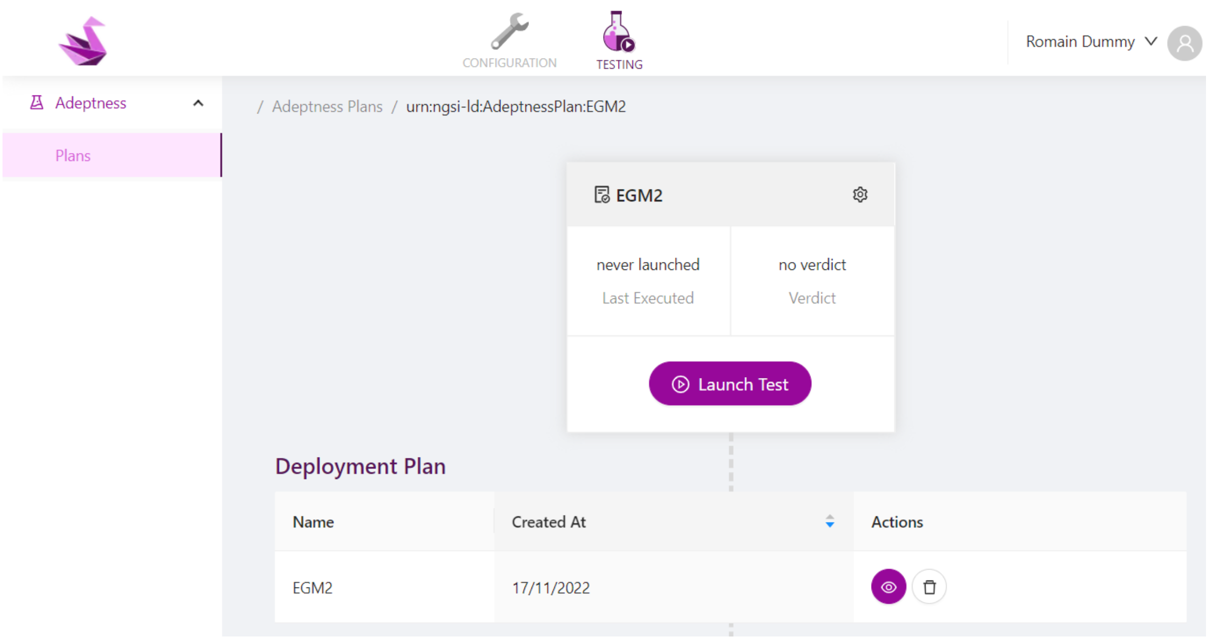
Name	Description	Number of Pa	
AlstomTestBed		1	
Liftqsd		0	
LabOronaTrgroup01		4	
Building A	Building in spain	1	28/03/2022

CPSoS

The Cyber-Physical System of Systems (CPSoS) represents the widest element which drives the scale of the things being tested in the physical world. It's a good entry point to start the provisioning. Click on the button to create one.

Figure 2 - Screenshot of the monitoring tutorial's first step

For security reasons, all those components that are being created during the configuration part are only reachable by default by the user who created them. Although he can decide to share them with each member of his groups. Which is a requirement for the Adeptness workflow, where different partners need access to some or all those components. Therefore, a dedicated Adeptness group has been created and any partner who needs access to those components can simply be added in it by an administrator and be able to share entities information amongst other user within the group.



never launched — Last Executed
no verdict — Verdict

Deployment Plan

Name	Created At	Actions
EGM2	17/11/2022	

Figure 3 - Screenshot of the Adeptness plan details page

Finally, from the Adeptness plan component page, the test can be launch with a unique click (cf.). The TaaS then uses all the configuration components to generate plans, such as Monitoring, Deployment and Validation. Three algorithms have been written for each plan's files generation. In addition, along each of those, lots of unit tests have been implemented to insure they keep generating files as intended, while the data model evolves, and the project keeps growing to fit new needs. Once these plans have been successfully generated, they are uploaded to the MinIO server. Then the path to those files on MinIO are passed to Jenkins when the TaaS interface triggers the Jenkins pipeline (see Figure 4).

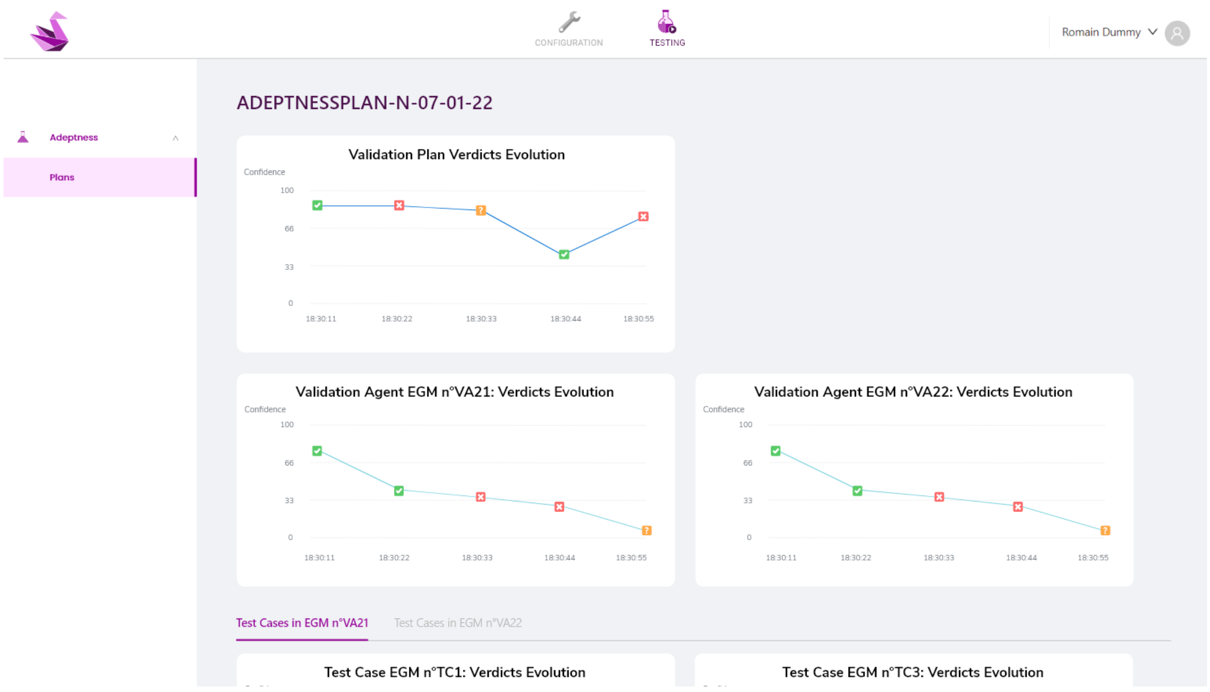

CONFIGURATION
TESTING
Romain Dummy
Adeptness
Plans
ADEPTNESSPLAN-N-07-01-22
Validation Plan Verdicts Evolution
Validation Agent EGM n°VA21: Verdicts Evolution
Validation Agent EGM n°VA22: Verdicts Evolution
Confidence
Test Cases in EGM n°VA21
Test Cases in EGM n°VA22
Test Case EGM n°TC1: Verdicts Evolution
Test Case EGM n°TC3: Verdicts Evolution

Figure 4 - Screenshot of an Adeptness Plan's Dashboard being validated

Once the test campaign is launched successfully, the TaaS front-end display the test Adeptness plan dashboard, and wait to be notified for each new verdict (by the socket notifier cf. 0), from each validation component type, and displays those verdicts in graphs representing their evolutions in time. This dashboard is dynamically generated based on how many validation components have been configured during the provisioning part by the tester (see Figure 4). The dashboard is also prepared to be able to show a history of previous testings, thanks to Stellio temporal API which allows the TaaS to retrieve data from a specific period of time only.

NGSI-LD socket notifier

In order to receive notifications from the Stellio logger during the testing phase, the TaaS subsystem needs to subscribe to multiple validation components' verdict. Each time a new TaaS instance (or, a new user) subscribes, the socket bridge API called *NGSI-LD socket notifier*, receives new notifications from Stellio. The notifier then dispatches them to the front-end using Web Sockets, according to the user/tester and the proper TaaS instance (see Figure 5). Thanks to this approach, the web app can be updated as soon as a new verdict is registered into the logger, and the user can be aware of those evolutions in real time.

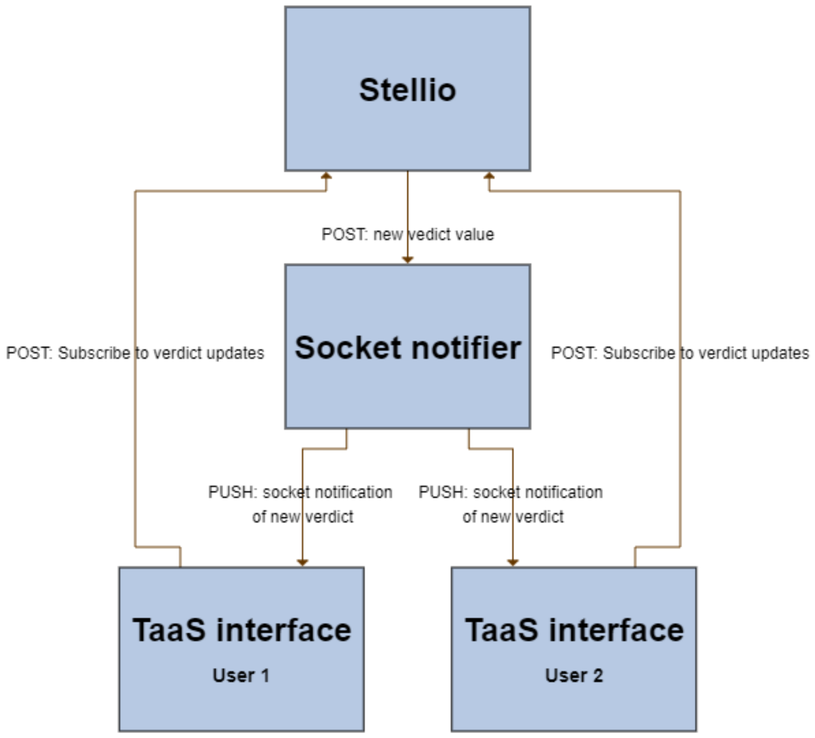
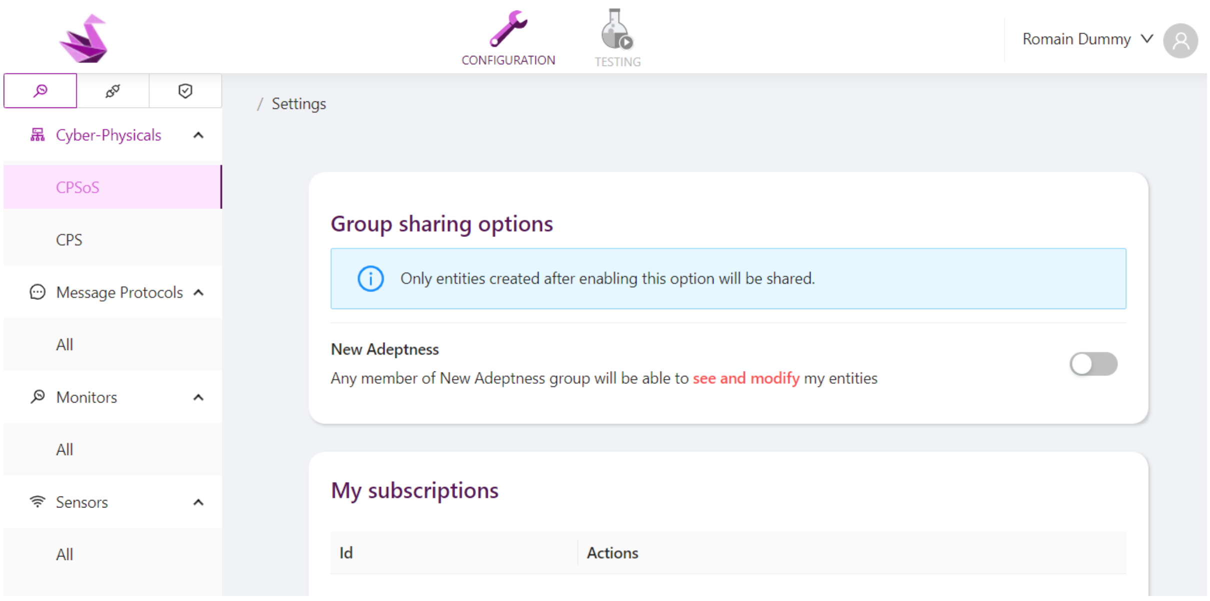

Stellio
POST: new vedict value
Socket notifier
POST: Subscribe to verdict updates
POST: Subscribe to verdict updates
PUSH: socket notification of new verdict
PUSH: socket notification of new verdict
TaaS interface
User 1
TaaS interface
User 2

Figure 5 - Diagram of Socket notifier in interaction between Stellio and TaaS

This architecture allows multiple users to see verdicts from the same test case's scenario or any other validation plan components being updated at the same time, provided that those components have been shared between these users. The TaaS interface is able to automatically share any newly created entities with selected group(s) by the current user, with a simple parameter to be set in the settings page (see Figure 6). However, sometimes, more fine-tuning with rights management is required. That is why the NGSI-LD rights management tool interface has been developed and deployed.



Group sharing options

Only entities created after enabling this option will be shared.

New Adeptness
Any member of New Adeptness group will be able to see and modify my entities

My subscriptions

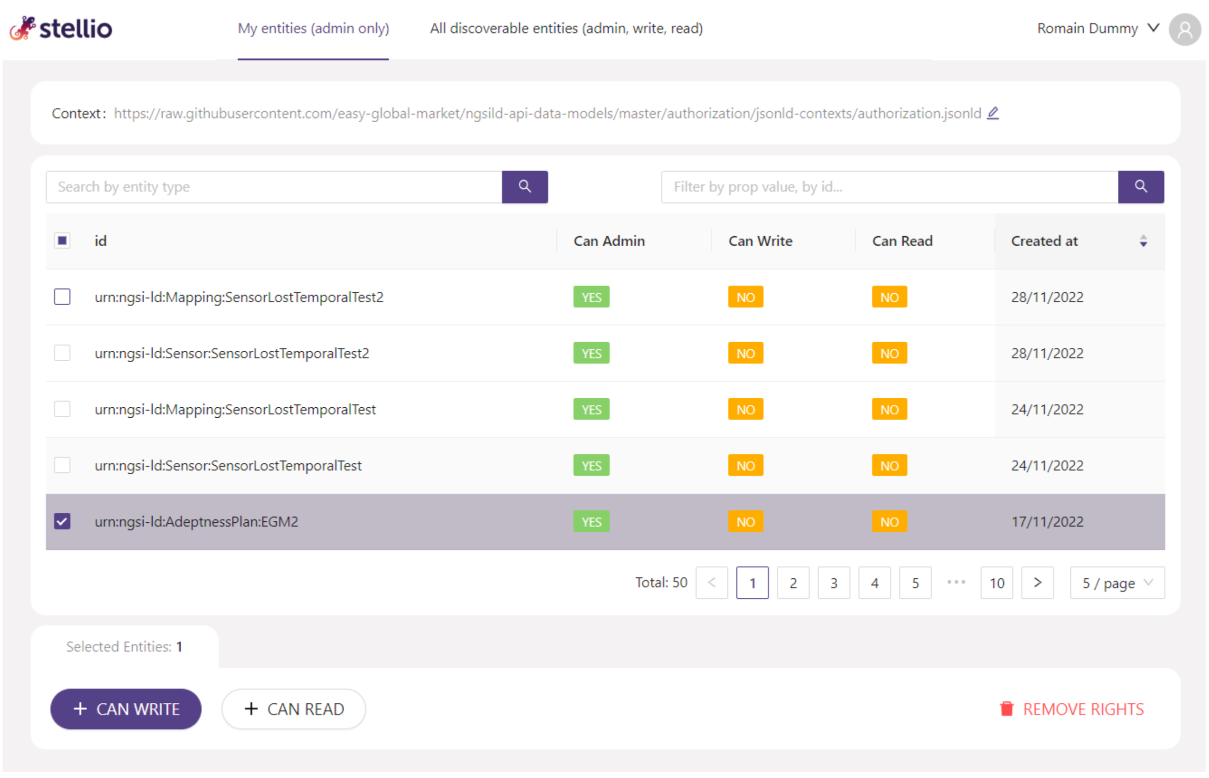
Id	Actions

Figure 6 - Screenshot of the user settings page

NGSI-LD Rights management

The NGSI-LD rights management tool is a separated interface from TaaS and is dedicated to managing rights on entities. Linked to Stellio, the tester user can find here every entity previously created on TaaS (or by any other mean). Those entities are shown by rights type, such as:

- Can admin: the given right by default to the user who is the entity creator. With the "admin" right, the user has full access to it.
- Can write: with a given write access, a user can reach, see and modify entity's information, but won't be able to delete it.
- Can read: with a given read access, a user can only reach and see the information of an entity.


stellio
My entities (admin only)
All discoverable entities (admin, write, read)
Romain Dummy

Context: https://raw.githubusercontent.com/easy-global-market/ngsild-api-data-models/master/authorization/jsonld-contexts/authorization.jsonld

Search by entity type
Filter by prop value, by id...

id	Can Admin	Can Write	Can Read	Created at
urn:ngsi-ld:Mapping:SensorLostTemporalTest2	YES	NO	NO	28/11/2022
urn:ngsi-ld:Sensor:SensorLostTemporalTest2	YES	NO	NO	28/11/2022
urn:ngsi-ld:Mapping:SensorLostTemporalTest	YES	NO	NO	24/11/2022
urn:ngsi-ld:Sensor:SensorLostTemporalTest	YES	NO	NO	24/11/2022
urn:ngsi-ld:AdeptnessPlan:EGM2	YES	NO	NO	17/11/2022

Figure 7 - Screenshot of the NGSI-LD Right management tool interface

Thanks to this interface, the user can still manage rights on components even those which have been reated without being shared. Or mistakenly shared with another group. Rights can be added or removed, to or from any entities selected in the table in the centre of the screen (see Figure 7). Then the user is asked to select on which group to perform the wanted action (add/remove).

3.2 Extensions

3.2.1 *Test Generation Subsystem*

The test generation subsystem is responsible for generating, automatically, test cases using a DSL (Domain-Specific Language). The test generation subsystem follows a tool-supported model-based testing (MBT= workflow that includes domain specific requirements, utilizes them for modelling and generates concrete and executable test scripts to validate the embedded system under test. The test generation subsystem makes use of an open-source MBT tool, GraphWalker, and its associated capabilities to be run as a Restful service.

The following diagram in Figure 8 shows all REST API Commands used to control GraphWalker (GW) and interact with the model. Below we also describe the commands in detail.

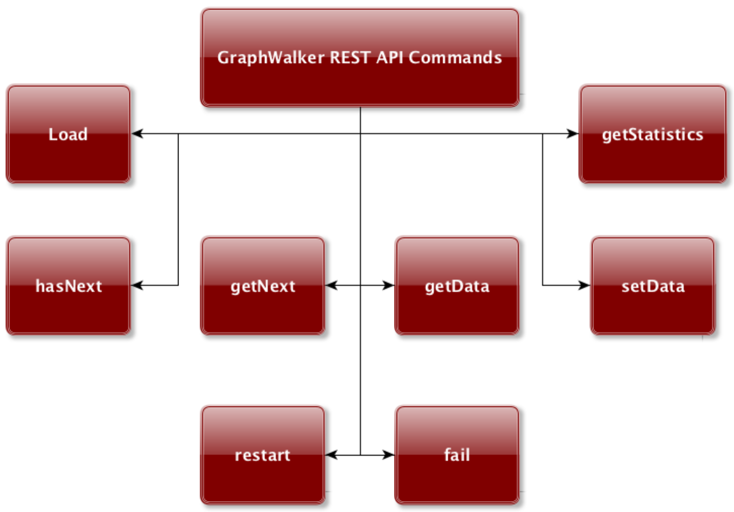


Figure 8 - GraphWalker REST API commands

When GW is run as a Restful service, there is an API that is used to interact with it. There exist eight REST API commands to interact with the GW service: `load`, `hasNext`, `getNext`, `getData`, `setData`, `restart`, `fail` and `getStatistics`. Below is the description of each of these commands:

- `REST:load`

  The Rest call `load` uploads model(s) in JSON format and resets GraphWalker with the new test. Upload the model to the service using `curl`:

  ```
  curl -vH "Content-Type: text/plain" --data @<YOUR MODEL NAME IN JSON>.json
  http://localhost:8887/graphwalker/load
  ```

  Response: If the request was successful, the "result" will be "ok".

- `REST:hasNext`

  The Rest call `hasNext` queries the service if there are any more elements to fetch. If yes, then the fulfilment of the stop conditions has not yet been reached.

  GET Request

  ```
  http://service-host:8887/graphwalker/hasNext
  ```

  From a Linux terminal, using `curl`:

  ```
  curl http://localhost:8887/graphwalker/hasNext
  ```

  Response: If the request was successful, the "result" will be "ok". If there are more elements to get, "**`hasNext`**" will be "true".

- `REST:getNext`

  The Rest call `getNext` is used to get the next element from the path generation. GraphWalker will, given the path generator, calculate what the next element should be and step one step forward in the execution of the model. The element name is returned in the response.

  GET Request

```
http://service-host:8887/graphwalker/getNext
```

From a Linux terminal, using curl:

```
curl http://localhost:8887/graphwalker/getNext
```

Response: If the request was successful, the "**result**" will be "**ok**". "**currentElementName**" will hold the name of the element.

- `REST:getData`

  The Rest call `getData` is used to ask GraphWalker for the current data values of the current model.

  GET Request

  ```
  http://service-host:8887/graphwalker/getData
  ```

  From a Linux terminal, using curl:

  ```
  curl http://localhost:8887/graphwalker/getData
  ```

  Response: If the request was successful, the "**result**" will be "**ok**". The "data" part will hold the data as key value elements.

- `REST:setData`

  The Rest call `setData` is used to set data in the current model.

  PUT Request

  ```
  http://service-host:8887/graphwalker/setData/&lt;JAVA SCRIPT>
  ```

  From a Linux terminal, using curl:

  ```
  curl -X PUT http://localhost:8887/graphwalker/setData/<YOUR KEY>=<YOUR
  VALUE>;
  ```

  Response: If the request was successful, the "**result**" will be "**ok**".

- `REST:restart`

  The Rest call restart will reset currently loaded model(s) to their initial states.

  PUT Request

  ```
  http://service-host:8887/graphwalker/restart
  ```

  From a Linux terminal, using curl:

  ```
  curl -X PUT http://localhost:8887/graphwalker/restart
  ```

  Response: If the request was successful, the "**result**" will be "**ok**".

- `REST:fail`

  The Rest call **fail** terminates the path generation of the test session.

  PUT Request

  ```
  http://service-
  host:8887/graphwalker/fail/String%20to%20explain%20the%20failure
  ```

  From a Linux terminal, using curl:

```
curl -X PUT
http://localhost:8887/graphwalker/fail/String%20to%20explain%20the%20failure
```

- `REST:getStatistics`

  The Rest call getStatistics will fetch the current statistics of the session.

  POST Request

```
http://service-host:8887/graphwalker/getStatistics
```

  From a Linux terminal, using curl:

```
curl http://localhost:8887/graphwalker/getStatistics
```

  Or, using the python -m json.tool to prettify the output:

```
curl -s http://localhost:8887/graphwalker/getStatistics | python -m
json.tool
```

  Response: If the request was successful, the "**result**" will be "**ok**".

3.2.2 DSL2Oracle tool

The DSL2Oracle tool, on the one hand, enables the specification of test oracles specifically designed to assess functional and non-functional properties of CPSoS. On the other hand, it enables the automated generation of the test oracle microservice. This tool has been developed in xTEXT[16], but the test generator has been instantiated on the cloud and integrated with a GitLab pipeline for the automated generation of the test oracle microservice.

3.2.3 OSLC Bridge

The OSLC Bridge subsystem makes use of the information model available at the system, to enable Lifecycle Management using OSLC (Open Services for Lifecycle Management) specification. The full description about the basics of OSLC and OSLC Bridge is available in D4.5 – Report on traceability mechanism from operational data to development lifecycle.

This subsystem uses an independent server (in this case, IBM Rational Team Concert Server) containing the domain definition relevant for Adeptness, such CCM (Change and Configuration Management), QM (Quality Management, based on test planning, test construction, and test artifact management) and RM (Requirements Management).

The OSLC Bridge brings a common standardized interface connected to product lifecycle management tools. The full specification is available at Open Services Specifications Website[17].

OSLC enables the integration of federated, shared information across tools that support different but related domains. The most relevant for Adeptness project are:

[16] https://www.eclipse.org/Xtext/

[17] https://open-services.net/specifications/

- Core Specification, which defines the overall approach to OSLC-based specifications and capabilities. These capabilities are often needed across several domains and provide a solid foundation for reading and writing linked data resources.
- Configuration and Change Management (CCM) Specification eases the management of product change requests, activities, tasks and relationships between those, and related resources such as requirements (Requirements Management domain) or test cases (Quality Management domain).
- Requirements Management (RM) Specification provides a management of requirements, requirement collections and supporting resources defined in OSLC Core specification.
- Quality Management (QM) Specification defines the test plans, test cases and test results of the software delivery lifecycle. These represent individual resources along with their relationships to other shared resource types such change requests and requirements.

The main microservice provided by the OSLC Bridge is to receive all oracles test case execution results and the conversion of these results in the form of standardized test case execution results and defects.

For this purpose, a REST API-based periodic subscription to data logger requests the validation results according to the Adeptness validation plan, and the corresponding artifacts are created to be used to adapt the validation plan to the appeared defects at HiL/SiL validation phase.

3.2.4 STL-Refined Prometheus Monitoring

STL-Refined Prometheus Monitoring (SRPM) plugin is a monitoring system with formal language, Signal Temporal Logic (STL) refinement of alerting rule that enables it to connect to MQTT. This extension work focuses on how the formal language STL guides the generation of monitoring rules for Prometheus.

SenML-MQTT Prometheus agent

MQTT Bridge is implemented within the project. It is employed to connect to the cloud broker and listen to the configured topic in MQTT. The MQTT Bridge will output the message, which is accepted by SRPM Bridge as input.

STL runtime monitor

Signal Temporal Logic (STL) is a formal, declarative specification formalism for capturing sophisticated temporal requirements, such as safety, reachability and liveness property, for CPS/IoT applications, which are typically interpreted over real-time and real-valued behaviours and admit quantitative semantics to allow measuring how far are behaviours from satisfying or violating specifications. In this extension, STL will be utilized to guide the generation of alerting rules in Prometheus on the basis of the monitoring requirement of the CPS system.

3.2.5 Uncertainty Subsystem

The uncertainty subsystem includes two parts:

- Uncertainty Generation at design-time.
- Uncertainty Detection at run-time.

Uncertainty Generation

Uncertainty Generation employs various techniques such as Genetic Algorithm (GA) and Reinforcement Learning (RL), aiming to generate unforeseen situations from different aspects automatically at design-time so as to collaborate with the validation subsystem by providing the validation subsystem with unforeseen traffic profiles.

Uncertainty Detection

Uncertainty Detection is responsible for detecting anomalies and poor robustness of the CPSoS. The anomaly detection employs a digital twin and various machine learning techniques, e.g., Generative Adversarial Network (GAN), to detect anomalies that were not known at the design-time using the time-series data from a CPSoS. If no anomalies are detected, the robustness of the CPSoS can be further assessed. The uncertainty-aware robustness assessment employs various statistical tests to comprehensively quantify the robustness of the CPSoS against various uncertainties occurring in the run-time, e.g., unpredicted loading and unloading time of a passenger. Such uncertainties are specified with uncertainty datatypes in the uncertainty libraries, e.g., uncertain mass of passengers is specified with interval.

3.2.6 Recovery microservice

The recovery microservice is responsible for managing the execution of recovery actions once an uncertain situation has been detected or a verdict in an oracle has failed. It uses the results obtained from the oracles in order to assess the status of the CPSoS. Then, based on the Recovery Rules and depending on the information it receives, the Recovery Microservice decides to launch or not a specific recovery action. The launched recovery actions are sent to their corresponding microservices in order to be executed.

4 ADEPTNESS WORKFLOW

This section explains the workflow expected for configuration and runtime of the microservices contained in the subsystems defined in Section 3.

Here, the workflow identifies four different stages:

- Test Artifacts Generation: This phase is in charge of generating both Oracles and Test Cases using automatic generation methods.
- Plans Definition: This phase defines the setup and all the components needed for the validation of the CPSoS and provisions them in Stellio. Adeptness plan is generated, including deployment, monitoring and validation plans.
- Execution: This phase is in charge of running on target platform the SiL and HiL validation.
- Extensions: The extensions provide plug-ins, which are not essential to the validation of CPSoS, but provide a benefit to the execution results. In addition, the extensions feed from validation execution and operation, and perform reasoning about uncertainty, defected tests and monitoring.

The following subsections provide a detailed sequence of the phases described in Figure 9.

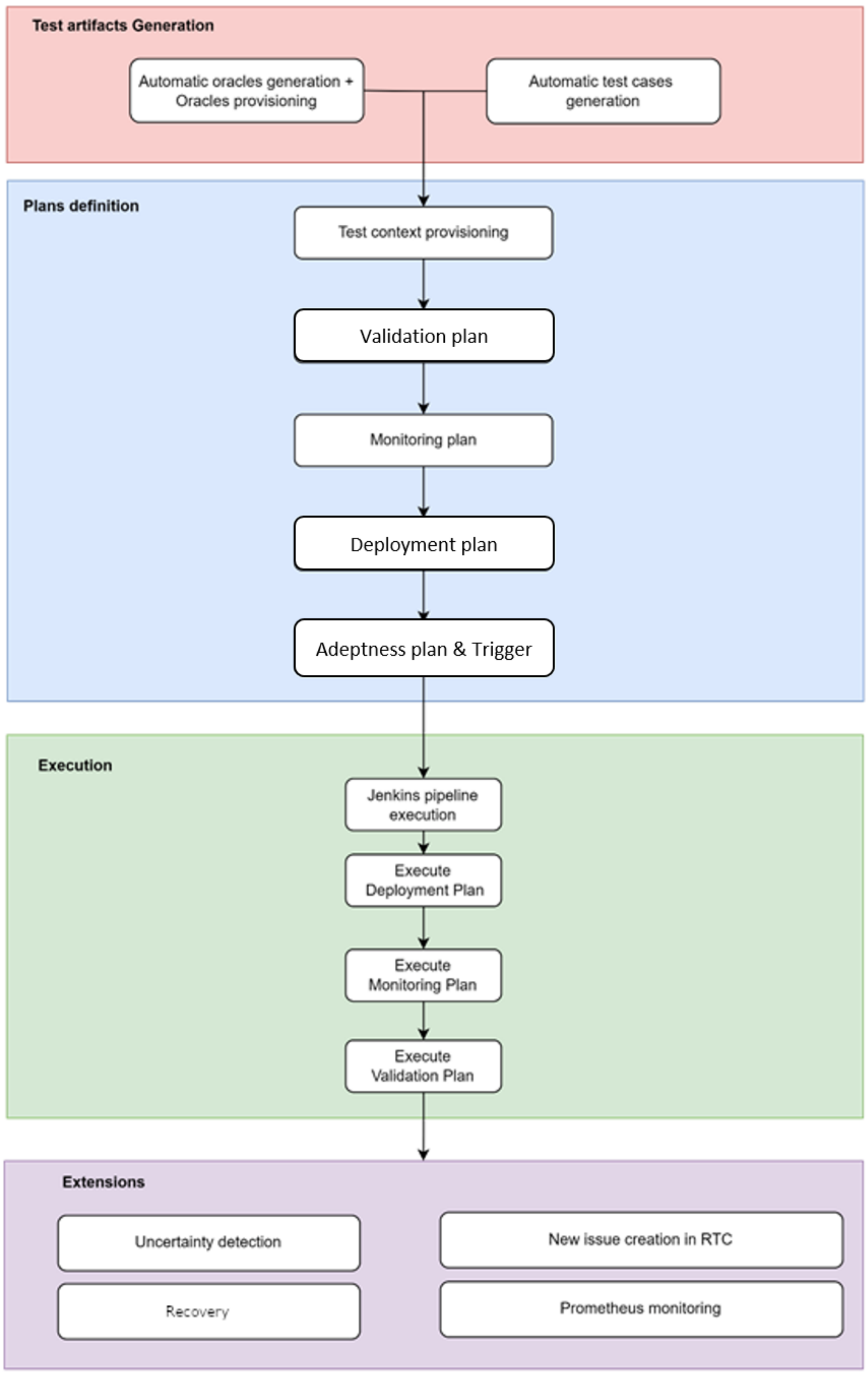


Figure 9 - Workflow definition

4.1 Test Artifact Generation and Provision

The Test Artifact Generation and Provision refers to the automatic or semi-automatic generation of oracles and test cases. This is realized through two components (see Figure 10):

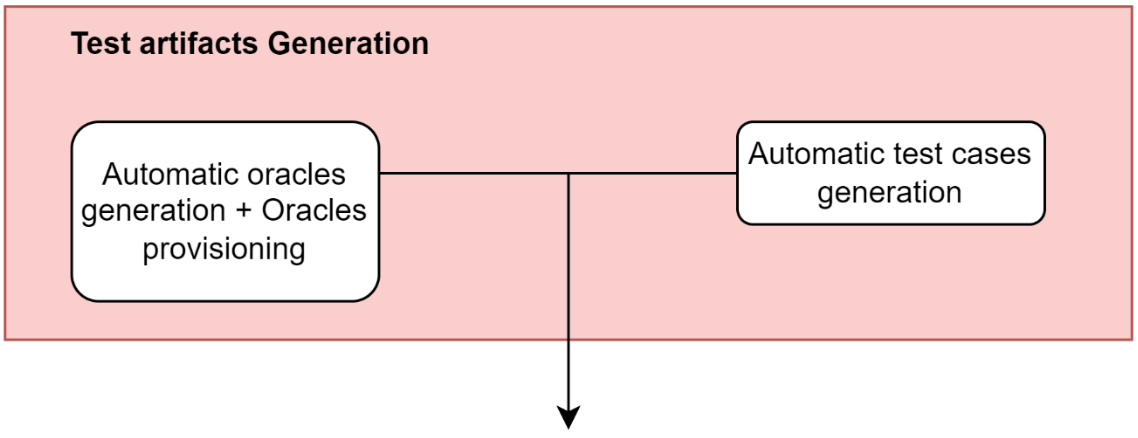

Test artifacts Generation
Automatic oracles generation + Oracles provisioning
Automatic test cases generation

Figure 10 - Workflow to Test Artifacts Generation

4.1.1 Test Case Generation and Provisioning

The Test Generator microservice takes as input the requirements specification in a defined format. The format conforms to the domain specific language (DSL) defined for the use case partners. In case of Alstom-Bombardier, the requirements specification of the train control management system follows the *Given-When-Then* style, where the *Given* part describes the pre-conditions, the *When* part describes the specified behaviour and the Then part describes the expected changes due to the specified behaviour. While the offline test generation acts completely on the DSL-specified requirements specification, the online test generation functionality takes current test execution status through its log file from the Logger microservice, with the goal to affect test generation based on the operation of the SUT.

Towards the automation of the testing process, test cases can be generated using Model based Testing (MBT). MBT is an automated testing technique, which generates the test artefacts based on a model representing the SUT. After specifying the requirements in *Gherkin*[18] format, a tester can model the system using certain modelling notations i.e. Unified Modelling Language (UML), Finite State Machine (FSM) etc. and can generate the abstract test cases by traversing through the model elements (i.e. states and transitions). To facilitate the modelling efforts, a Domain Specific Language (DSL) could be used to define the requirements in a well-defined format and to extract certain information (i.e. states, transitions, guard variables and their values) from it. A DSL contains an abstract syntax based on meta-model to provide a custom and platform independent support for a specific domain.

The proposed MBT workflow that covers requirements specification to test verdict assessment has been divided into three phases as shown in Figure 11.

- Phase 1: Requirements description and automatic generation of supporting artefact for modelling.

[18] https://cucumber.io/docs/gherkin/

- Phase 2: Generation of executable test scripts using Model-Based Test scrIpt GenEration fRamework (TIGER).
- Phase 3: Execution of generated test scripts on simulation levels and test verdict assessment.

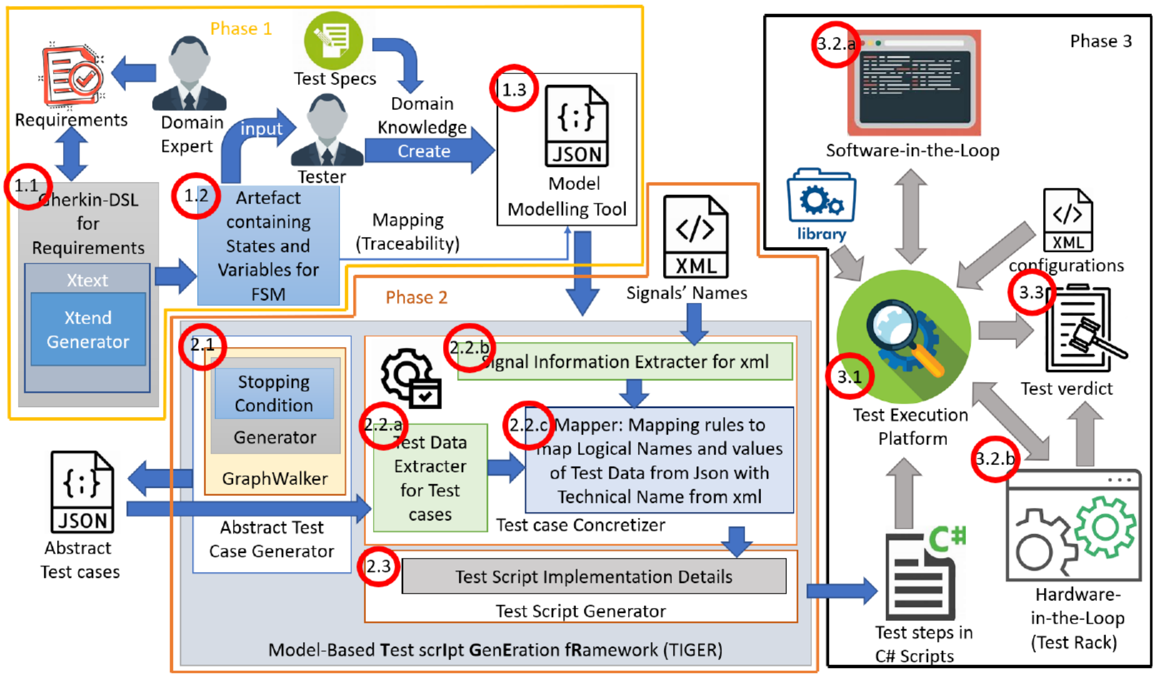


Figure 11 - Sequence of the Test Case Generation

The first phase deals with the modelling aspect of the SUT. To model a SUT, data sources (such as code or specification documents) are required to identify the domain specific entities. In our case, we have the requirements specification as well as the test specification (to include the tester perspective) as an input to the modelling. The requirements specification, in our case, is written in a specific DSL that resembles the Gherkin format. The domain expert specifies these requirements using the Gherkin-like DSL and are used to extract model entities, such as states and transitions, along with variables and their corresponding values involved in the transitions. The result of Phase 1 is an FSM model in JSON or GraphML format, which is in turn used as an input to Phase 2 to generate the executable test scripts.

In order to model the SUT, the first input is a descriptive form of requirement scenarios written in a Gherkin-like DSL. These scenarios have pre-conditions, post-conditions and actions to specify the behaviour of the SUT. In the Gherkin format, the precondition is expressed using the Given keyword, action is specified using the When keyword and outcome is described using the Then keyword. Hence, in the meta-model, the top-level element is the *RequirementSpecification*. The *RequirementSpecification* contains Requirements, each with a unique identifier to validate the atomicity of the requirement. Each requirement contains the definition of Precondition that specifies the initial state of the system, Trigger defining the actions required for an event and state achieved after an event, SystemResponse specifying the state that will be achieved after the response of a system and Time to define the timing constraint for a system to respond.

We have also implemented the Xtend[19] generator to extract the information from the Gherkin-like DSL produced in the Eclipse-based Xtext editor[20]. The Eclipse editor provides multiple built-in features such as highlighting the syntax based on DSL (i.e. preferences for font and colour, style for comments and keywords), predefined templates, an outline view, and assistance for code completion, and error handling. The Xtend generator contains the mapping between each meta-model element of defined DSL and model elements of FSM (i.e. states, transitions etc.). It extracts the information such as model name, state name, transitions, transition variables and their corresponding values from each requirement specified by the requirement engineer in the editor. The validation checks to validate each requirement's atomicity, completeness and unambiguity can be included as an advanced feature in Xtend, which is currently done manually in our case.

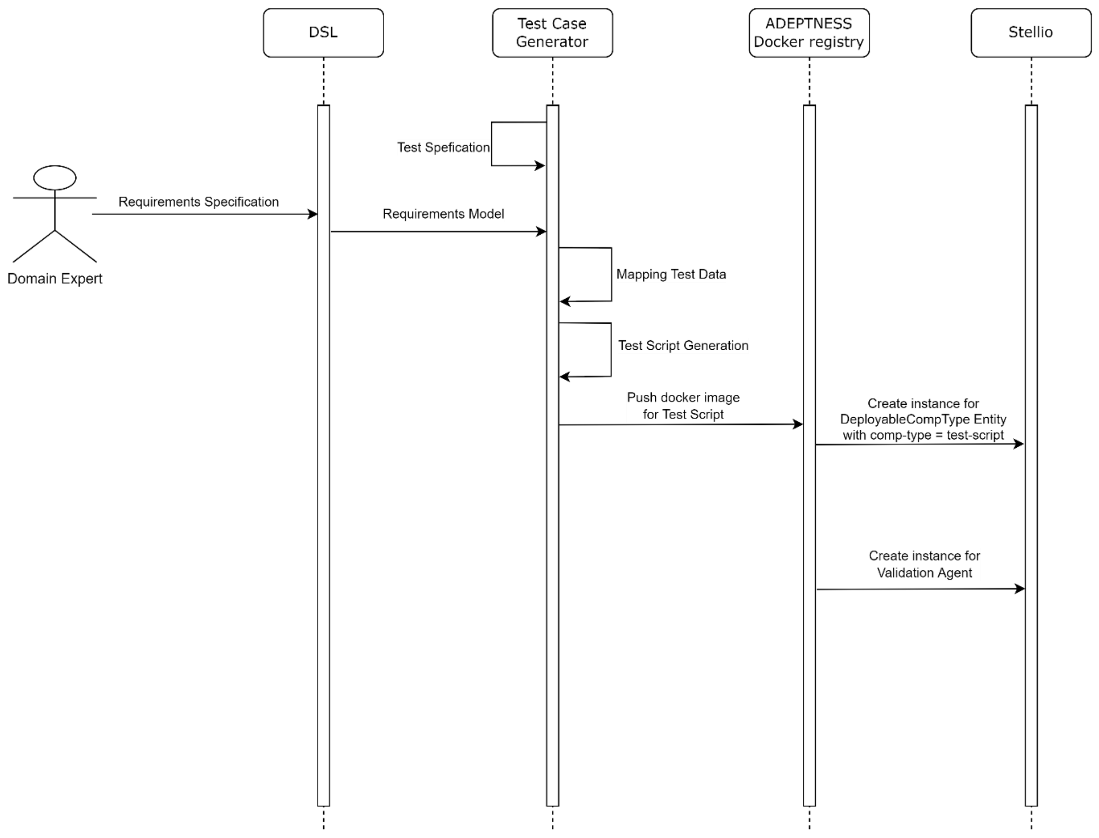

DSL
Test Case Generator
ADEPTNESS Docker registry
Stellio
Test Spefication
Requirements Specification
Domain Expert
Requirements Model
Mapping Test Data
Test Script Generation
Push docker image for Test Script
Create instance for DeployableCompType Entity with comp-type = test-script
Create instance for Validation Agent

Figure 12 - Test Case Generation and Provisioning Workflow

The sequence in Figure 12 provides an overview on the provisioning workflow for Test Case Generation. In the first step, the Test Case Generator is populated using the Test Specification using templates and target SUT configuration. In the second step, the domain expert provides the input for the DSL (Domain Specific Language), in order to formalize the Requirements Model. This model is then sent to the Test Case Generator.

After this, the requirements model and the test specification provide the basis for:

- Mapping the test input data.

[19] https://www.eclipse.org/xtend/documentation/203_xtend_expressions.html

[20] https://www.eclipse.org/Xtext/

- Generate the target test scripts.

The generated scripts and location of input data is then packed into Docker images, and provided to the Docker registry.

4.1.2 DSL-based Oracle Generation & provisioning

A test oracle is the source that determines whether a System Under Test (SUT) behaves as expected. In the Adeptness project, we have developed a Domain Specific Language (DSL) for the specification of test oracles specifically designed for CPSoS. Besides, we have developed a test oracle generator that automatically generates test oracle microservices by considering the specified test oracles with the DSL.

The workflow for generating test oracles works as follows (see Figure 13). The test engineer/validator first specifies the test oracles of the CPSoS under test by using the DSL we have developed. The test oracle generator generates the oracle code based on the specified DSL. These are stored into test oracle DSL files. In a second step, the validator commits and pushes these files to a git repository (in the prototype we have developed we use a GitLab repository, although these could be generalizable to other repositories). After committing and pushing these files, in three completely automated steps we are able to automatically generate the test oracle microservice:

- First, the oracle code is generated in the C programming language. For each of the specified test oracles in the DSL, we generate individual .c and .h files.
- Second, these oracle files are obtained, cross-compiled and integrated with the Adeptness microservice, which is dockerized.
- Third, the test oracle microservice is pushed to the Docker registry and an instance for a deployable component entity is created in Stellio.

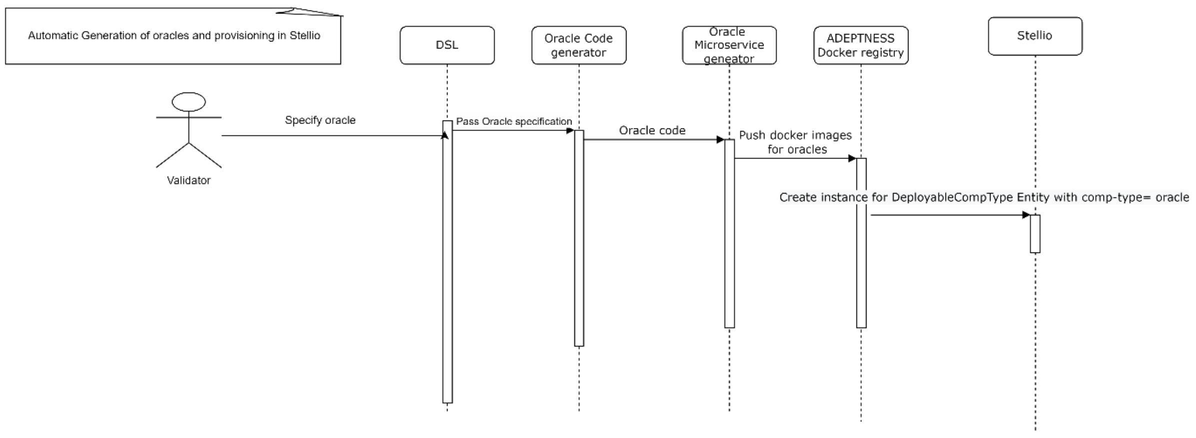

Automatic Generation of oracles and provisioning in Stellio
DSL
Oracle Code generator
Oracle Microservice generator
ADEPTNESS Docker registry
Stellio
Validator
Specify oracle
Pass Oracle specification
Oracle code
Push docker images for oracles
Create instance for DeployableCompType Entity with comp-type= oracle

Figure 13 - DSL-based Oracle Generation Workflow

At plan definition phase, the registered test oracles can be then used for the Adeptness plan, and at runtime, the registered oracles (in Docker registry and Stellio logger) can be downloaded and instantiated on edge nodes.

4.2 Plans definition

The provisioning workflow populates Stellio with the components and microservices that will be used during the execution of validations. This implies create instances for the types of entities defined in NGSI_LD data model for Adeptness in Stellio.

The plan definition has the folowingkey steps (see):

- Context Provisioning: The test scenario on which the plans are to be executed must be previously defined. This consists of describing each of the SCPs in the CPSoS. Each CPS shall define the variables it is able to provide and its network characteristics. The locations of the software artefacts to be used must also be defined...
- Validation Plan: Provisioning of the agents, oracles and external tools, including test inputs, responsible for validating the SUT.
- Monitoring Plan: Provisioning of the monitoring agents that are able to obtain and to publish the values of the variables needed by the testcases.
- Deployment Plan: Provisioning of the deployment plan. This plan includes the list of all components and the node where to deploy them as well as the deployment parameters.
- Adeptness Plan: This step includes the aggregation of the deployment, monitoring and validation plans to the Adeptness plan.

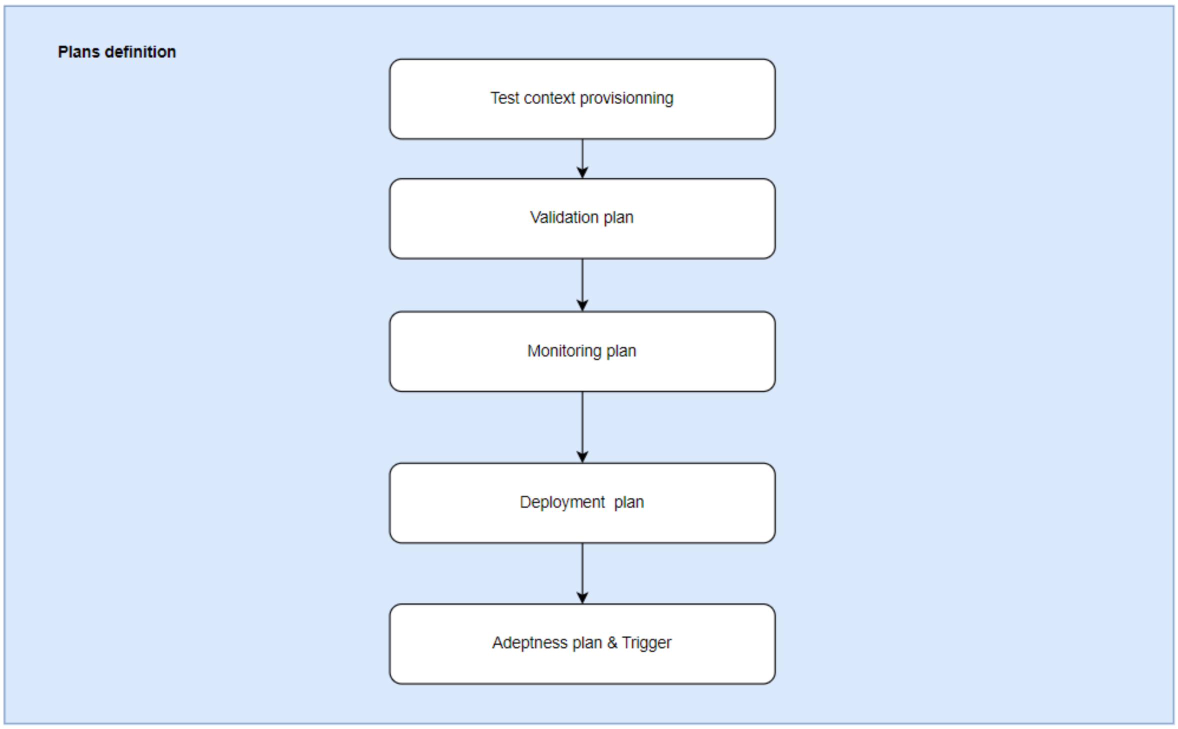


Figure 14 - Planning Workflow

The following subsections provide details about the sequences in each of the plan definition steps.

4.2.1 Context Provisioning

In our data driven approach, all components must be configured, registered, and sent to the Logger, our context broker (Stellio) which handles context management, time and space, in an Entities – Relationships – Properties graph model. For this purpose, the TaaS front-end, as a "data producer & consumer" communicates with Stellio in a Restful way to handle the provisioning (see D5.1 for more in depth description of the provisioning phase).

4.2.1.1 CPSoS Definition

The features of the CPSoS that are going to be validated shall be provisioned in Stellio. For each CPS, the monitoring variables that can be observed are also provisioned. These variables are named Sensors. The hardware nodes where CPSs and Adeptness deployable components are running shall also be provisioned.

For each deployable component type, the model contains the associated references to the compressed file location or container image registry, and the kind of component, such as a monitor, oracle, validation agent, SUT, external tool or test input.

In a first step, the system engineers shall declare the target CPSoS, by defining:

- The CPSoS itself
- The CPS contained in the CPSoS.
- The Sensors or Monitoring Variables of CPS. These can be grouped into Sensor Groups.
- The Nodes, this is, the logical entity providing CPS network information.

In the following subsections, we provide the details on how the context broker in Adeptness is provisioned.

CPSoS

The first step is to define the logical context of validation. For that reason, the system engineer shall specify a logical entity "cyber-physical system of systems", which groups all the cyber-physical systems under validation. A unique descriptive name and a short description shall be provided.

The sequence diagram is depicted in Figure 15.

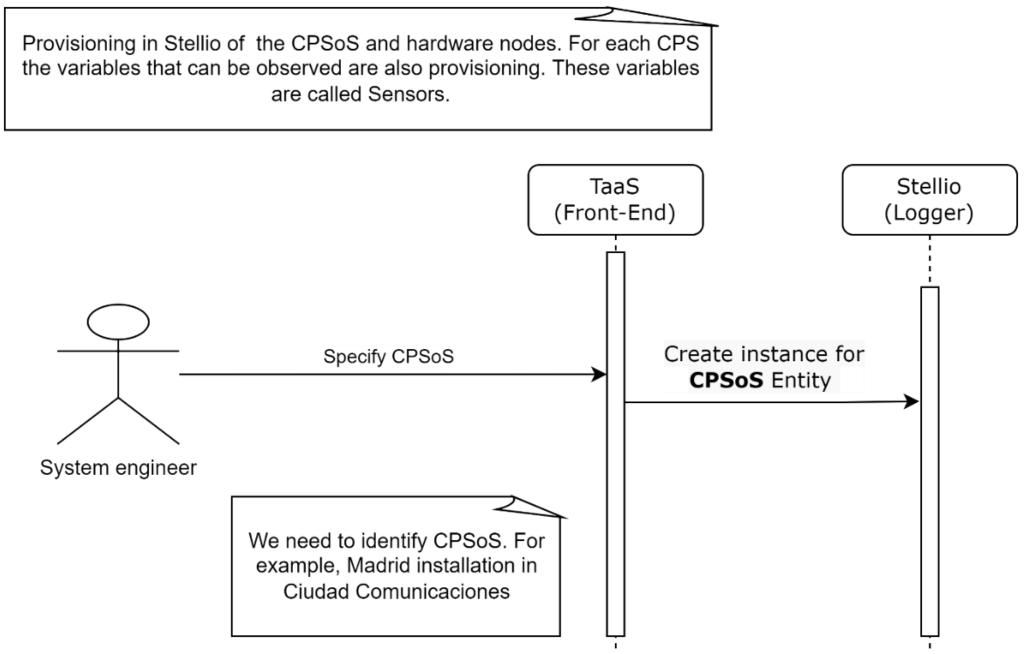
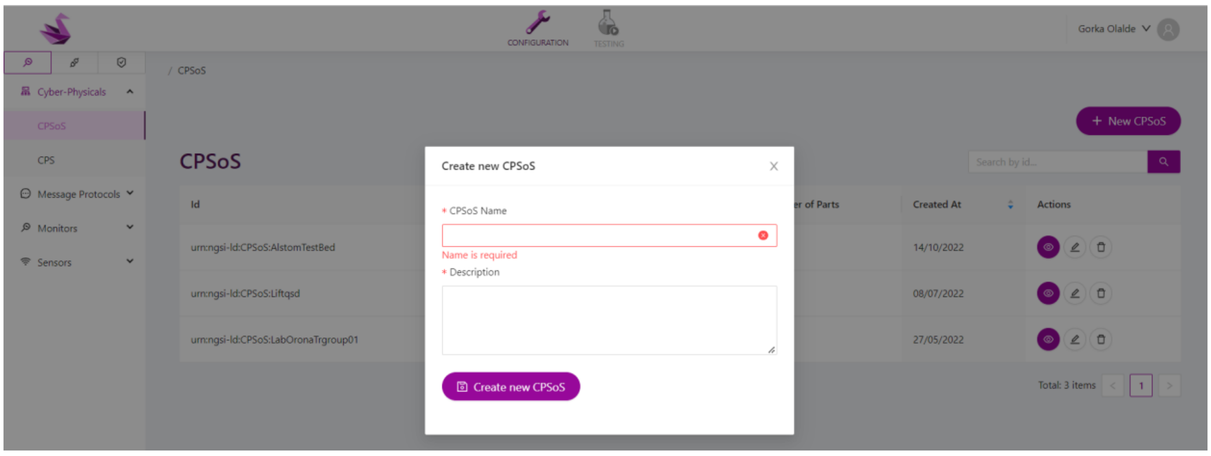

Provisioning in Stellio of the CPSoS and hardware nodes. For each CPS the variables that can be observed are also provisioning. These variables are called Sensors.
TaaS (Front-End)
Stellio (Logger)
Specify CPSoS
Create instance for **CPSoS** Entity
System engineer
We need to identify CPSoS. For example, Madrid installation in Ciudad Comunicaciones

Figure 15 - Sequence diagram for provisioning CPSoS in Stellio

TaaS implements the above sequence diagram. Figure 16 displays the screen in TaaS that allows the provisioning of CPSoS.



CPSoS

Id	er of Parts	Created At	Actions
urn:ngsi-ld:CPSoS:AlstomTestBed		14/10/2022	
urn:ngsi-ld:CPSoS:Liftqsd		08/07/2022	
urn:ngsi-ld:CPSoS:LabOronaTrgroup01		27/05/2022	

Total: 3 items 1

Create new CPSoS

* CPSoS Name

Name is required

* Description

Create new CPSoS

Figure 16 - TaaS screen to provision a CPSoS

CPS

The system engineer shall declare the cyber-physical systems (CPS), which are defined at the CPSoS. For specifying a new CPS, the user should first select to which CPSoS attaches the CPS being defined. After that, a unique descriptive name for CPS and a short description shall be provided. The sequence diagram for the CPS provision can be seen in Figure 17.

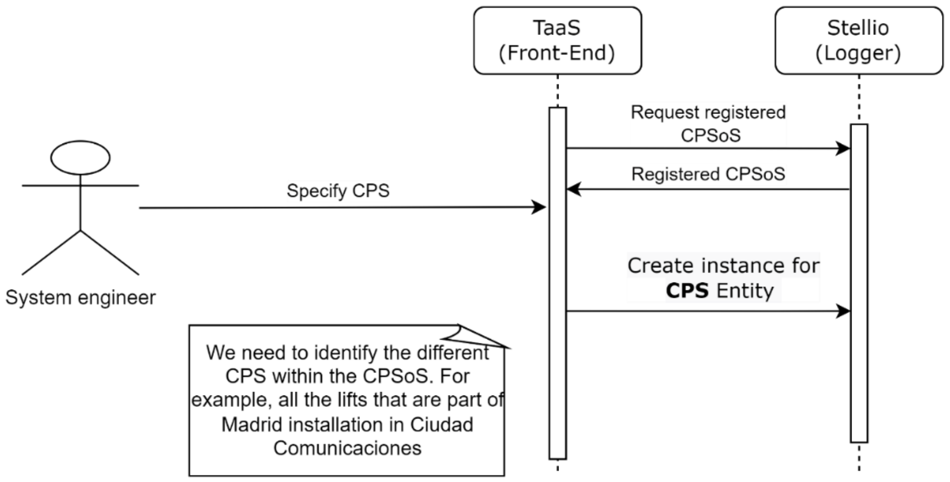
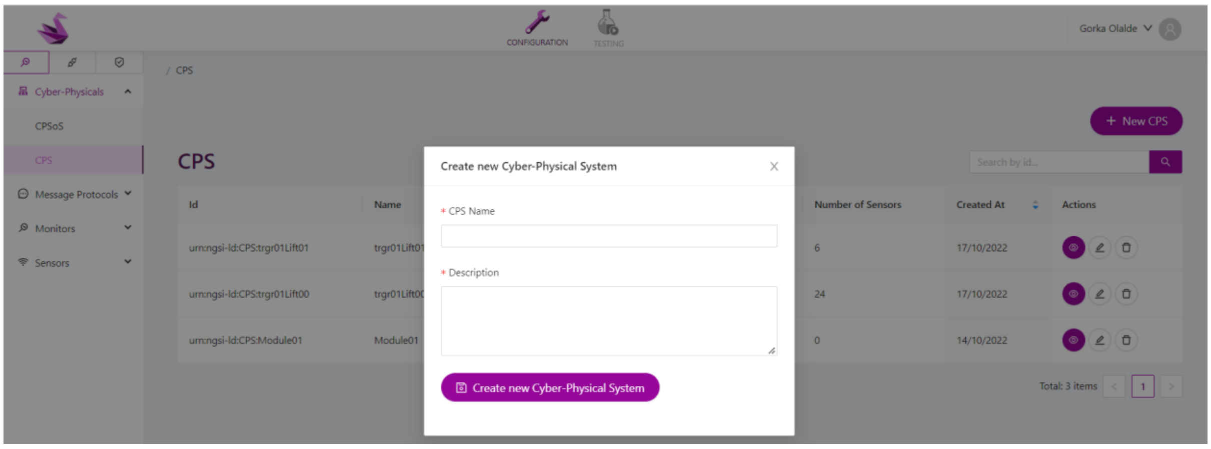

TaaS (Front-End)
Stellio (Logger)
Request registered CPSoS
Registered CPSoS
Specify CPS
Create instance for **CPS** Entity
System engineer
We need to identify the different CPS within the CPSoS. For example, all the lifts that are part of Madrid installation in Ciudad Comunicaciones

Figure 17 - Sequence diagram for provisioning CPS in Stellio

TaaS implements the above sequence diagram. Figure 18 displays the screen in TaaS that allows the provisioning of CPS.



CPS

Id	Name	Number of Sensors	Created At	Actions
urn:ngsi-ld:CPS:trgr01Lift01	trgr01Lift01	6	17/10/2022	
urn:ngsi-ld:CPS:trgr01Lift00	trgr01Lift0	24	17/10/2022	
urn:ngsi-ld:CPS:Module01	Module01	0	14/10/2022	

Total: 3 items

* CPS Name

* Description

Figure 18 - TaaS screen to provision a new CPS

Sensors

A **sensor** in Adeptness represents a monitoring variable for a given device. In our case, this is specifically a monitoring variable on the CPS. A generic sensor shall contain, at least the unit, the sampling rate and a communication protocol, which will be used at runtime to transfer collected data to the Adeptness ecosystem.

The user shall first select the CPS to which the sensor is attached. Every sensor is allocated to a single CPS. specify a new sensor, selecting the CPS to which will be attached. Additionally, the user should provide a unique sensor descriptive name, a unit, a sampling rate, and a protocol. Depending on the selected protocol, the user shall specify protocol-specific attributes to know how to extract the values of the variables from the underlying protocol. Later on, in the creation of the monitoring plan, the sensor shall be associated to one of the defined monitoring agents, which are the way of notifying to Stellio at runtime the sampled data.

Figure 19 displays graphically the sequence diagram described above.

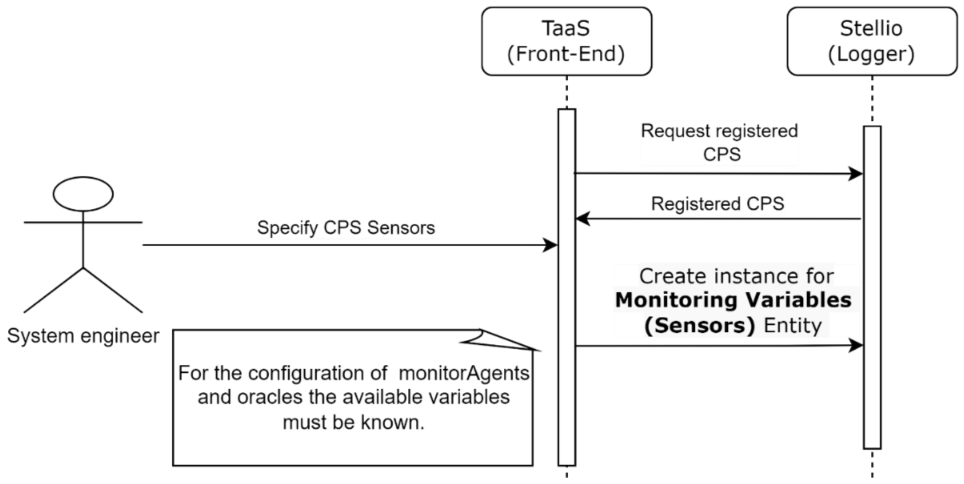
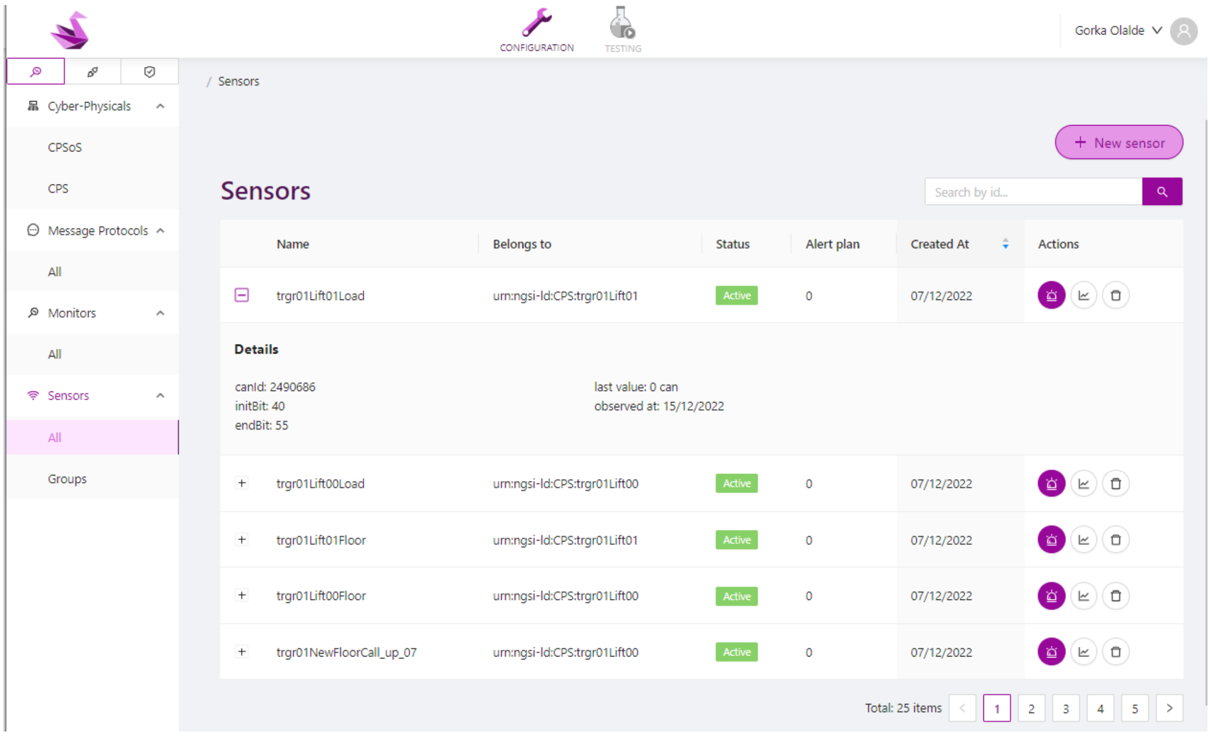

TaaS
(Front-End)
Stellio
(Logger)
Request registered CPS
Registered CPS
Specify CPS Sensors
Create instance for **Monitoring Variables (Sensors)** Entity
System engineer
For the configuration of monitorAgents and oracles the available variables must be known.

Figure 19 - Sequence diagram for provisioning Sensors in Stellio

The sensors can be grouped intro Sensor Groups. This allows publishing the value of a group of sensors at the same publishing rate. The groups shall be defined after at least one Sensor is defined. TaaS implements the above sequence diagram. Figure 20 and Figure 21 display the screen in TaaS that allows the provisioning of sensors and sensor groups.



Sensors

	Name	Belongs to	Status	Alert plan	Created At	Actions
−	trgr01Lift01Load	urn:ngsi-ld:CPS:trgr01Lift01	Active	0	07/12/2022	

Details

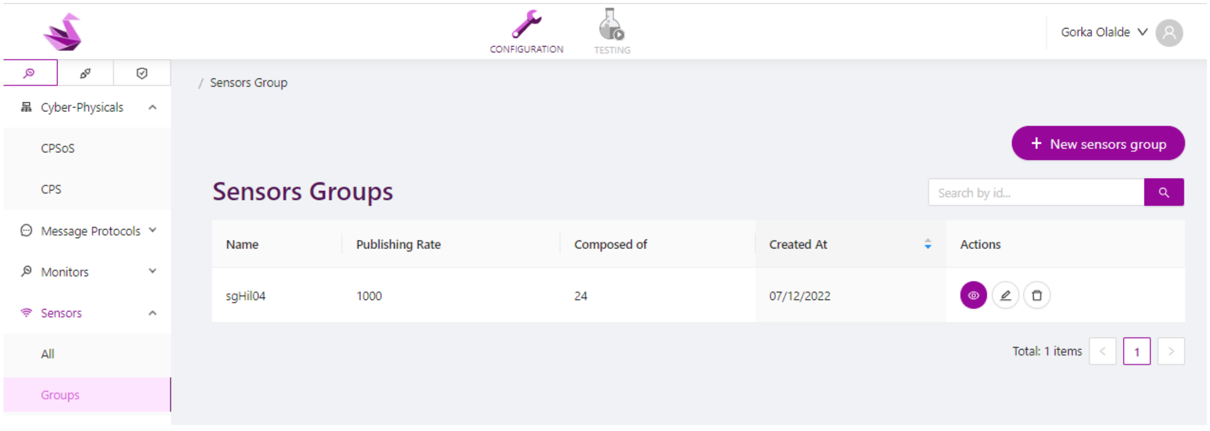
canId: 2490686
initBit: 40
endBit: 55

last value: 0 can
observed at: 15/12/2022

	Name	Belongs to	Status	Alert plan	Created At	Actions
+	trgr01Lift00Load	urn:ngsi-ld:CPS:trgr01Lift00	Active	0	07/12/2022	
+	trgr01Lift01Floor	urn:ngsi-ld:CPS:trgr01Lift01	Active	0	07/12/2022	
+	trgr01Lift00Floor	urn:ngsi-ld:CPS:trgr01Lift00	Active	0	07/12/2022	
+	trgr01NewFloorCall_up_07	urn:ngsi-ld:CPS:trgr01Lift00	Active	0	07/12/2022	

Total: 25 items < 1 2 3 4 5 >

Figure 20 - TaaS screen to display all Sensors provisioned and to provision a new one



Sensors Groups

Name	Publishing Rate	Composed of	Created At	Actions
sgHil04	1000	24	07/12/2022	

Total: 1 items

Figure 21 - TaaS screen to display all SensorGroups provisioned and to provision a new one

Hardware Nodes

The logical entity Node represents the network features of the CPS. Figure 22 provides the sequence diagram for creating the hardware nodes.

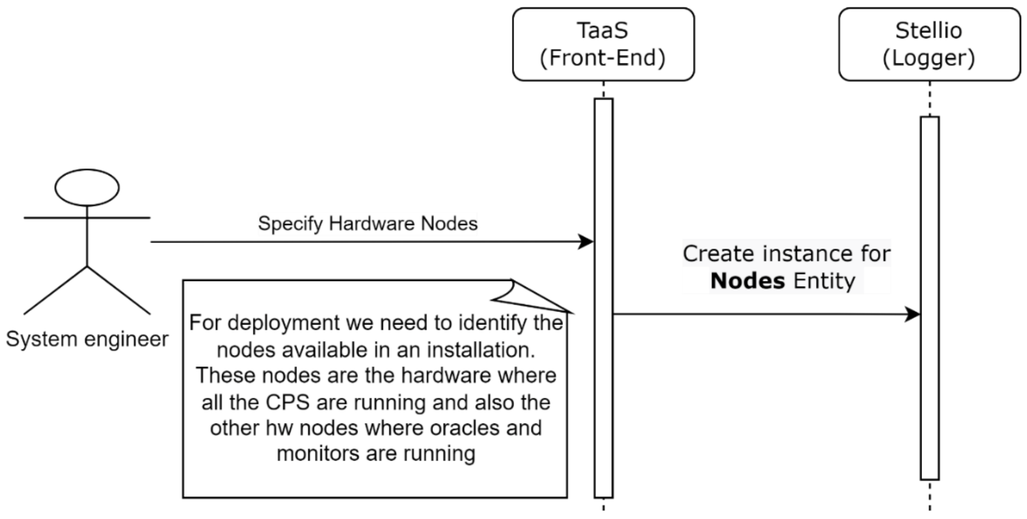
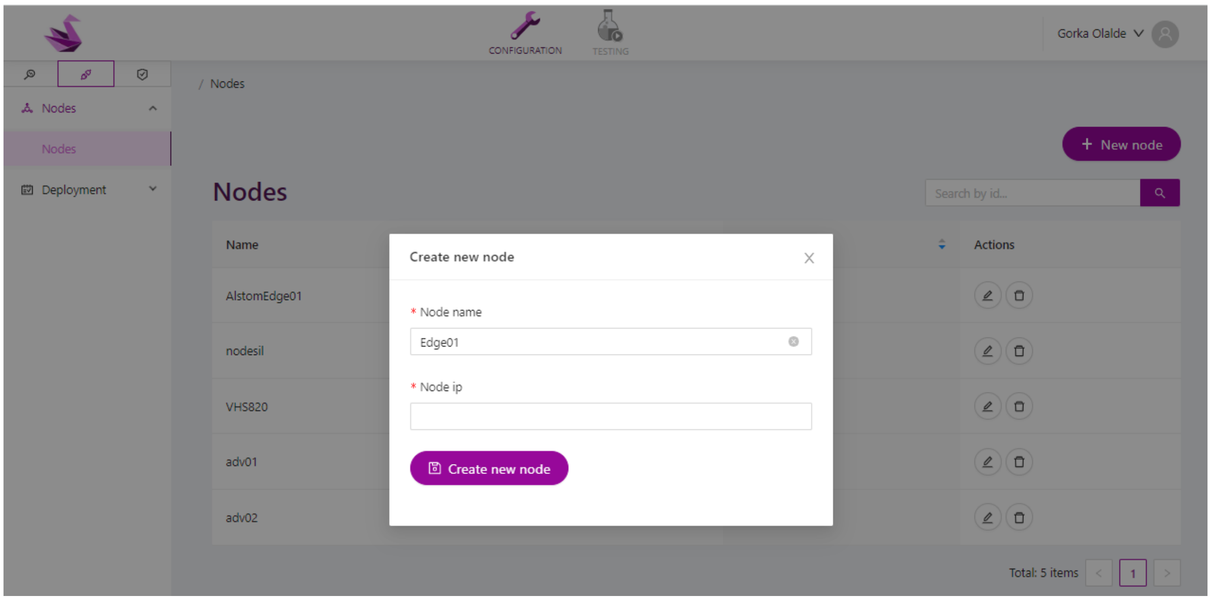

TaaS (Front-End)
Stellio (Logger)
System engineer
Specify Hardware Nodes
Create instance for **Nodes** Entity
For deployment we need to identify the nodes available in an installation. These nodes are the hardware where all the CPS are running and also the other hw nodes where oracles and monitors are running

Figure 22 - Sequence diagram for provisioning Nodes in Stellio

TaaS implements the above sequence diagram. Figure 23 displays the screen in TaaS that allows the provisioning of Nodes.



Nodes

Name
AlstomEdge01
nodesil
VHS820
adv01
adv02

Create new node

* Node name

Edge01

* Node ip

Create new node

Figure 23 - TaaS screen to provision a new node

4.2.1.1 *Deployable components specification*

The next logical step is the specification of deployable components in Stellio through TaaS. When provisioning a deployable component, the type of the component shall be provided. The available deployable component types in Adeptness are:

- System Under Test (SUT).
- Oracles
- Monitor Agents.
- External Tools.
- Test Inputs
- Validation Agents

The System Under Test (SUT) represents the target validation entity. For our use case, SUT is a deployable component, which is the validation subject of the validation plan. This can be some software component attached to the physical system, such as an executable,a library, an algorithm, or any element likely to be validated.

The Oracles represent an entity, which evaluate some conditions at validation time and trigger evaluation verdicts, representing whether the conditions at testing are met. The validation agent collects, at validation completion, the associated oracles verdicts. The implementation and logic underlying to Oracles is automatically generated using DSL2Oracle microservice (see Section 3.2.2), and provisioned to Stellio using the TaaS front-end specification available in Section 3.1.4.2.

The Monitor Agents deployable components are the implementation of the monitors in charge of reading the variables. This agents are compliance with the Adeptness interfaces and are ready to be configured with the suitable variables (sensors) during the execution of the monitoring plan. These are specified when provisioning the monitoring plan in Stellio.

The External Tools allow to integrate domain-specific or legacy tools into the Adeptness ecosystem. Test Inputs are used to stimulate the SUT or help on the execution of test cases. The Validation Agents are the software component in charge of the oracles and external tool configuration, start of oracles execution and the implementation of the oracle verdict aggregation.

Figure 24 provides the sequence diagram for creating the deployable components

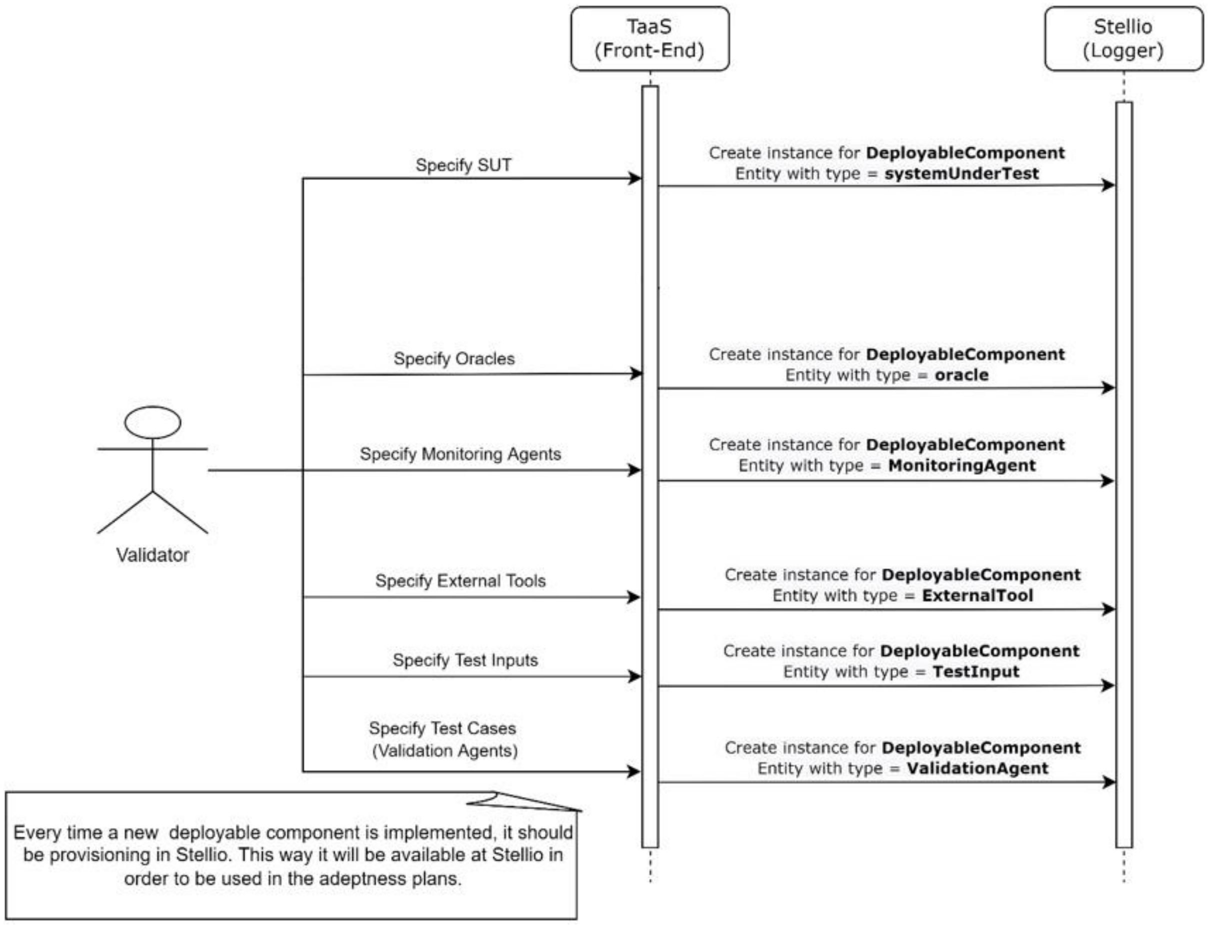
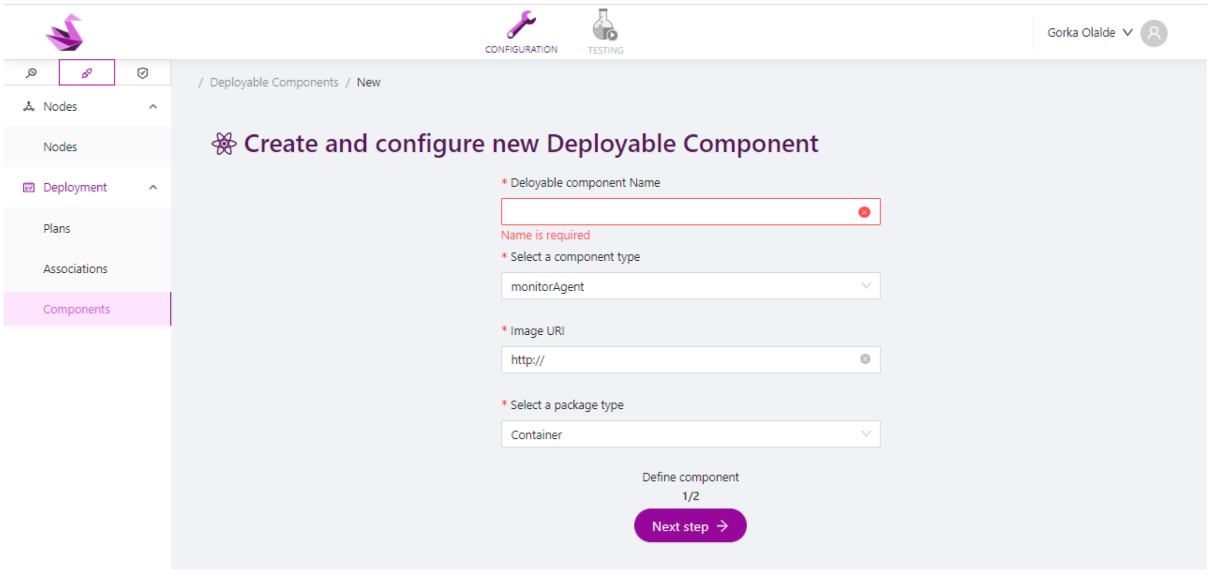

TaaS (Front-End)
Stellio (Logger)
Validator
Specify SUT
Create instance for **DeployableComponent** Entity with type = **systemUnderTest**
Specify Oracles
Create instance for **DeployableComponent** Entity with type = **oracle**
Specify Monitoring Agents
Create instance for **DeployableComponent** Entity with type = **MonitoringAgent**
Specify External Tools
Create instance for **DeployableComponent** Entity with type = **ExternalTool**
Specify Test Inputs
Create instance for **DeployableComponent** Entity with type = **TestInput**
Specify Test Cases (Validation Agents)
Create instance for **DeployableComponent** Entity with type = **ValidationAgent**
Every time a new deployable component is implemented, it should be provisioning in Stellio. This way it will be available at Stellio in order to be used in the adeptness plans.

Figure 24 - Specification of deployable component in Stellio

TaaS implements the above sequence diagram. Figure 25 displays the screen in TaaS that allows the provisioning of DeployableComponent.



Create and configure new Deployable Component

* Deloyable component Name

Name is required

* Select a component type: monitorAgent

* Image URI: http://

* Select a package type: Container

Define component 1/2

Figure 25 - TaaS screen to provision a new DeployableComponent

4.2.2 Creation of the Deployment Plan

The deployment plan creation with the TaaS shall be done in two steps: first definition of the deployment associations and second deployment plan creation.

Deployment Association.

The deployment associations are simply the relationship between nodes and deployable components instances, in any of the available deployable component types: monitoring agents, SUTs, external tool agents, validation agents, oracles or test inputs.

Next figure shows a high-level sequence diagram for the creation of a deployment association.

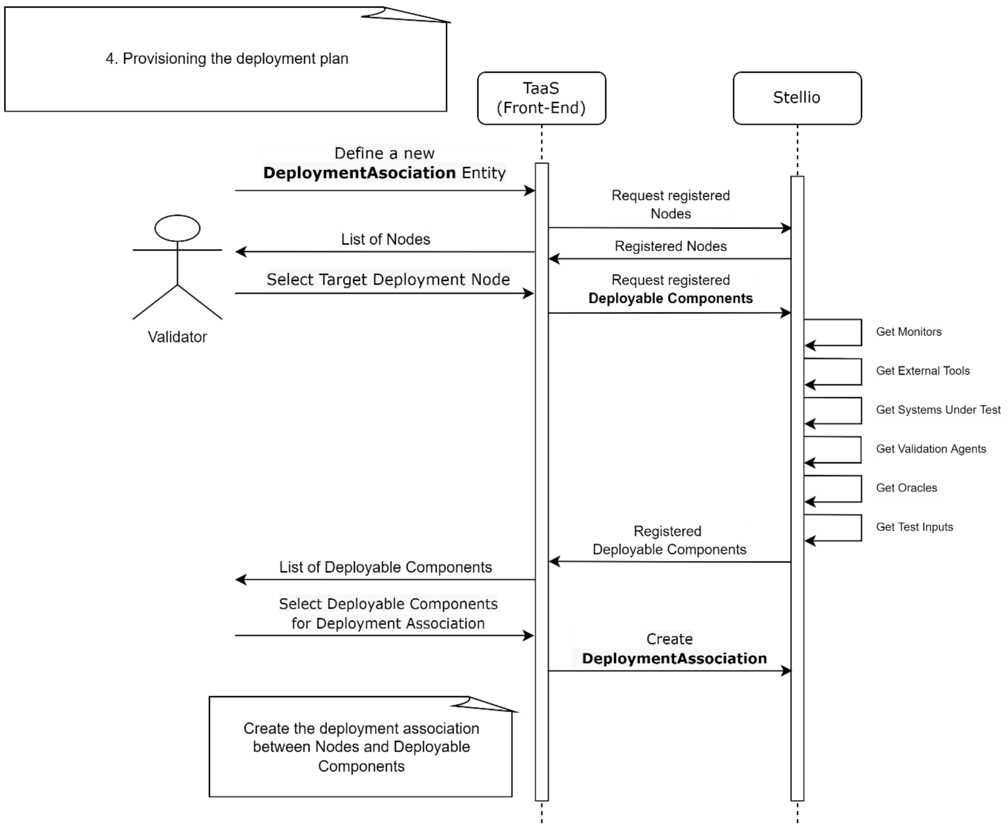
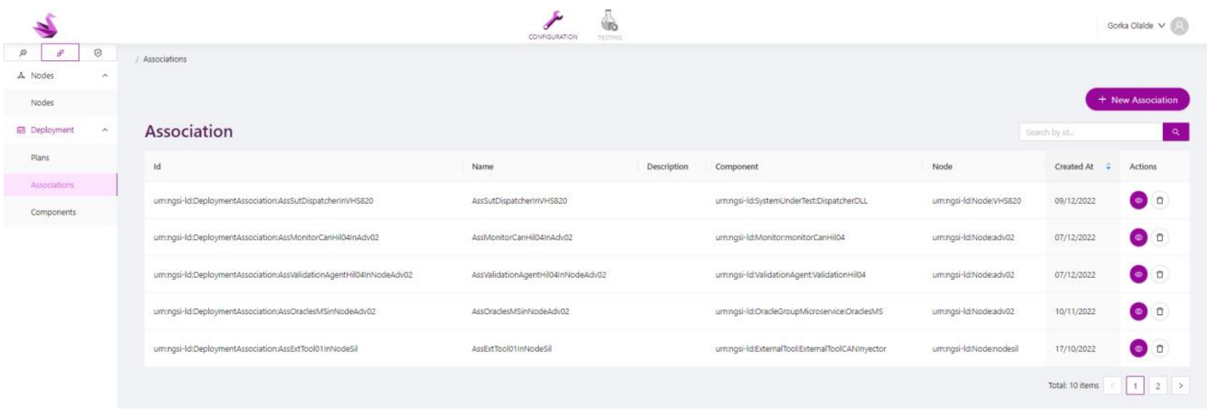

4. Provisioning the deployment plan
TaaS (Front-End)
Stellio
Validator
Define a new DeploymentAsociation Entity
Request registered Nodes
List of Nodes
Registered Nodes
Select Target Deployment Node
Request registered Deployable Components
Get Monitors
Get External Tools
Get Systems Under Test
Get Validation Agents
Get Oracles
Get Test Inputs
Registered Deployable Components
List of Deployable Components
Select Deployable Components for Deployment Association
Create DeploymentAssociation
Create the deployment association between Nodes and Deployable Components

Figure 26 - Deployment association creation sequence diagram

The user (in this case, the validator) using TaaS requests the edge nodes defined in the system and select one of them. Afterwards, the user selects the available deployable components specified in the system. These deployable components can be either monitors, external tools, SUTs, validation agents, oracles or test inputs. Only one deployable component and the node where it is deployed are part of the deployment association.

Figure 27 displays the screen in TaaS that allows the provisioning of a deployment asociation.



Association

Id	Name	Description	Component	Node	Created At	Actions
urn:ngsi-ld:DeploymentAssociation:AssSutDispatcherInVHS820	AssSutDispatcherInVHS820		urn:ngsi-ld:SystemUnderTest:DispatcherDLL	urn:ngsi-ld:Node:VHS820	09/12/2022	
urn:ngsi-ld:DeploymentAssociation:AssMonitorCanHil04InAdv02	AssMonitorCanHil04InAdv02		urn:ngsi-ld:Monitor:monitorCanHil04	urn:ngsi-ld:Node:adv02	07/12/2022	
urn:ngsi-ld:DeploymentAssociation:AssValidationAgentHil04InNodeAdv02	AssValidationAgentHil04InNodeAdv02		urn:ngsi-ld:ValidationAgent:ValidationHil04	urn:ngsi-ld:Node:adv02	07/12/2022	
urn:ngsi-ld:DeploymentAssociation:AssOraclesMSInNodeAdv02	AssOraclesMSInNodeAdv02		urn:ngsi-ld:OracleGroupMicroservice:OraclesMS	urn:ngsi-ld:Node:adv02	10/11/2022	
urn:ngsi-ld:DeploymentAssociation:AssExtTool01InNodeSil	AssExtTool01InNodeSil		urn:ngsi-ld:ExternalTool:ExternalToolCANInyector	urn:ngsi-ld:Node:nodesil	17/10/2022	

Total: 10 items < 1 2 >

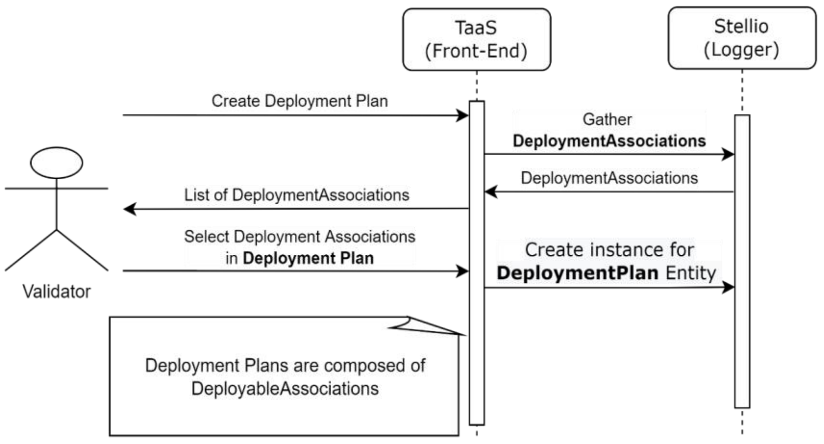
Figure 27 - TaaS screen to provision a new DeploymentAssociation

The user can create as many associations as needed, to cover the targeted deployment plan, which is explained in the following subsection.

Deployment Plan

A deployment plan is a container of deployment associations. The specification of deployment plans consists only on selection previously defined DeploymentAssociations entities. A deployment plan can store as many deployment associations as defined in the system, containing the deployable components instances required for the deployment at validation targets. This can comprehend one or more edge nodes.

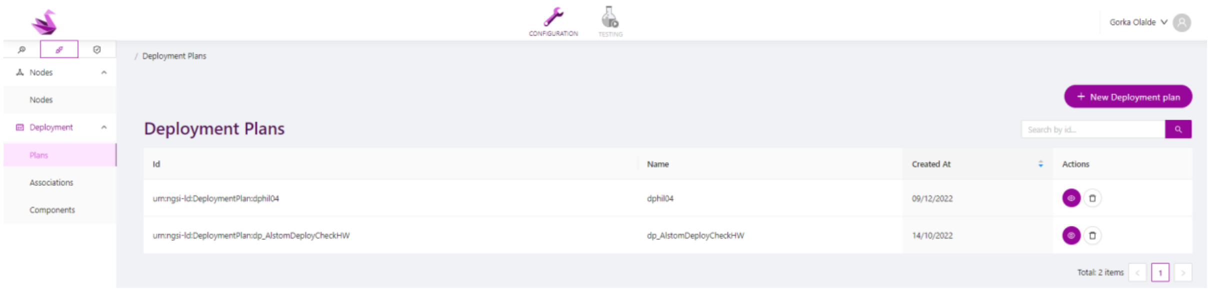
Next figure shows a high-level sequence diagram for the creation of a deployment plan:


TaaS (Front-End)
Stellio (Logger)
Create Deployment Plan
Gather **DeploymentAssociations**
DeploymentAssociations
List of DeploymentAssociations
Select Deployment Associations in **Deployment Plan**
Create instance for **DeploymentPlan** Entity
Validator
Deployment Plans are composed of DeployableAssociations

Figrue 28 - Deployment plan creation sequence diagram

Deployment Plans are generated from a list of associations between deployable components instances and Nodes. TaaS will climb up the entity graph tree thanks to relationships and retrieve all related entities, such as sensor groups, sensors, CPS and CPSoS. Once all data has been gathered, JSON files for a Deployment plan and Monitoring plan are built together since they are closely intertwined (they can then be downloaded or be used to trigger an Adeptness Plan and a testing phase).

For the definition of a new deployment plan, the user requests the creation of a new deployment plan using TaaS interface. The user shall provide at least the Deployment plan name, and a description. Figure 29 displays the screen in TaaS that allows the provisioning of a deployment plan.

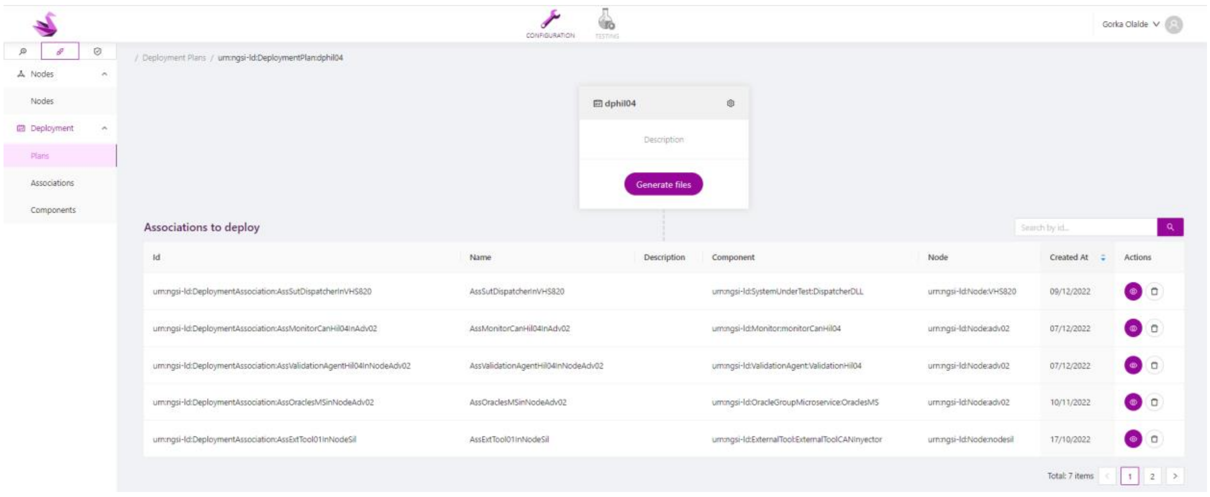


Deployment Plans

Id	Name	Created At	Actions
urn:ngsi-ld:DeploymentPlan:dphil04	dphil04	09/12/2022	
urn:ngsi-ld:DeploymentPlan:dp_AlstomDeployCheckHW	dp_AlstomDeployCheckHW	14/10/2022	

Figure 29 - TaaS screen to provision a new DeploymentPlan

In a second step, the user can select the set of deployment associations, which will be included in the plan. Figure 30 displays the screen in TaaS that shows the deployment associations contained in a deployment plan.



/ Deployment Plans / urn:ngsi-ld:DeploymentPlan:dphil04

Associations to deploy

Id	Name	Description	Component	Node	Created At	Actions
urn:ngsi-ld:DeploymentAssociation:AssSutDispatcherInVHS820	AssSutDispatcherInVHS820		urn:ngsi-ld:SystemUnderTest:DispatcherDLL	urn:ngsi-ld:Node:VHS820	09/12/2022	
urn:ngsi-ld:DeploymentAssociation:AssMonitorCanHil04InAdv02	AssMonitorCanHil04InAdv02		urn:ngsi-ld:Monitor:monitorCanHil04	urn:ngsi-ld:Node:adv02	07/12/2022	
urn:ngsi-ld:DeploymentAssociation:AssValidationAgentHil04InNodeAdv02	AssValidationAgentHil04InNodeAdv02		urn:ngsi-ld:ValidationAgent:ValidationHil04	urn:ngsi-ld:Node:adv02	07/12/2022	
urn:ngsi-ld:DeploymentAssociation:AssOraclesMSinNodeAdv02	AssOraclesMSinNodeAdv02		urn:ngsi-ld:OracleGroupMicroservice:OraclesMS	urn:ngsi-ld:Node:adv02	10/11/2022	
urn:ngsi-ld:DeploymentAssociation:AssExtTool01InNodeSil	AssExtTool01InNodeSil		urn:ngsi-ld:ExternalTool:ExternalToolCANInyector	urn:ngsi-ld:Node:nodesil	17/10/2022	

Total: 7 items 1 2

Figure 30 - TaaS screen to show the content of a DeploymentPlan

TaaS will be able with the deployment plan information to automatically generate a JSON file compliance with the schema expected by the deployment orchestrator and well defined in the deployment orchestrator interface.

4.2.3 *Creation of the Monitoring Plan*

In order to generate a monitoring plan, you need to have previously created and configured the monitors to be used in the plan. The monitoring plan itself is not instantiated. TaaS will automatically generate this plan with the information contained in the monitors that are part of a deployment plan. When configuring a monitor, the binary artefact, the rest port where the monitor will provide its Rest API as well as the variables to be observed (from those offered by the CPS) are selected. Figure 31 contains the sequence diagram to create a monitor.

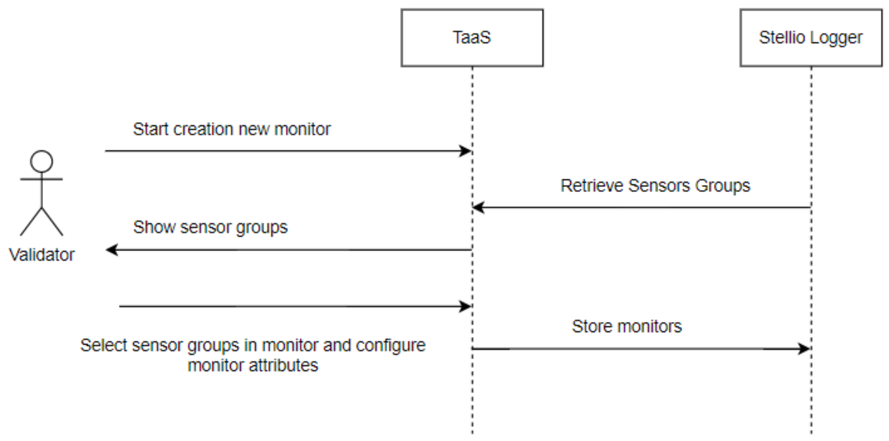
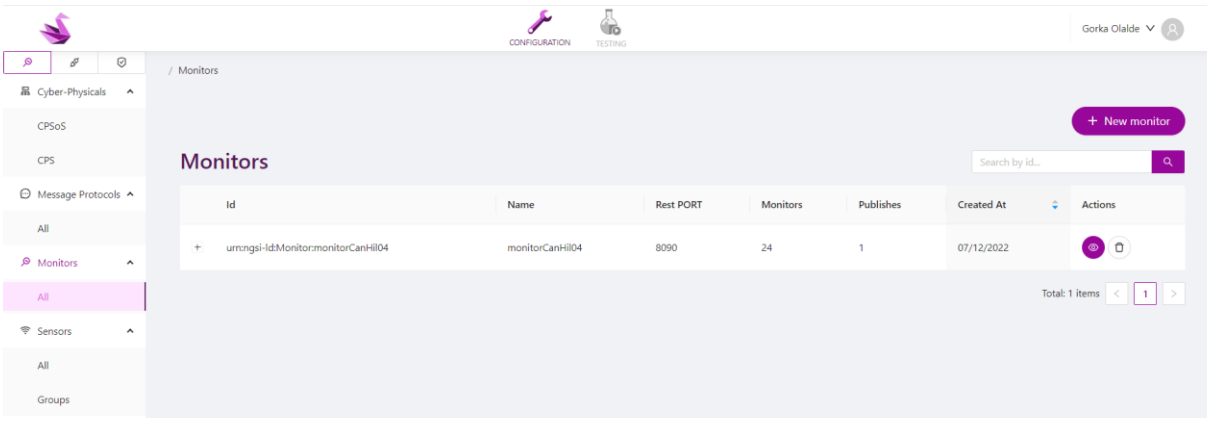

TaaS
Stellio Logger
Validator
Start creation new monitor
Retrieve Sensors Groups
Show sensor groups
Select sensor groups in monitor and configure monitor attributes
Store monitors

Figure 31 - Monitoring plan configuration workflow

Figure 32 displays the screen in TaaS that allows to create a Monitor.



/ Monitors

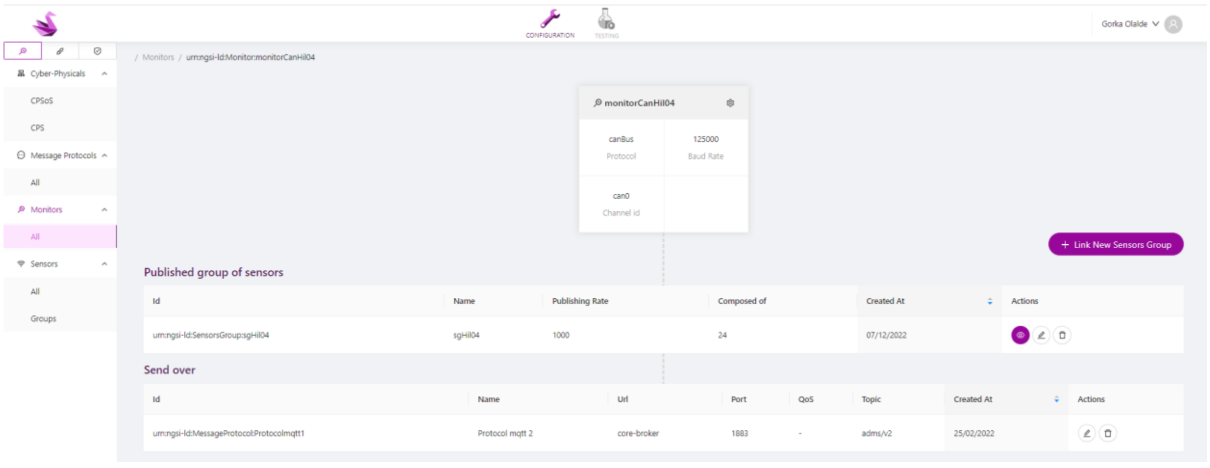
Monitors

	Id	Name	Rest PORT	Monitors	Publishes	Created At	Actions
+	urn:ngsi-ld:Monitor:monitorCanHil04	monitorCanHil04	8090	24	1	07/12/2022	

Total: 1 items 1

Figure 32 - TaaS screen to provision a new monitor

Figure 33 displays a screen in TaaS with a monitor configuration.



/ Monitors / urn:ngsi-ld:Monitor:monitorCanHil04

monitorCanHil04

canBus
Protocol

125000
Baud Rate

can0
Channel id

Published group of sensors

Id	Name	Publishing Rate	Composed of	Created At	Actions
urn:ngsi-ld:SensorsGroup:sgHil04	sgHil04	1000	24	07/12/2022	

Send over

Id	Name	Url	Port	QoS	Topic	Created At	Actions
urn:ngsi-ld:MessageProtocol:Protocolmqtt1	Protocol mqtt 2	core-broker	1883	-	adms/v2	25/02/2022	

Figure 33 - TaaS screen to show the monitor configuration

TaaS will be able with the deployment plan information to automatically generate a JSON file compliance with the schema expected by the monitoring orchestrator and well defined in the monitoring orchestrator interface.

4.2.4 Creation of the Validation Plan

The validation plan is targeted to run a set of test cases, previously defined, which focuses on the functionality, capabilities and constraints, which apply to the SUT. The sequence to create a validation plan is depicted in Figure 34.

Essentially, the Validation Plan makes use of the Stellio registered components, to carry out a validation strategy through the Validation Agents as the core component executing the defined Test Cases. For doing this, the Validation Agent can make use of the External Tools, Test Inputs, and Oracles to generate the evaluation results.

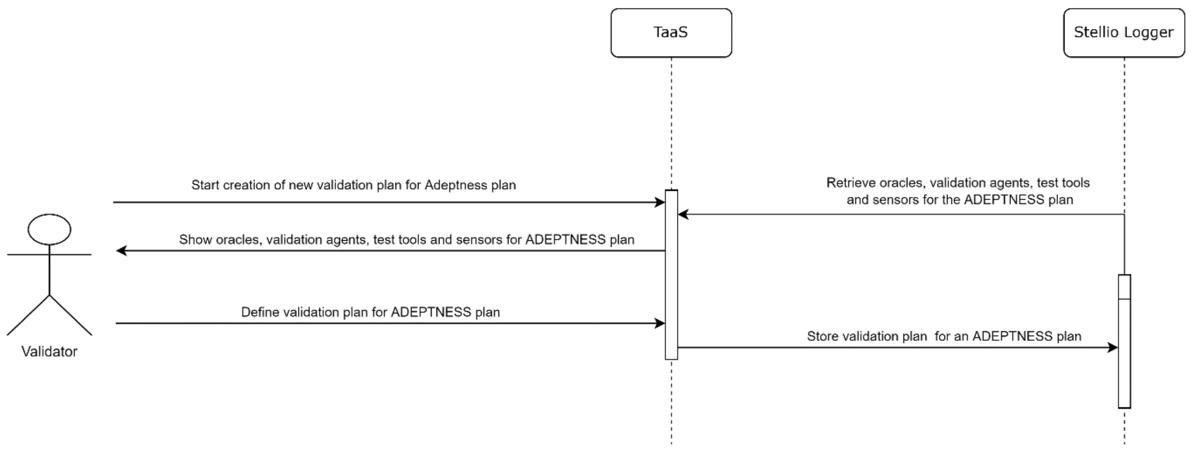
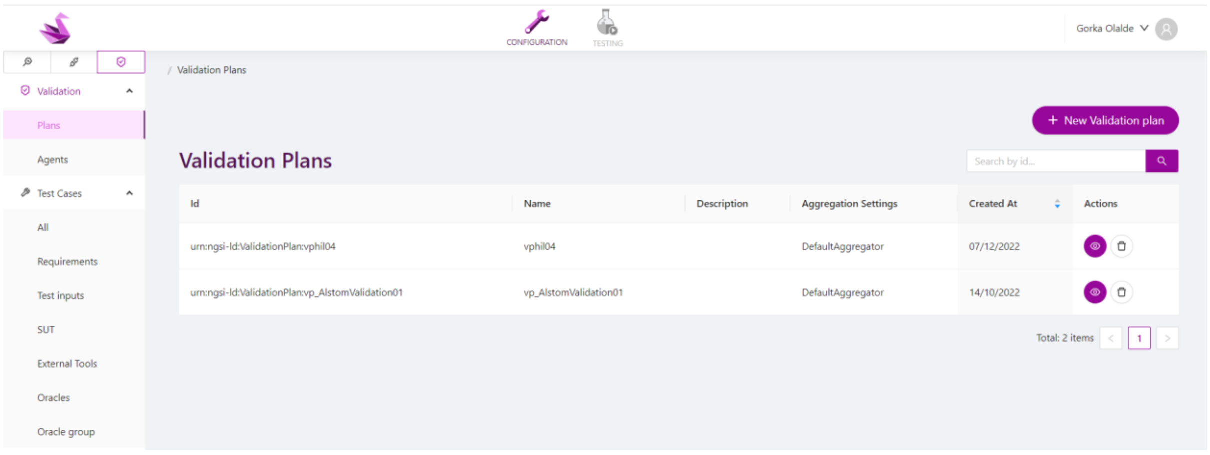

TaaS
Stellio Logger
Validator
Start creation of new validation plan for Adeptness plan
Retrieve oracles, validation agents, test tools and sensors for the ADEPTNESS plan
Show oracles, validation agents, test tools and sensors for ADEPTNESS plan
Define validation plan for ADEPTNESS plan
Store validation plan for an ADEPTNESS plan

Figure 34 - Validation plan configuration workflow

Figure 35 displays the screen in TaaS that allows the provisioning of a validation plan.



Validation Plans

Id	Name	Description	Aggregation Settings	Created At	Actions
urn:ngsi-ld:ValidationPlan:vphil04	vphil04		DefaultAggregator	07/12/2022	
urn:ngsi-ld:ValidationPlan:vp_AlstomValidation01	vp_AlstomValidation01		DefaultAggregator	14/10/2022	

Total: 2 items 1

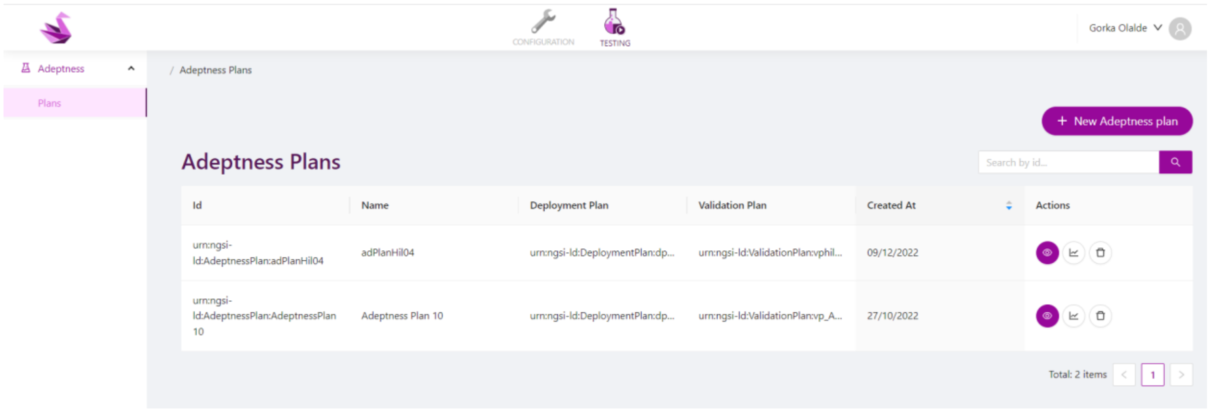
Figure 35 - TaaS screen to provision a new validation plan

Before creating a new validation plan, the necessary components must be provisioned in Stellio. TaaS allows you to do this with the menu that can be seen on the left side of the figure above

TaaS will be able with the validation plan information to automatically generate a JSON file compliance with the schema expected by the validation orchestrator and well defined in the validation orchestrator interface.

4.2.5 Creation of an Adeptness Plan and trigger of the execution workflow

The selection of a deployment plan and a validation plan is called an Adeptness plan. This Adeptness plan allows the automation of the execution of test cases. Figure 36 displays the screen in TaaS that allows the provisioning of an Adeptness plan.



Adeptness Plans

Id	Name	Deployment Plan	Validation Plan	Created At	Actions
urn:ngsi-ld:AdeptnessPlan:adPlanHil04	adPlanHil04	urn:ngsi-ld:DeploymentPlan:dp...	urn:ngsi-ld:ValidationPlan:vphil...	09/12/2022	
urn:ngsi-ld:AdeptnessPlan:AdeptnessPlan 10	Adeptness Plan 10	urn:ngsi-ld:DeploymentPlan:dp...	urn:ngsi-ld:ValidationPlan:vp_A...	27/10/2022	

Total: 2 items 1

Figure 36 - TaaS screen to provision an AdeptnessPlan

Once a plan has been created, a user can start the plan execution. TaaS will send a notification to Jenkins to start the execution. Next figure shows the trigger sequence diagram.

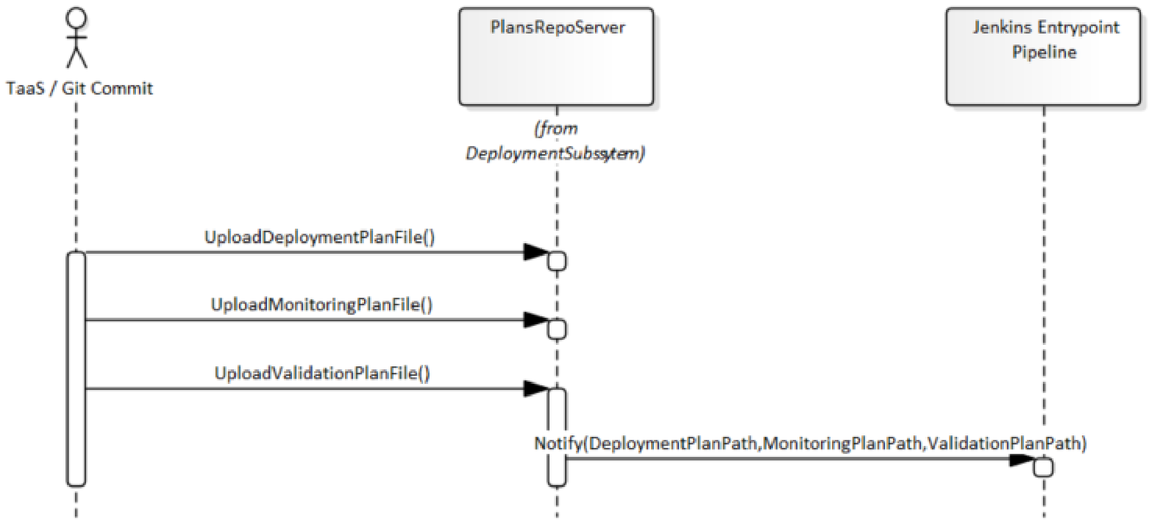
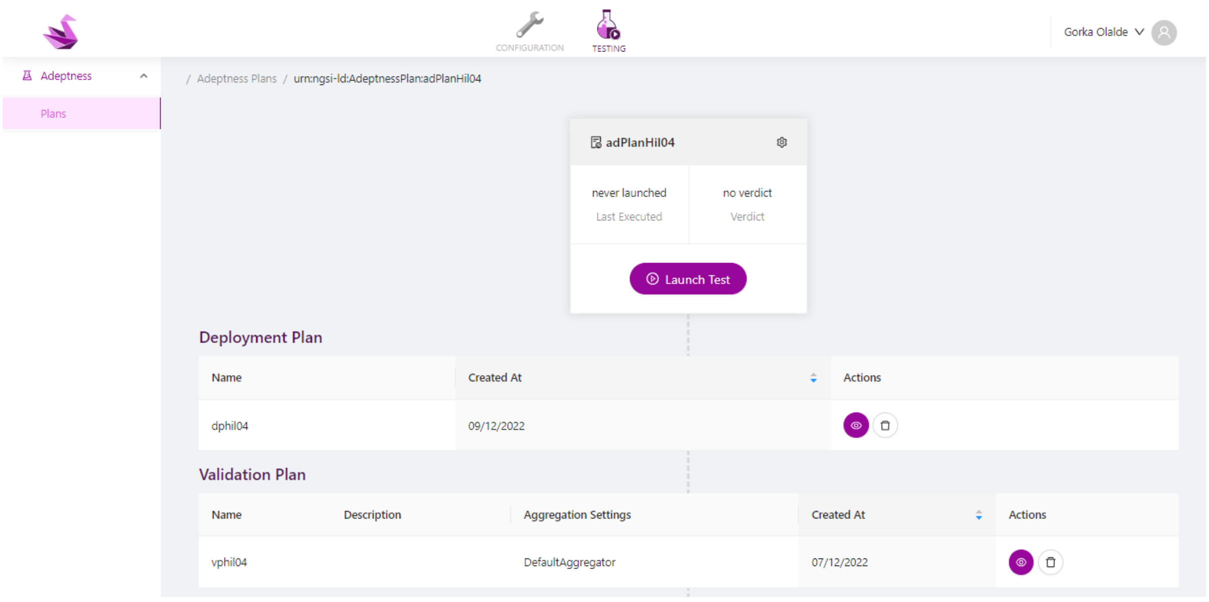

TaaS / Git Commit
PlansRepoServer
(from DeploymentSubsystem)
Jenkins Entrypoint Pipeline
UploadDeploymentPlanFile()
UploadMonitoringPlanFile()
UploadValidationPlanFile()
Notify(DeploymentPlanPath,MonitoringPlanPath,ValidationPlanPath)

Figure 37 - Workflow execution triggering sequence diagram

The request is directly sent from TaaS, which queries Stellio to explore the deployment, and validation plans components, and build three files with the JavaScript Object Notation (JSON)[22] format expected by the orchestrators. These files are uploaded to a repository and a signal with the files path is sent to the automation server. TaaS allows you to start the execution workflow. Next figure shows the screen to begin the validation process.

[22] https://www.json.org/



never launched	no verdict
Last Executed	Verdict

Deployment Plan

Name	Created At	Actions
dphil04	09/12/2022	

Validation Plan

Name	Description	Aggregation Settings	Created At	Actions
vphil04		DefaultAggregator	07/12/2022	

Figure 38 - TaaS screen to launch the adeptness plan execution

The result of the testcases in the plan can be also monitored with TaaS. In section 3.1.4.2, there is a detailed description of this feature.

4.3 Execution Workflow

This stage consists of the execution of an Adeptness Plan. The workflow is based on the sequential execution of several coordinated processes. Adeptness has used the features of an automation server, specifically Jenkins, to control these processes.

The following figure shows this flow in a very schematic way.

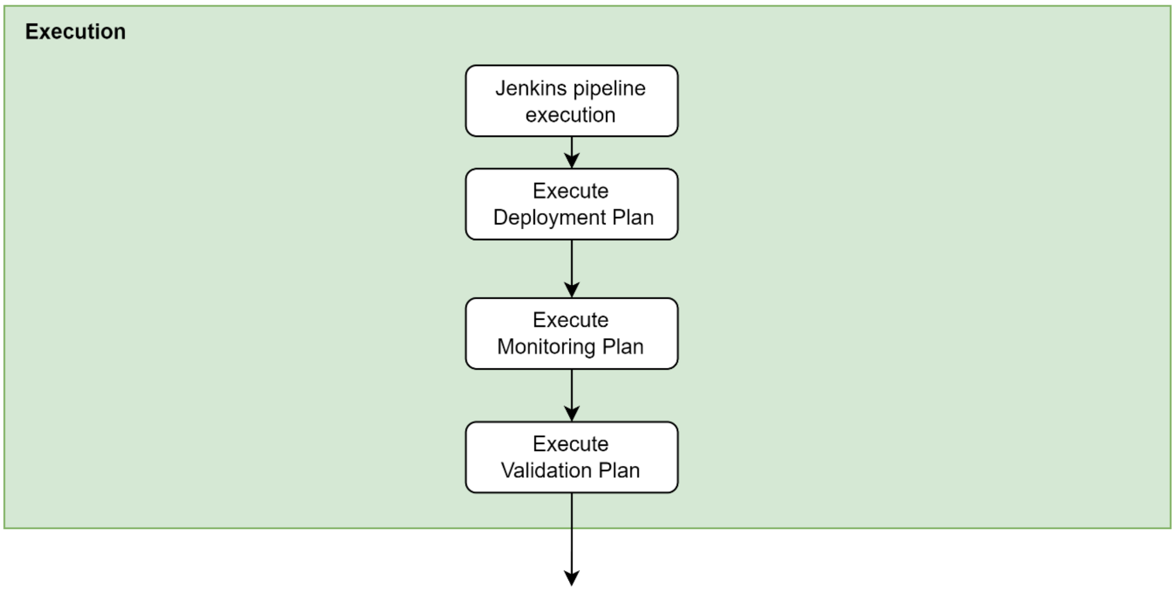
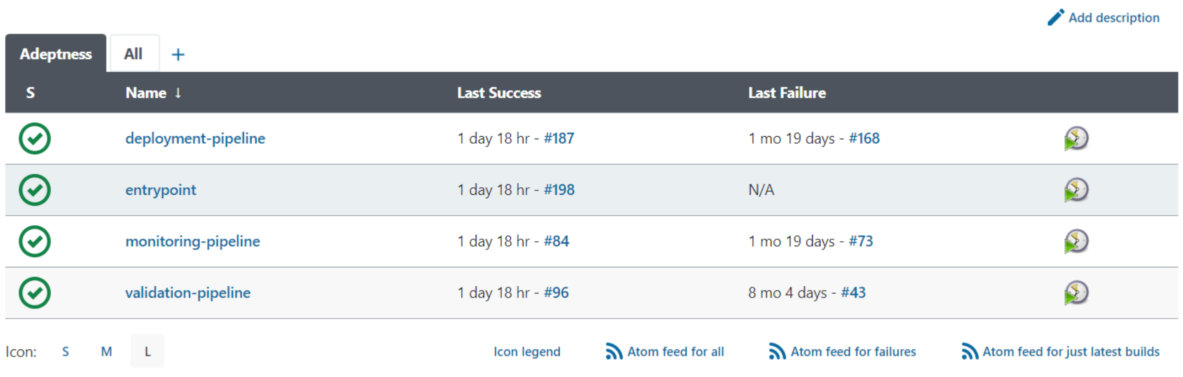


Figure 39 - Execution workflow

4.3.1 Jenkins Execution Pipelines

The workflow control is based on the development of 4 pipelines in Jenkins as shown in the following figure.


Add description

Adeptness | All | +

S	Name ↓	Last Success	Last Failure	
✓	deployment-pipeline	1 day 18 hr - #187	1 mo 19 days - #168	
✓	entrypoint	1 day 18 hr - #198	N/A	
✓	monitoring-pipeline	1 day 18 hr - #84	1 mo 19 days - #73	
✓	validation-pipeline	1 day 18 hr - #96	8 mo 4 days - #43	

Icon: S M L   Icon legend   Atom feed for all   Atom feed for failures   Atom feed for just latest builds

Figure 40 - Jenkins Screenshot with the Adeptness pipelines

These four pipelines are part of the basic Adeptness infrastructure. The start of the execution flow begins with the receipt of a notification at the entrypoint pipeline. TaaS triggers this notification with information about the location of these three files:

- Deployment plan: Information to be able to deploy components needed for validation.
- Monitoring plan: Information needed to configure the monitors required by the testcases.
- Validation plan: Information to configure the testcases and to be able to launch them. This includes also the launching of external tools.

The entrypoint pipeline will be responsible for launching the deployment-pipeline. Once the deployment-pipeline is launched, it will be in charge of controlling the triggering of the following pipelines. Figure 41 shows the execution sequence diagram of the pipelines.

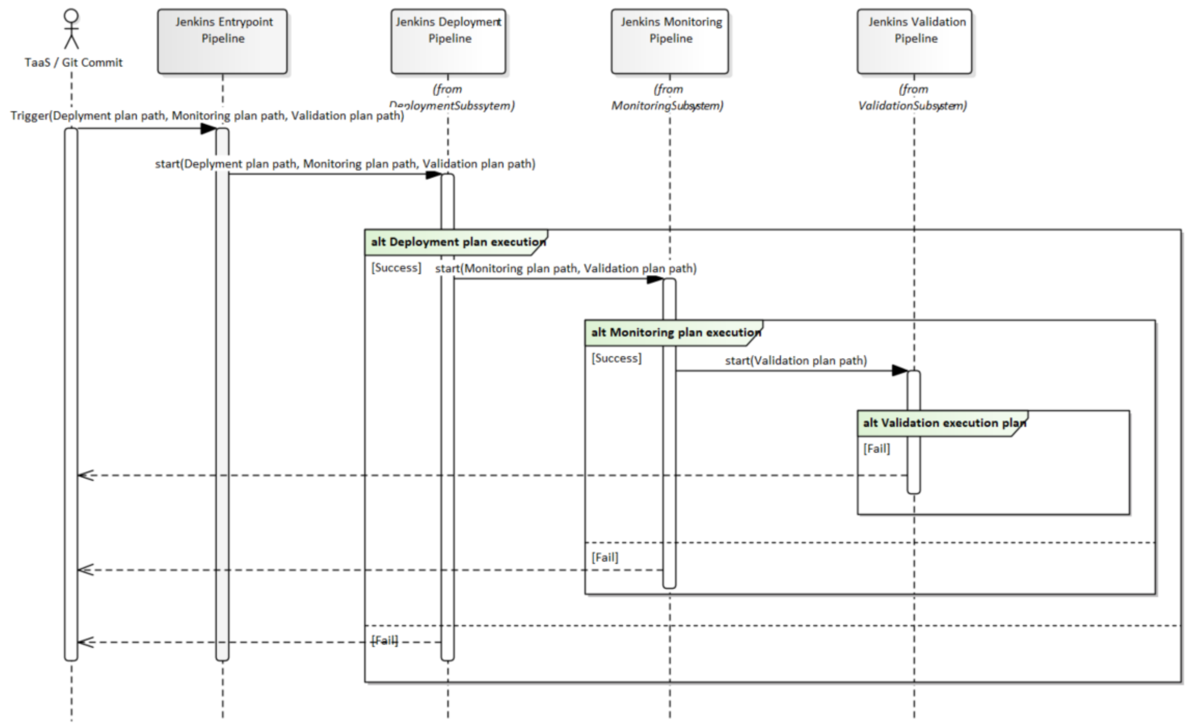

TaaS / Git Commit
Jenkins Entrypoint Pipeline
Jenkins Deployment Pipeline
(from DeploymentSubssytem)
Jenkins Monitoring Pipeline
(from MonitoringSubsystem)
Jenkins Validation Pipeline
(from ValidationSubsystem)
Trigger(Deplyment plan path, Monitoring plan path, Validation plan path)
start(Deplyment plan path, Monitoring plan path, Validation plan path)
alt Deployment plan execution
[Success]
start(Monitoring plan path, Validation plan path)
alt Monitoring plan execution
[Success]
start(Validation plan path)
alt Validation execution plan
[Fail]
[Fail]
[Fail]

Figure 41 - Pipelines execution sequence diagram

As can be seen, the execution of the pipelines is sequential and is conditioned to the result of the execution of the previous pipeline. The execution of the validation cannot be carried out if the necessary artefacts have not been correctly deployed beforehand and if the monitors have not been launched and configured.

The following sections detail the functions performed in each pipeline.

4.3.2 Deployment Plan execution

The deployment-pipeline is in charge of the deployment plan execution. The deployment plan contains the procedure to deploy all agents established in the Adeptness plan. This includes SUT, monitors, agents, oracles, external tools and test inputs. In a first stage, the deployment-pipeline will download the deployment plan file from the MinIO repository, previously uploaded by TaaS. This file will be compliance with the format expected by the deployment orchestrator. The following code extract shows this schema.

```
{
    "deployment-plan-id": "Deployment plan ID",
    "nodes": [
        {
            "node-id": "Node IP",
            "deployable-comps": [
                {
                    "id": "Service unique ID",
                    "package-type": "container",
                    "image-uri": "Source to obtain the container from. Uri
of the container registry",
                    "container-action-type": "action to execute on the
container: create,delete,update",
                    "net-mode": "optional (host), only when the container
should be launch as out of the internal docker network",
                    "enviroment-settings":{
                        "rest-port": "Microservice REST port ",
                        "mqtt-broker-ip": "Mqtt broker IP ",
                        "mqtt-broker-port": "Mqtt broker port ",
                        "mqtt-broker-qos": "qos"
                    }
                },
                {
                    "id": "Service unique ID",
                    "package-type": "genericfile",
                    "image-uri": "Source to obtain the GenericFile from.
Uri shall respond to one of these 2 schema: 1=
minio:MinioServerIP:MinioServerPort:bucket:*[.zip:zip | .tar:tar]   2=
http:WebServerIP:WebServerPort:*[.zip:zip | .tar:tar]",
                    "destination-path": "Path on the target node filesystem
to deploy in ",
                    "file-action-type": "action to execute on the
downloaded file (uncompress,  uncompress_execute, delete)",
                    "file-to-execute": "Needed only if the file-action-type
is uncompress_execute"
                }
            ]
        }
    ]
}
```

Next, the pipeline will use the REST API provided by the orchestrator to upload the deployment plan. An HTTP PUT request will be made on the resource */adms/v2/deployment-orchestrator/config/plan* and the deployment plan will be included in the body.

Once the plan is already loaded, the pipeline will use the REST API to order the execution of the deployment plan. An HTTP POST request will be made on the resource */adms/v2/deployment-orchestrator/config/execute-plan*. Figure 42 shows the overall view of the pipeline sequence diagram.

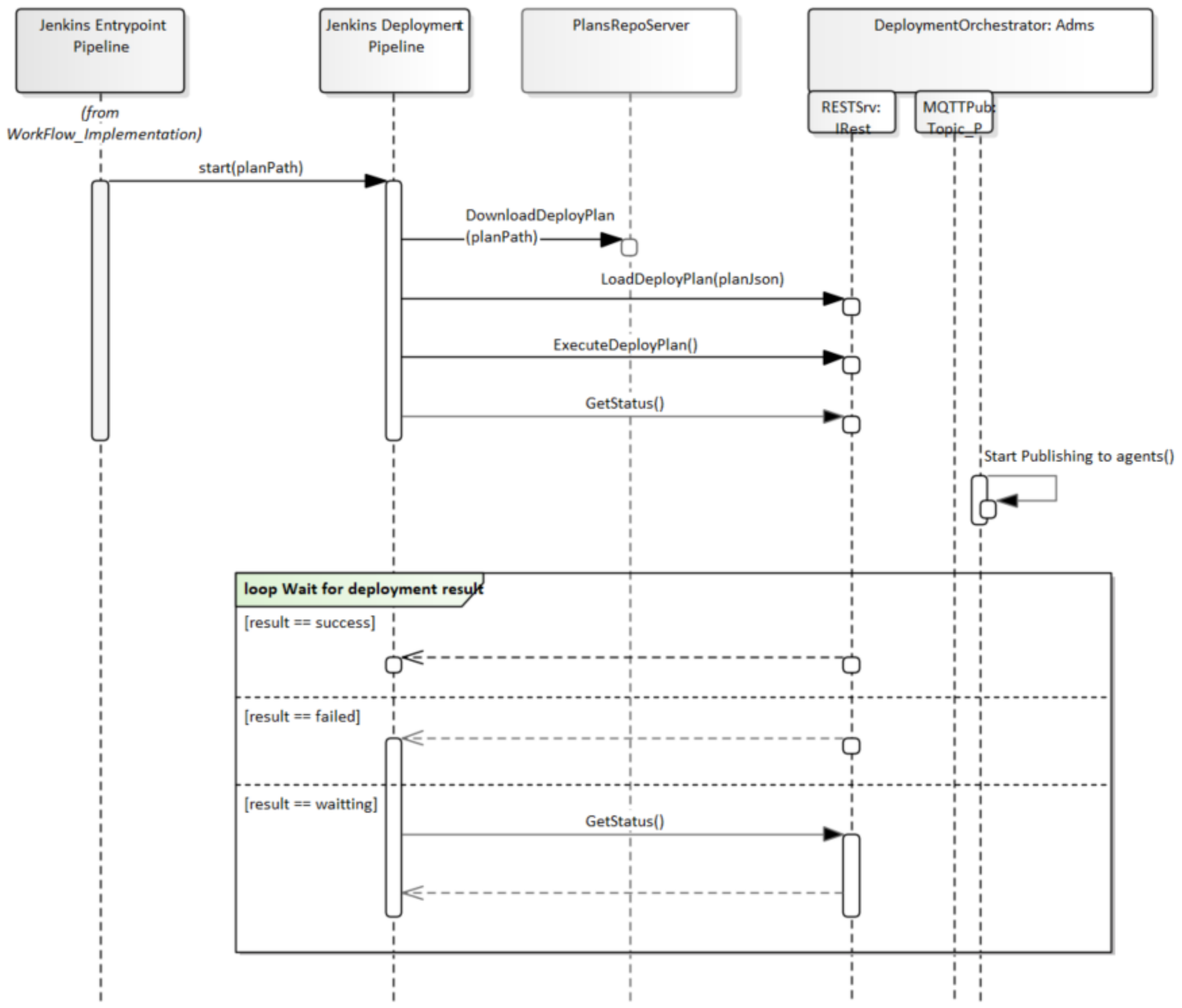

Jenkins Entrypoint Pipeline
(from WorkFlow_Implementation)
Jenkins Deployment Pipeline
PlansRepoServer
DeploymentOrchestrator: Adms
RESTSrv: IRest
MQTTPub: Topic_P
start(planPath)
DownloadDeployPlan (planPath)
LoadDeployPlan(planJson)
ExecuteDeployPlan()
GetStatus()
Start Publishing to agents()
loop Wait for deployment result
[result == success]
[result == failed]
[result == waitting]
GetStatus()

Figure 42 - Deployment pipeline sequence diagram

The orchestrator will extract from the plan the nodes it has to communicate with and prepare the MQTT messages that it will send to each node's deployment agent to download the component and launch it if necessary. The orchestrator will connect to the MQTT broker and for each component to be deployed will publish an MQTT message. Messages for components that are deployed on the same node will be published in the same topic (*adms/v2/deployment-orchestrator/{nodeId}/deploy-package*) but will have different payloads. The following figure shows the UML sequence diagram of the operations performed in the deployment process.

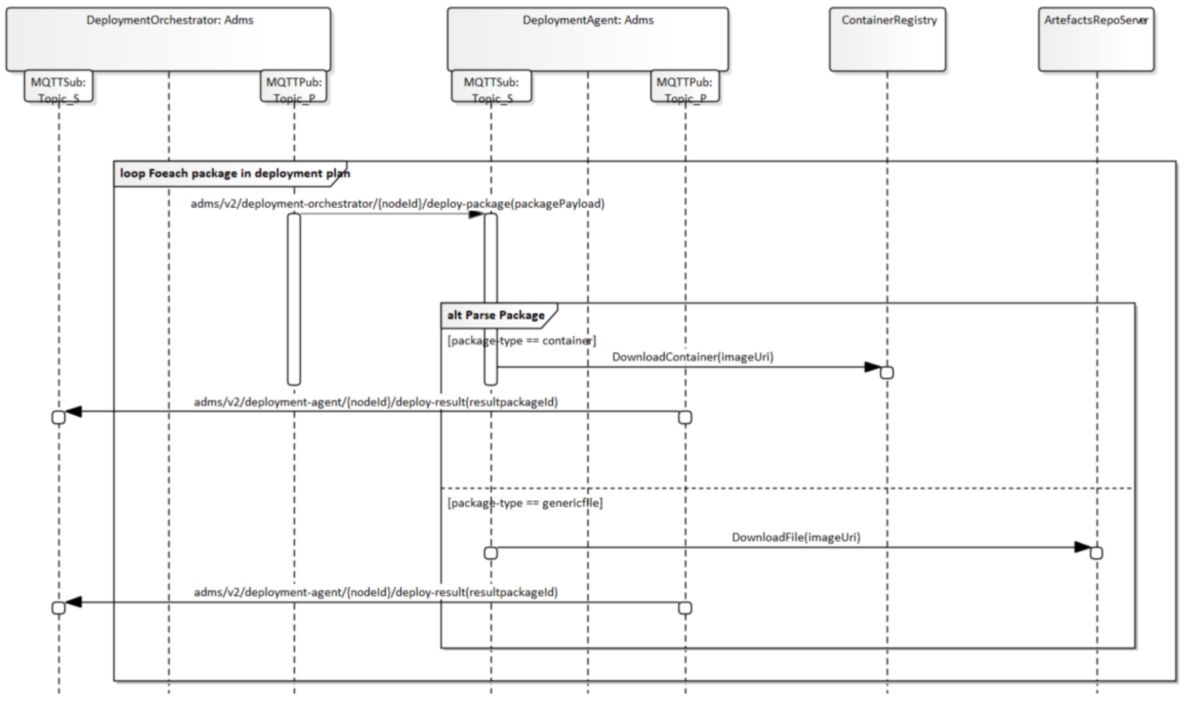

DeploymentOrchestrator: Adms
MQTTSub: Topic_S
MQTTPub: Topic_P
DeploymentAgent: Adms
MQTTSub: Topic_S
MQTTPub: Topic_P
ContainerRegistry
ArtefactsRepoServer
loop Foeach package in deployment plan
adms/v2/deployment-orchestrator/{nodeId}/deploy-package(packagePayload)
alt Parse Package
[package-type == container]
DownloadContainer(imageUri)
adms/v2/deployment-agent/{nodeId}/deploy-result(resultpackageId)
[package-type == genericfile]
DownloadFile(imageUri)
adms/v2/deployment-agent/{nodeId}/deploy-result(resultpackageId)

Figure 43 - Deployment process sequence diagram

In turn, the orchestrator will subscribe to the result topic (*adms/v2/deployment-agentr/{nodeId}/deploy-package-result*) sent by each node with the result of the component deployment. With all the information received from the subscriptions, it will store in memory the global result of the deployment and will publish this result in the MQTT broker so that a third party can make use of this information.

After the execution order has been requested, the pipeline will begin to query the orchestrator about the result of the deployment execution. Once the orchestrator has the result of all deployed components, the pipeline orders the execution of the monitoring-pipeline or ends the execution of the workflow if the result of the deployment of any of the components has not been successful.

4.3.3 Monitoring Plan execution

The monitoring-pipeline is in charge of the monitoring plan execution. The monitoring plan contains the configuration of the monitors deployed in the Adeptness plan. In a first stage, the monitoring-pipeline will download the monitoring plan file from the MinIO repository, previously uploaded by TaaS. This file will be compliance with the format expected by the monitoring orchestrator. The following code extract shows this schema.

```
{
    "monitoring-plan-id": "Monitoring plan ID",
    "monitor-agents": [
        {
            "monitor-agent-id": "Identifier of the monitor to configue",
            "node-id": "Ip of the node where the monitor agent is
deployed",
            "deployable-comp-rest-port": "Rest port of the monitor agent to
configure",
            "connection-settings": [
                {
                    "key": "Name of one connection parameter to handle the
physical interface",
                    "value": "Value of the connection parameter"
                }
            ],
            "sensors": [
                {
                    "sensor-id": "Unique identifier for the sensor to
monitor",
                    "sensor-name": "Display name for the sensor to
monitor",
                    "sensor-type": "Type of the sensor measurement",
                    "sampling-rate": "Frequency used to poll the sensor",
                    "sensor-settings": [
                        {
                            "key": "Name of one parameter to extract sensor
value from physical interface",
                            "value": "Parameter Value"
                        }
                    ]
                }
            ],
            "sensor-groups": [
                {
                    "sensor-group-id": "Unique identifier for the
SensorGroup that the monitor shall publsih",
                    "publish-rate": "Publication frequency",
                    "sensors": [
                        {
                            "sensor-id": "Unique identifier of the sensor
included in this group"
                        }
                    ]
                }
            ]
        }
    ]
}
```

The pipeline will use the REST API provided by the orchestrator to load the monitoring plan. An HTTP PUT request shall be made on the *resource /adms/v2/monitor-orchestrator/config/plan* and the monitoring plan file will be included in the body.

The pipeline will use the REST API provided by the orchestrator to order the execution of the monitoring plan. An HTTP POST request will be made on the resource */adms/v2/monitor-orchestrator/config/execute-plan*. Figure 44 shows the overall view of the pipeline sequence diagram.

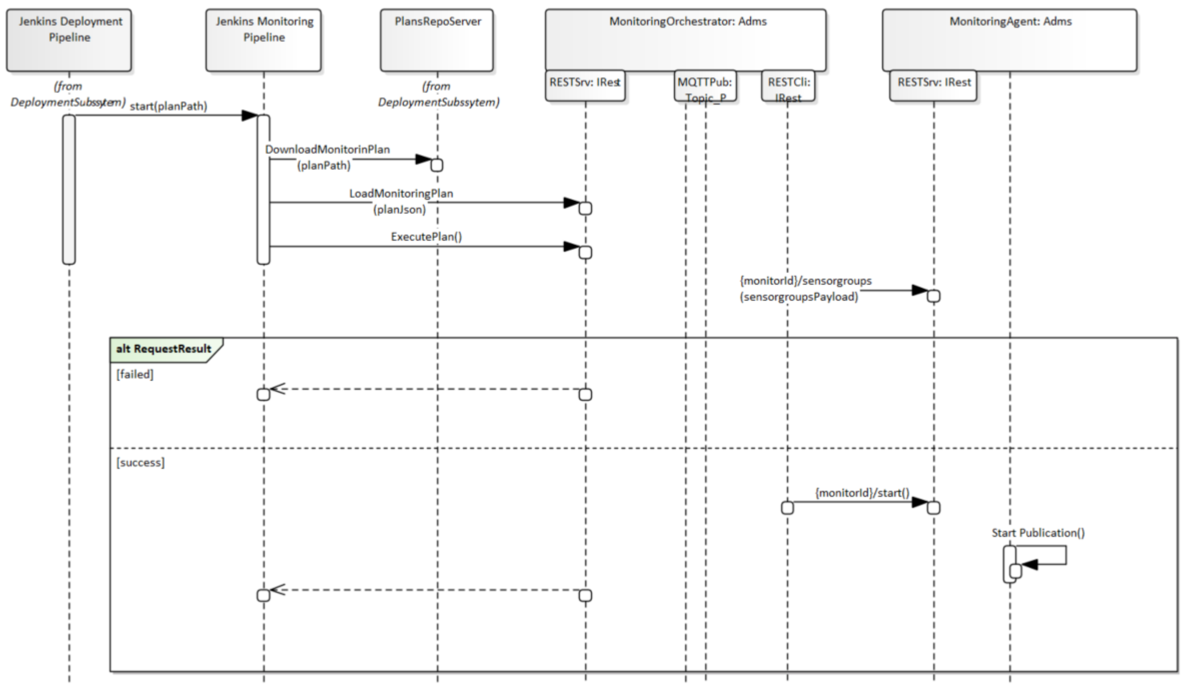

Jenkins Deployment Pipeline
(from DeploymentSubssytem)
Jenkins Monitoring Pipeline
PlansRepoServer
(from DeploymentSubssytem)
MonitoringOrchestrator: Adms
RESTSrv: IRest
MQTTPub: Topic_P
RESTCli: IRest
MonitoringAgent: Adms
RESTSrv: IRest
start(planPath)
DownloadMonitorinPlan (planPath)
LoadMonitoringPlan (planJson)
ExecutePlan()
{monitorId}/sensorgroups (sensorgroupsPayload)
alt RequestResult
[failed]
[success]
{monitorId}/start()
Start Publication()

Figure 44 - Monitoring pipeline sequence diagram

The monitoring orchestrator, once ordered to execute the plan it has loaded, parses it and executes one (or several) HTTP request to configure each of the monitors in the plan. Using the Rest API offered by each monitor, the orchestrator will configure the variables to monitor, as well as the subscription groups (groups of variables that are published together in the same MQTT message).

Finally, once the monitors in the plan have been configured, the orchestrator will send them the order to start reading and publishing the variables. The following sequence diagram shows the operations that the monitor performs once the orchestrator has ordered it to start reading the variables. The monitor starts a thread from which it parses the frames arriving from physical interface, updates its internal structures with the received values after parsing them according to the configuration and publishes the messages via MQTT using SenML format. The receiver of this information is the MQTT Broker, an MQTT_SenML Bridge is listening to the broker and sends to Stellio context broker the sensor values received. Next figure shows the sequence of this process.

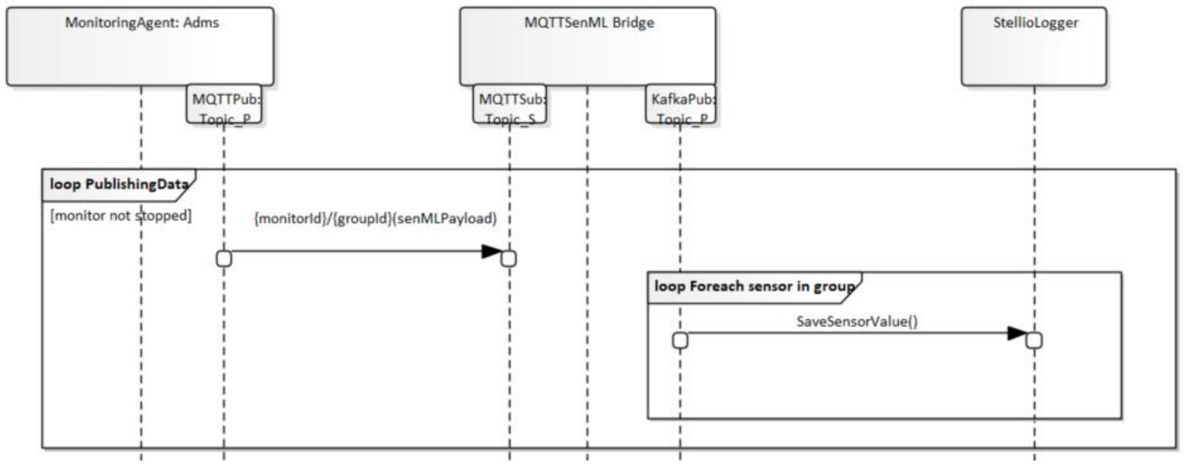


Figure 45 - Monitoring process sequence diagram

If all the requests from the orchestrator to configure and start the monitors finish successfully the pipeline orders the execution of the validation-pipeline otherwise ends the execution of the workflow.

4.3.4 Validation Plan execution

The validation-pipeline is in charge of the validation plan execution.The validation engineer will start defining the validation plan on TaaS, which will be stored on the Stellio backend. When the plan execution is requested by the user, the automation process will start. Stellio will generate a JSON formatted validation plan based on the entities stored in Stellio. This plan will be pushed to the automation server, that will start a pipeline to coordinate the execution on the validation plan.

On request by the pipeline, the validation orchestrator receives a validation plan containing all the required information for performing a validation procedure via a REST API call. This plan will be compliance with the format expected by the validation orchestrator. The following code extract shows this schema.

```
{
    "urn": "Validation plan unique identifier",
    "agents": {
      "Validation agent unique identifier": {
        "endpoints": {
          "rest_endpoint": "REST endpoint required to communicate with the
validation agent",
          "mqtt_endpoint": "MQTT endpoint required to communicate with the
validation agent"
        },
        "tests": {
          "Test case unique identifier": {
            "sut": {
              "name": "Name of the System Under Test",
              "type": "Type of the System Under Test",
              "configured_on": "Identifier of the node where System Under
Test is located"
            },
            "external_tools": {
              "External tool unique identifier": {
                "tool_type": "External tool type",
                "endpoints": {
                  "rest_endpoint": "REST endpoint required to communicate
with the external tool",
                  "mqtt_endpoint": "MQTT endpoint required to communicate
with the external tool"
                },
                "tool_settings": "Additional optional external tool
settings"
              }
            },
            "oracles": {
              "Oracle unique identifier": {
                "internal_name": "Name of the oracle",
                "endpoints": {
                  "rest_endpoint": "REST endpoint required to communicate
with the oracle",
                  "mqtt_endpoint": "MQTT endpoint required to communicate
with the oracle"
                },
                "inputs": {
                  "Name of an input of the oracle": {
                    "monitor": "Identifier of the monitor which the input
is binding to",
                    "sensor_group": "Identifier of the sensor group which
the input is binding to",
                    "sensor": "Identifier of the sensor which the input is
binding to"
                  }
                }
              }
            },
            "result": {
              "aggregation_type": "Aggregation procedure to be followed for
combining the received oracles verdicts and raising a global verdict for
the test case"
            }
          }
        },
        "result": {
          "aggregation_type": "Aggregation procedure to be followed for
combining the received test cases verdicts and raising a global verdict for
the validation agent"
        }
      }
    },
    "result": {
      "aggregation_type": "Aggregation procedure to be followed for
combining the received validation agent verdicts and raising a global
verdict for the validation plan"
    }
}
```

The validation orchestrator interprets the validation plan, requesting additional information of the components involved in the validation if necessary, and generates sub plans that contain the validation procedures to be delegated to each of the validation agents.

Upon an evaluation start request sent by the pipeline, the validation orchestrator sends the generated sub-plans to the corresponding validation agents through their REST API, delegating the coordination of the evaluation procedure specified in the sub plan to the validation agents. Figure 46 shows the overall view of the pipeline sequence diagram.

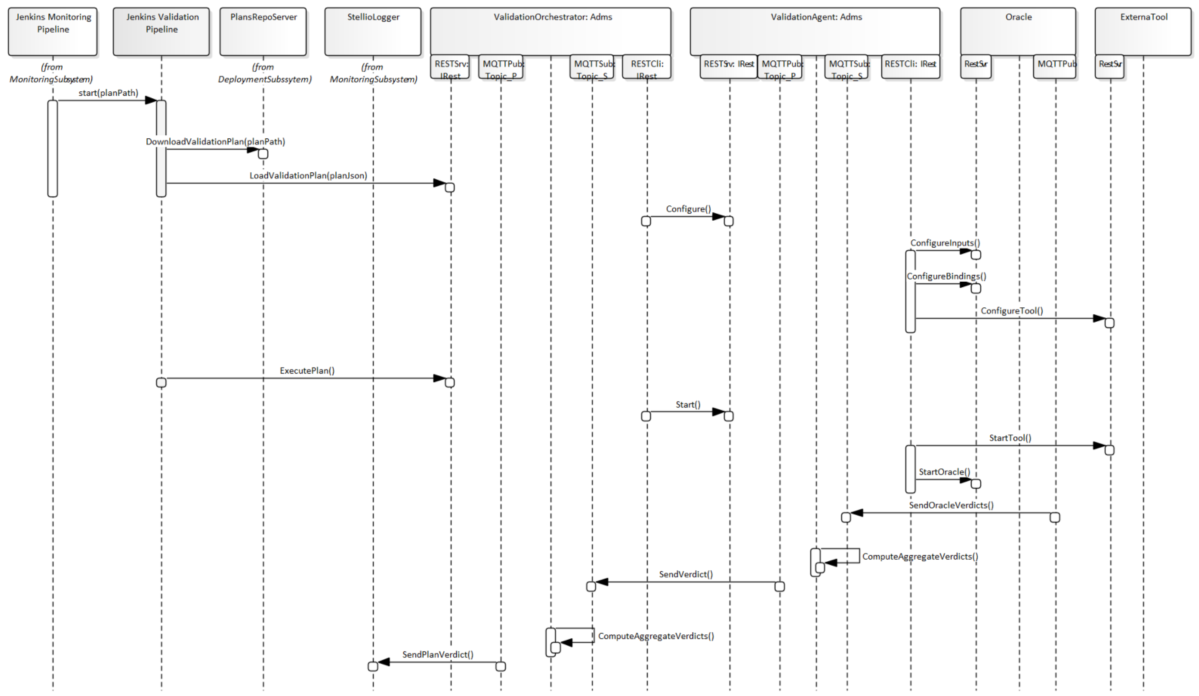


Figure 46 - Validation pipeline & validation process sequence diagram

When the orchestrator sends a JSON formatted validation plan to the '`adms/v1/validation-agent/plan`' REST endpoint through a POST call, the validation agent starts the interpretation and execution of the received plan. After the interpretation of the validation plan, the configuration of each of the child components will start.

Once all the child elements have been configured, the validation procedure will start. For this, both the oracle manager and the external tool manager will request status changes for oracles and external tools. The oracles will be the first components to be started, as the external tools might cause the publication of values to be evaluated. After all the oracles have been started, the external tool manager will be requested to send status change requests to the external tools.

The validation agent will subscribe to verdicts to be raised by child oracles. When an oracle verdict is received, the aggregator for each of the test structures will be called.

There is an aggregator file, implemented in the Adeptness python template, where all aggregators are implemented. Each aggregator defines a set of rules that must be complied in order to generate successful evaluation results. The default aggregator, for instance, checks if all verdicts of a specific group of components (oracles, external tools or test cases) are passed, otherwise, if one of them fails, it generates a 'failed' as a result. The aggregator has been developed in such a way that it can be easily extended, creating new aggregators with different rules, in order to meet the needs of specific group of components.

If the top-level aggregator reaches a verdict for all the test cases, the global verdict will be stored in the validation plan internal structure, and then forwarded to the validation orchestrator, which will run the corresponding aggregation for the parent validation plan.

Upon reception of a verdict, the validation orchestrator checks the aggregation rules defined on the validation plan, and computes a global verdict according to the aggregation rules. If a global verdict is obtained from the aggregation, the global verdict is published through MQTT and additionally, the verdict is also made available on the '**`adms/v1/validation-orchestrator/plan/verdict`**' REST API endpoint. The pipeline will collect the verdict for the validation plan and mark the pipeline as finished. Stellio will detect the verdict change and notify TaaS and therefore, the user about the resulting verdict for the validation plan.

4.4 Extensions

The extensions provide means to extract knowledge from the operational data at execution and/or operational time. Three main subsystems are involved under the extensions chapter:

- Uncertainty Subsystem: generates profiles related to unforeseen situations at design time, and detects uncertainty on the operational functionality of CPSoS at runtime.
- OSLC Bridge: provides traceability on discovered malfunctions of the SUT, by using both Change Management, Requirements Management and Quality Management OSLC domains.
- STL-Refined Prometheus Monitoring: receives the monitoring variable notifications from the edge nodes, and perform knowledge extraction over the behavioural functionality of the SUT using the validation strategy as the main topic.
- Recovery Subsystem: The recovery subsystem is responsible for managing the execution of recovery actions.

These subsystems are depicted in Figure 47.

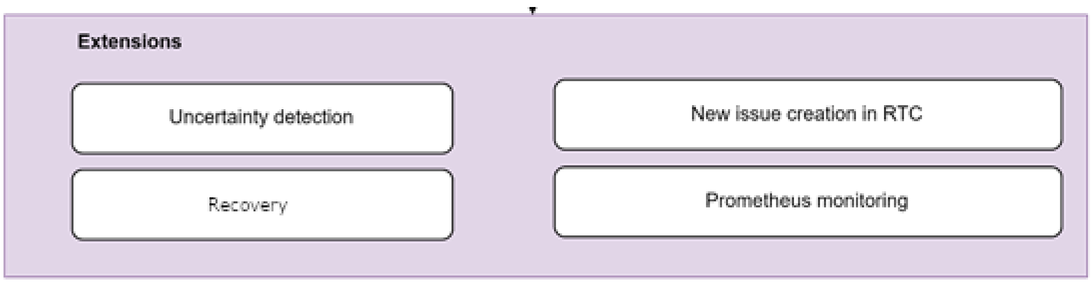


Figure 47 - Subsystems for knowledge extraction from operational data

These subsystems are explained in detail in the following subsections.

4.4.1 Uncertainty Subsystem: Generation + Detection

The uncertainty subsystem (shown in Figure 48) includes two parts: uncertainty generation and uncertainty detection. The uncertainty generation part aims to generate unforeseen profiles, which can be used as extra test inputs by the validation subsystem. As shown in the overview figure, four types of uncertainty generation methods have been developed, which applies various techniques, e.g., GA and RL. Different from uncertainty generation at design-time, the uncertainty detection part first employs Digital Twin technique and various machine learning techniques, e.g., Generative Adversarial Networks (GANs) and curriculum learning, to detect anomalies that are not known during the design-time. As a further step, the uncertainty-aware robustness assessment is performed to measure the robustness of the SUT in the presence of various uncertainties. If an anomaly is detected, corresponding actions should be taken, e.g., to handle the anomaly or inform the recovery subsystem. If no anomalies are detected, the robustness of the SUT against various uncertainties is then further assessed, and the robustness optimization strategy should be investigated once the poor robustness is detected. If a poor robustness is due to, e.g., software updates, then the corresponding recovery action should be taken.

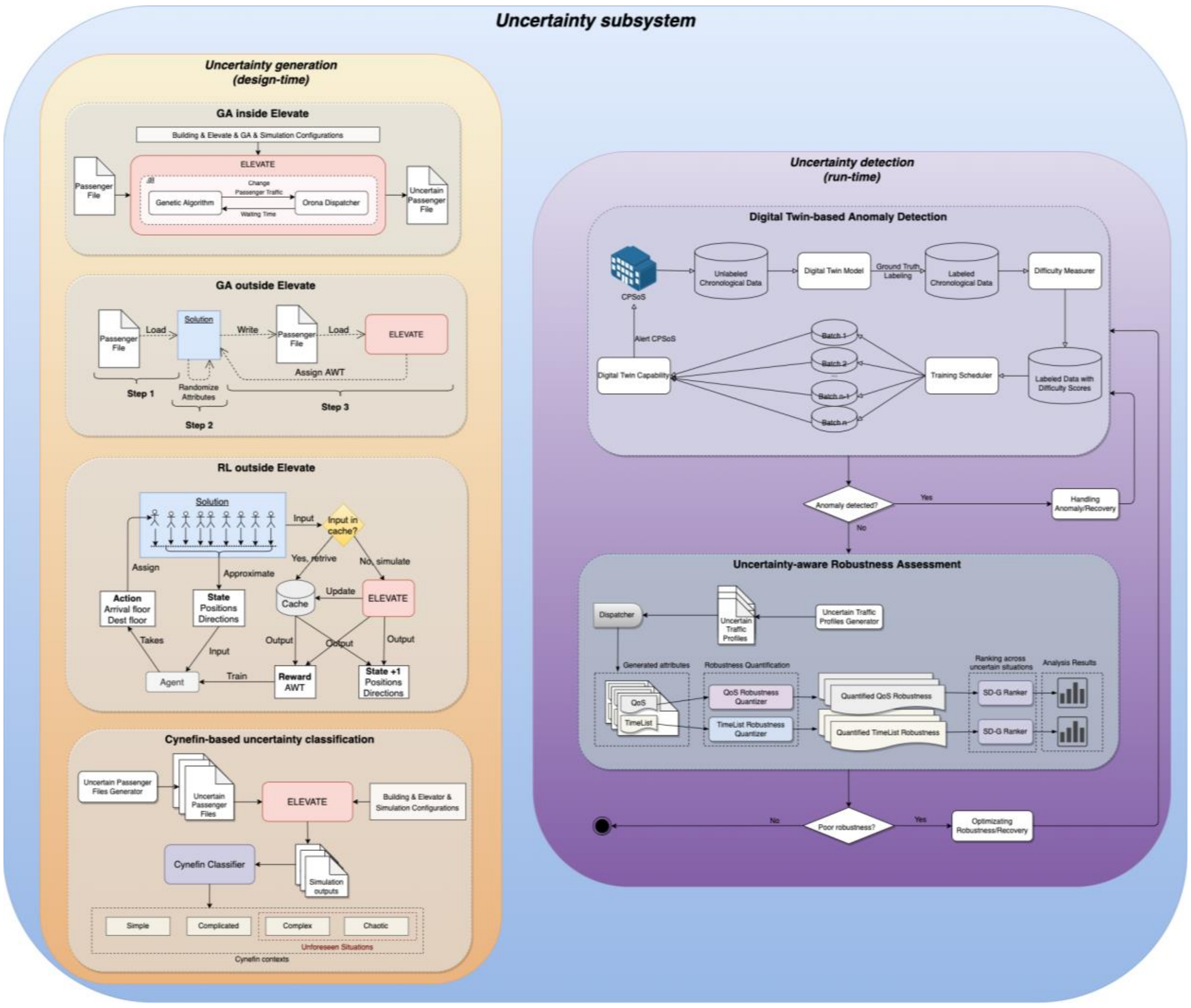


Figure 48 - Overview of uncertainty subsystem

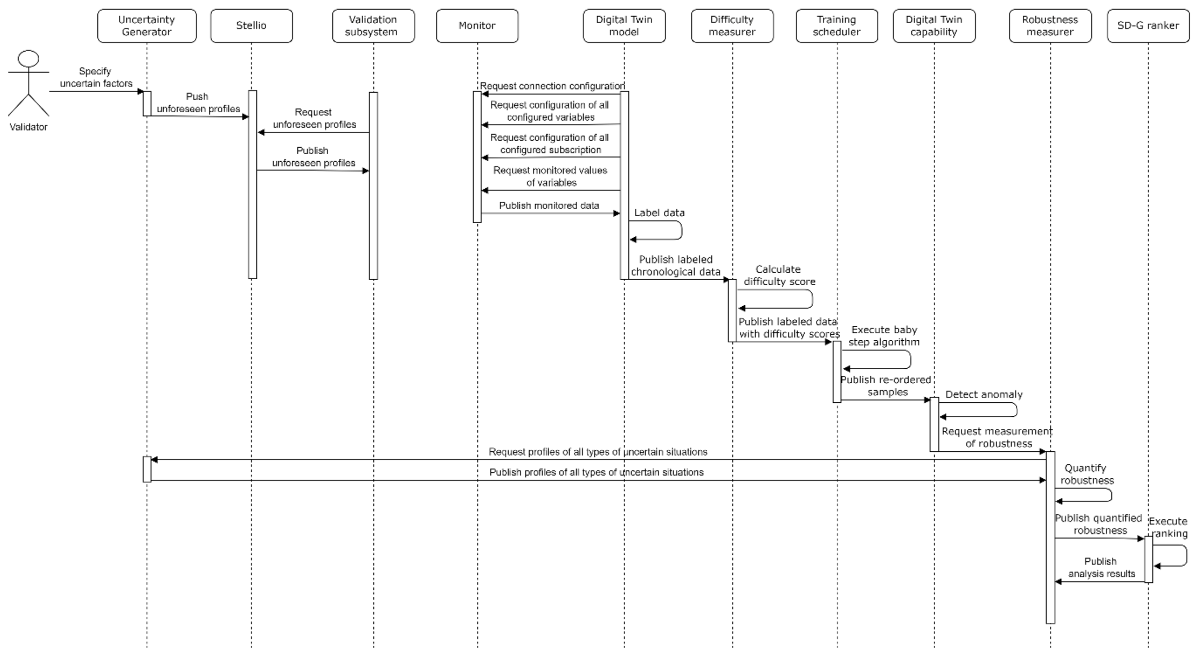

Uncertainty Generator
Stellio
Validation subsystem
Monitor
Digital Twin model
Difficulty measurer
Training scheduler
Digital Twin capability
Robustness measurer
SD-G ranker
Validator
Specify uncertain factors
Push unforeseen profiles
Request unforeseen profiles
Publish unforeseen profiles
Request connection configuration
Request configuration of all configured variables
Request configuration of all configured subscription
Request monitored values of variables
Publish monitored data
Label data
Publish labeled chronological data
Calculate difficulty score
Publish labeled data with difficulty scores
Execute baby step algorithm
Publish re-ordered samples
Detect anomaly
Request measurement of robustness
Request profiles of all types of uncertain situations
Publish profiles of all types of uncertain situations
Quantify robustness
Publish quantified robustness
Execute ranking
Publish analysis results

Figure 49 - Sequence diagram for the workflow of uncertainty subsystem

4.4.1.1 *Uncertainty generation at design-time*

The uncertainty generation aims to generate unforeseen situations at design-time. Four types of uncertainty generation methods have been developed using the elevator simulator-Elevate and employed various techniques.

- GA inside Elevate. To generate unforeseen situations due to several passengers, the GA was implemented, together with Orona's dispatcher, as a .dll inside Elevate. The GA selects passengers with highest waiting times as parents, then performs the crossover and mutation operations on passengers' attributes to generate offspring (e.g., x number of children). The generated offspring are used to update the attributes of the next x number of passengers, which are further simulated and assigned corresponding waiting times, based on which the GA performs parent selection again. The genetic loop continues for a certain number of generations so as to achieve good fitness, i.e., bad QoS. More details are presented in D4.4.
- GA outside Elevate. To generate unforeseen situations due to an entire passenger file, the GA was implemented outside Elevate. Different from GA inside Elevate, the initial population consists of several passenger files rather than passenger individuals. The GA selects passenger files with highest Average Waiting Times (AWTs) as parents, based on which the crossover and mutation operations are performed to generate offspring. Then all the generated offspring are simulated with Elevate and produce corresponding AWTs, based on which the GA performs the parent selection again. The GA continues until the desired fitness (i.e., bad QoS) is achieved. See D4.4 for more details.
- RL outside Elevate. To generate worst-case scenarios, the RL was implemented outside Elevate. The agent acts based on the perceived state, i.e., elevator's height and direction, of the environment. Then the agent can get a reward (i.e., AWT), based on the selected action, for training. More details are presented in D4.4.

- Cynefin based uncertainty classification. To classify various uncertain situations into Cynefin contexts, the Cynefin framework was customized for the elevator use case with statistical tests. The uncertain situations that are classified into Complex or Chaotic contexts are unforeseen situations, which means that the uncertain situations have significant impact on the QoS of the elevator system. Details are presented in D4.4.

As shown in the sequence diagram, the validator first specifies the uncertain factors (e.g., the loading and unloading time of each passenger), based on which the uncertainty generator, e.g., using GA, can generate unforeseen profiles. The generated unforeseen profiles are then pushed to the Stellio and can be further used by the validation agent to perform extra tests.

4.4.1.2 *Uncertainty detection at runtime*

The uncertainty detection focuses on 1) detecting anomalies that are not known during design-time and 2) poor robustness of the SUT in the presence of various uncertainties.

As shown in the overview figure and the sequence diagram, the digital twin model, i.e., a virtual replica of the targeted CPSoS, receives chronological data from the CPS through the monitor, and labels the chronological data based on the ground truth. Then the difficulty scores of the labelled data are calculated with the difficulty measurer. The training scheduler takes input the labelled data with difficulty scores and executes the baby step algorithm, and further re-orders samples, with which the digital twin capability can detect anomalies using machine learning algorithms. If an anomaly is detected, then corresponding action should be taken. If no anomaly is detected, then start the robustness assessment. The uncertainty generator first generates uncertain traffic profiles of all types of uncertain situations, each traffic profile generates corresponding QoS and Time List attributes, based on which the robustness measurer quantifies the QoS robustness and Time List robustness of the SUT under each uncertain situation. With the quantified robustness, the statistical difference-based grouping (SD-G) test is performed to rank the QoS robustness and Time List robustness, across all the uncertain situations, respectively. If a poor robustness, e.g., compared to the predefined maximum tolerable threshold of the low robustness, is detected, then the corresponding robustness optimization strategy or recovery mechanism should be taken.

More technical details of the anomaly detection and uncertainty-aware robustness assessment are presented in D3.2.

4.4.2 *OSLC Bridge*

The traceability microservice is closely connected to the availability of test execution results provided by the validation subsystem on the target SUT. During the execution of the test campaign, the oracles partially provide a forecasted result, concretely associated to one or more test cases defined in the validation plan. The semantic information is stored in Stellio logger in the form of NGSI-LD event data.

The OSLC Bridge is subscribed to such NGSI-LD events in Stellio. The notification of new data (in this case, validation execution events) is automatically published to all subscribers. The subscribers will then act according to the received data.

In the Traceability case, this data is specifically the failed test result obtained from an oracle associated to a test case:

- If the oracle provides a positive response (test case execution result is pass), then, no further actions are required.
- If the oracle provides a negative response (test case execution result is fail), then, the subsystem collects context information from the subscription (rich data). This NGSI-LD event data is then translated into OSLC CCM semantic data through the OSLC Bridge. Then, an associated Product Lifecycle Management server (e.g. IBM Rational Team Concert) registers a new defect with provided data received from Stellio: date, time, test case reference, endpoint, affected objects –SUT–, validation context, etc.

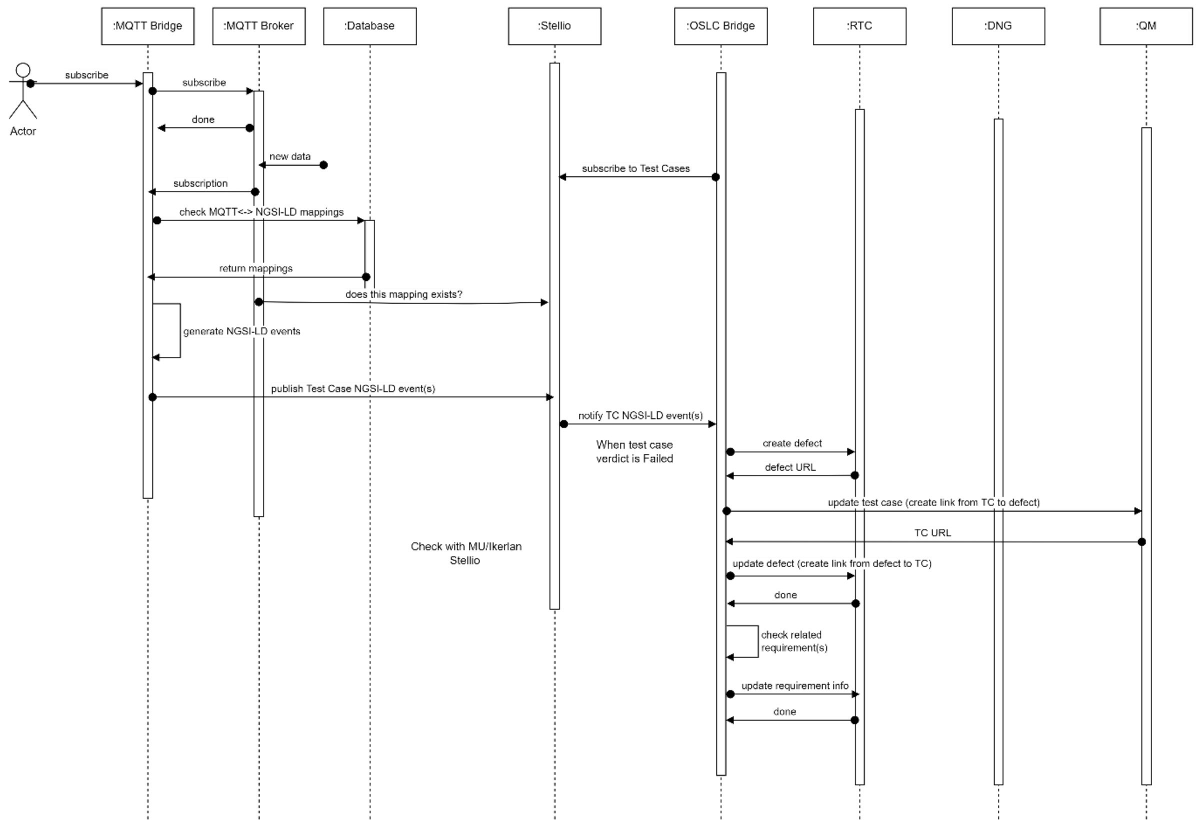

:MQTT Bridge
:MQTT Broker
:Database
:Stellio
:OSLC Bridge
:RTC
:DNG
:QM
Actor
subscribe
subscribe
done
new data
subscribe to Test Cases
subscription
check MQTT<-> NGSI-LD mappings
return mappings
does this mapping exists?
generate NGSI-LD events
publish Test Case NGSI-LD event(s)
notify TC NGSI-LD event(s)
When test case verdict is Failed
create defect
defect URL
update test case (create link from TC to defect)
TC URL
Check with MU/Ikerlan Stellio
update defect (create link from defect to TC)
done
check related requirement(s)
update requirement info
done

Figure 50 - OSLC Traceability from the Stellio subscription

Based on the data published in Stellio, feed by the validation strategy at runtime, the OSLC Bridge collects the changes in Test Cases, in order to perform reasoning about the existence of new defects in the SUT.

As far as Stellio contains context information, the identification of the SUT, the affected nodes and the details about the failing context (conditions, monitoring variables, failing point, etc.). All this information is collected and aggregated to an external product lifecycle management application, enabling the propagation of the defect, creating a change requests, that is associated to one or more requirement stored in a Requirements Management system, and even, notifying the existence of such change request to the Test Case using a Quality Management component. This facilitates the traceability on the validation strategy implemented in Adeptness.

Figure 50 provides details about how, from the subscription to Stellio, the verdict of a given Test Case triggers a chain reaction on the whole lifecycle traceability.

4.4.3 *STL-Refined Prometheus Monitoring plugin*

STL-Refined Prometheus Monitoring (SRPM) plugin is a monitoring system with formal language, Signal Temporal Logic (STL) refinement of alerting rule that enables it to connect to MQTT. This extension work focuses on how the formal language STL guides the generation of monitoring rule for Prometheus.

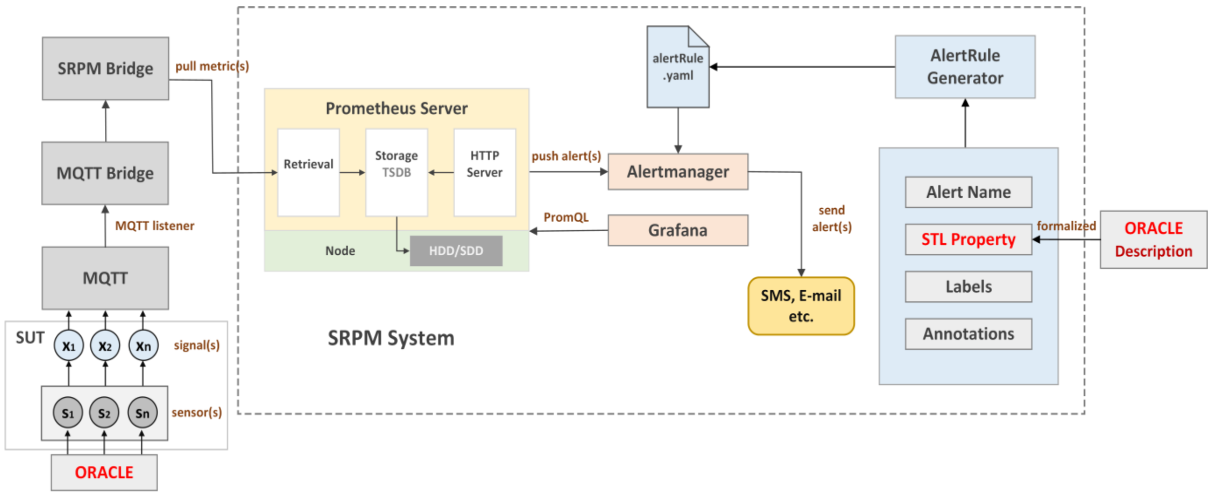


Figure 51 - Architecture of SRPM and SRPM Bridge to MQTT

As shown in Figure 51, SRPM system, inside the grey dashed box, integrates into the Adeptness core product through the connection to MQTT via MQTT and SRPM Bridge, which are implemented around two techniques. The most important one is how to formalize ORACLE into STL formula and how STL formula transformed into *PromQL*[24] (Prometheus query. Example: http_request_total >= 1000), and *for* clause, together with other clauses, for instance, *Alert Name*, *Labels* and *Annotations*, form the alerting rule file that can be automatically identified and configured by Alertmanager. After that, the Prometheus proceeds to perform monitoring duty and notifies the user if the monitored signal violates the alerting rule. Another one is the implementation of bridging the MQTT broker with Prometheus, in which signal(s) involved in the ORACLE will flow into the Prometheus by way of System Under Test (SUT), MQTT, MQTT Bridge and SRPM Bridge for monitoring. The following sections will unveil the technical details.

4.4.3.1 *SenML-MQTT Prometheus agent*

The implementation of bridging the MQTT and Prometheus illustrated in Figure 52Figure 51.

[24] https://prometheus.io/docs/prometheus/latest/querying/basics/

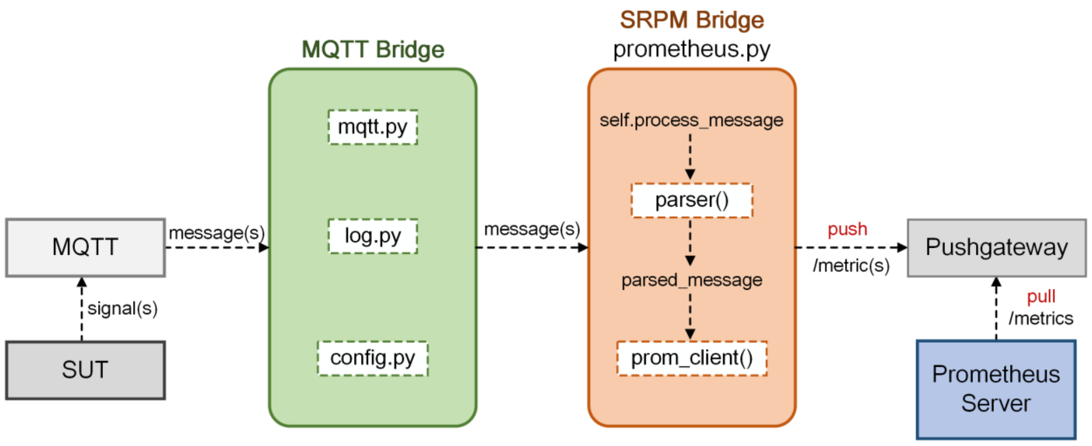


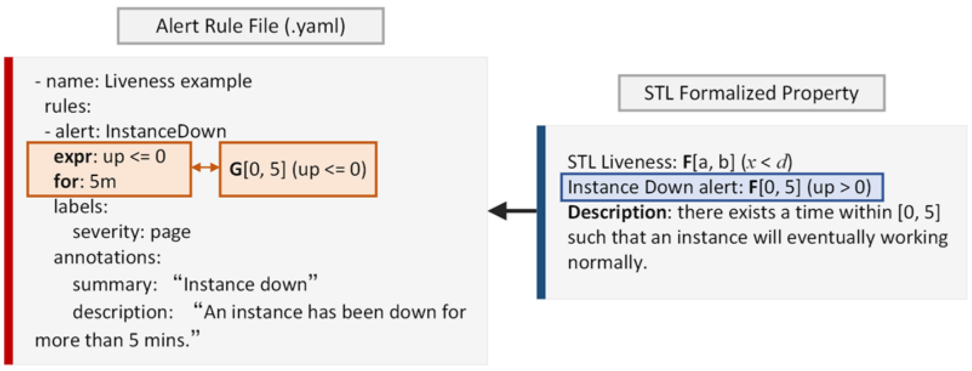
Figure 52 - Implementation of bridging the MQTT and Prometheus

MQTT Bridge is employed to connect to the cloud broker and listen to the configured topic in MQTT. The MQTT Bridge will output the message, which is accepted by SRPM Bridge as input.

4.4.3.2 *STL Refinement for Alerting Rule in Prometheus*

This section mainly elucidates how the STL is transformed into an important part of the alerting rule. STL is a formal, declarative specification formalism for capturing sophisticated temporal requirements, such as safety, reachability and liveness property, for CPS/IoT applications, which are typically interpreted over real-time and real-valued behaviours and admit quantitative semantics to allow measuring how far is a behaviour from satisfying or violating a specification. In this extension, STL will be utilized to guide the generation of alerting rule in Prometheus on the basis of the monitoring requirement of the CPS system.

Figure 53 exemplifies how STL formalized property of the requirement is transformed into the alert rule with equivalent monitoring objective.



```
- name: Liveness example
  rules:
  - alert: InstanceDown
    expr: up <= 0
    for: 5m
    labels:
      severity: page
    annotations:
      summary:  “Instance down”
      description:  “An instance has been down for
more than 5 mins.”
```

STL Liveness: **F**[a, b] $(x < d)$
Instance Down alert: **F**[0, 5] (up > 0)
Description: there exists a time within [0, 5] such that an instance will eventually working normally.

Figure 53 - Example of transformation for Alerting Rules in Prometheus

As shown in Figure 53, the alert rule (left column) and STL formalized property (right column) share the same monitoring objective, that is if an instance continues to be down for more than 5 minutes and the monitoring system will alert and send notification to the user. The logic of their monitoring rules is reversed. The workflow of SRPM is shown in Figure 54.

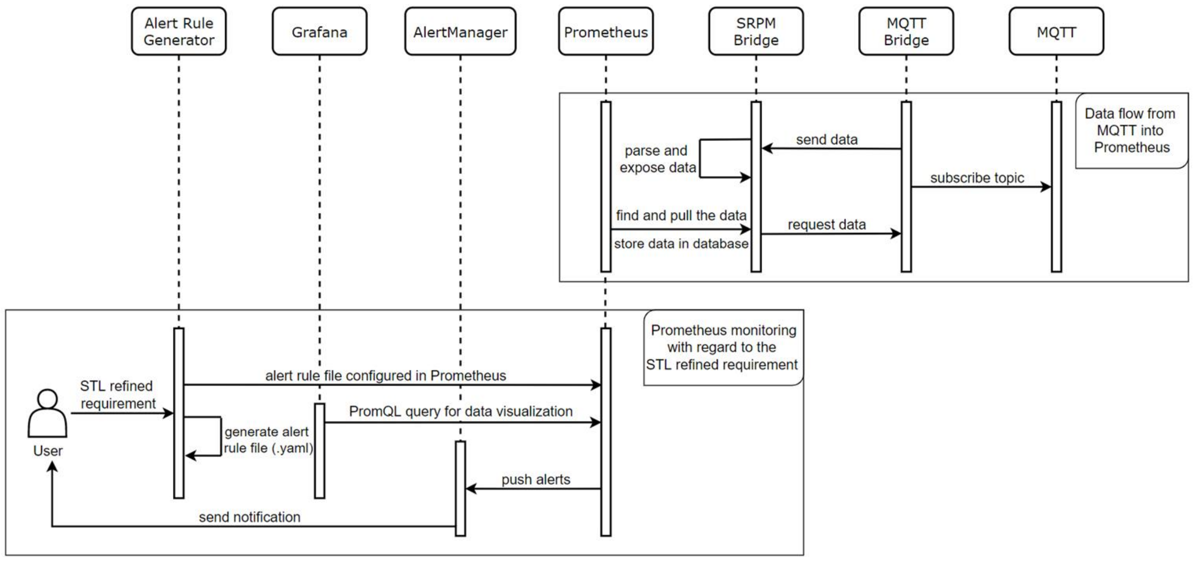


Figure 54 - Sequence diagram for the Prometheus workflow based on the configuration capabilities

5 CONCLUSIONS

This document represents the final approach to define the logical sequence to configure, provision and run microservices in the Adeptness platform. On the one side, in this document, we do not include the content already provided in the toolchain. On the other side, we assume that the reader is already familiar with the toolchain, including configuration and capabilities built upon the Adeptness infrastructure.

In the first part of the deliverable, we have provided a reference guide of involved subsystems in Adeptness ecosystem and the associated microservices. This allows to early adopters to map specific components in their infrastructure to check the available and missing component availability, and the added-value functionality, tooling and methods enabling design-operation continuum validation.

The second part of the deliverable focuses on the staged workflow to enable microservices-based architecture, the monitoring, validation, uncertainty detection and recovery of SUT and how the knowledge extraction of operational data potentially provide benefits on the deployment effort, a reduction in time to recovery and a reduction of bugs. The workflow represents an adopters' overview on the steps required to create from the scratch or to add microservices to the Adeptness platform. As far as the provisioning to Stellio context broker is a key step to declare the ecosystem context, TaaS front-end provides an easy way to define use case-specific elements for Adeptness. An adopted can use the provisioned artifacts to create their own plans, targeted to run a validation campaign over one or more cyber-physical systems, such as a distributed lift group, or a Train Control Management System (TCMS) targeted to improve the compliance and reaction on failure at validation. These use cases will be realised in the work of follow-up Work Package 6 and 7 in the Adeptness project.

Once the ecosystem is provisioned and the Adepteness plans specified, containing the deployment, monitoring and validation components, the design-continuum operation methods for validation campaign eases the understanding of the development process, and the visualization at runtime of the validation process, linked to the industrial partners' business goals.

Finally, the extensions defined for the current project enables the understanding of the current validation process, the detection of possible corner cases, and the enablement of backward and forward traceability over the full development lifecycle, both at software level and at system level.

6 RISK REGISTER

Risk Number	Description of Risk	Proposed Risk Mitigation Measure	Probability/ effect

The partners have not identified major risks during the preparation of the deliverable.

7 QUALITY ASSURANCE

The executive board is the body for quality assurance. The procedure for review and approval of deliverable is described in Deliverable Report D8.1 – "Project handbook". The quality will be ensured by checks and approvals by WP Leaders as part of the executive board (see front pages of all deliverables).

8 ACKNOWLEDGMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement no. 871319.

Disclaimer