Ref. Ares(2022)2431677 - 01/04/2022

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

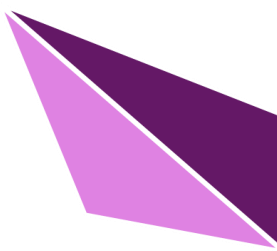
EUROPEAN COMMISSION

Horizon 2020

H2020-ICT-01-2019

GA No. 871319

Deliverable No.	ADEPTNESS D5.3	
Deliverable Title	Workflow and toolchain documentation	
Deliverable Date	2022-03-31	
Deliverable Type	Report	
Dissemination level	Public	
Written by	UES	2022-03-20
Checked by	SRL, TUW, MGEP, IKL	2022-03-31
Approved by	Executive board	2022-03-31
Status	v1.1	2022-03-20



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
Francisco Ruiz Sergio Rodríguez	UES
Aitor Arrieta Goiuria Sagardui Gorka Olalde	MGEP
Franck Le Gall Benoit Orihuela Romain Magnani	EGM
Aitor Agirre Blanca Kremer	IKL
Shaukat Ali Liping Han	SRL
Zlatan Tucaković Meixun Qu	TUW
Wasif Afzal Amirali Piadehbasmenj	MDH

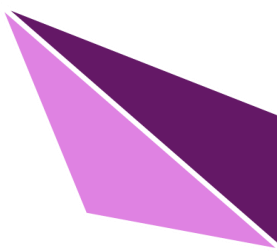
Document Change Log

Name	Date	Comments
V1.0	2022-02-15	Initial draft
V1.1	2022-03-31	Added contribution from partners

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 specify the variety of collaborative components.

Section 4 focuses on the workflow sequence to deploy and run design-operation continuum engineering for CPSoS for each of the four stages defined above.

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 M24 (2021-12-31) to M27 (2022-03-31).

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 three main domain subsystems defined in the core platform:

- Deployment subsystem
- Validation subsystem
- Monitoring 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 SUBSYSTEMS PARTICIPATING IN CPSOS

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 and logger subsystems.
- The Validation subsystem, including orchestrator, agents and oracles.
- The Test-as-a-Service (TaaS) subsystem, as specified in D5.1. (Adaptation of TaaS for continuous CPSoS Testing).
- The Test Case Generation subsystem.
- The Traceability subsystem, enabling Lifecycle Management.
- The Uncertainty subsystem, based on the results obtained from the Monitoring subsystems.
- 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.

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 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 deployment 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 agents[2].

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

3.1.1 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 platforms in order to not only instantiate the software components that contain the CPS functionality (SUT) but also those other software artefacts that are needed for the validation of the CPSoS in operation.

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 to execute the deployment pipeline.

The orchestrator needs a deployment agent on each of the devices 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.

3.1.2 Deployment agent microservice

The deployment agent is the component that enables the remote deployment on each Edge node. It receives and processes commands from the deployment orchestrator and, based on the content of the message received, configures and launches new services using the corresponding method. Currently, the deployment agent is able to deploy two types of applications: containerized applications based on Docker images and files or executables that are downloaded and executed without containerization.

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

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

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.

3.2 Monitoring and Logger Subsystem

The monitoring subsystem is explained in deliverables D1.1 and D2.2. Section 4.5.1 of deliverable D1.1 contains the first design of the monitoring subsystem. Section 4.5.2 of deliverable D1.1 contains the first design of the logger 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 section 3.3, 3.4, 4.2 and 4.3 of deliverable D2.4 describe the logger subsystem in-depth. This section contains a brief description of the two subsystems.

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 logger subsystem 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 other Adeptness services (such as the Test Generator and OSCL Bridge) and the data collected from monitors and oracles. During the project course, the logger subsystem 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 logger subsystem in Adeptness. This has not prevented us from also exploring and implementing options to connect Prometheus to the Adeptness workflow in particular scenarios.

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 solutions and extended tools used in logger subsystem are:

- Stellio context broker.
- SenML-MQTT Stellio Bridge.

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

- Prometheus Monitoring.
- SenML-MQTT Prometheus agent.
- Signal Temporal Logic (STL) runtime monitor.

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.

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

3.2.2 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 the sensors in SenML format using MQTT.

3.2.3 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.

3.2.4 Stellio

Stellio[4], created and maintained by EGM, is a FIWARE Generic enabler, freely accessible on GitHub[5], and released under the Apache Public License.

[4] https://stellio.io

[5] https://github.com/stellio-hub/stellio-context-broker

Beyond the implementation of the NGSI-LD standard[6], Stellio context broker is fully compliant with a "FIWARE architecture"[7]. 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.

3.2.5 SenML-MQTT Stellio Bridge

The SenML Bridge is intended to get data from deployed sensors, verdicts or else oracles, 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.

3.2.6 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.

3.2.7 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.

3.2.8 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

[6] https://www.etsi.org/deliver/etsi_gs/CIM/001_099/009/01.04.01_60/gs_cim009v010401p.pdf

[7] https://www.fiware.org/developers/catalogue/

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.

3.3 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 in validation subsystem are:

- Validation orchestrator microservice
- Validation agent microservice
- Oracle microservice
- External tool microservice
- DSL2 Oracle tool

3.3.1 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 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.

3.3.2 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.

3.3.3 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 it is REST and MQTT APIs. The validation agents will coordinate with the oracle microservices for configuring them and to expect verdicts raised based on the performed evaluations.

3.3.4 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 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.

3.3.5 DSL2Oracle tool microservice

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 xTEXT8, 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.

3.4 TaaS Subsystem

The TaaS (Test-as-a-Service) subsystem 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 tools to ease the configuration part, also called the provisioning part, where the Stellio context broker registers the needed components. The TaaS then uses those components to generate plans, such as Monitoring, Deployment and Validation, which are later used by the Adeptness Plan to start the test.

Once the test campaign is launched, the TaaS front-end is notified for each new verdict, 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.

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

The only defined microservices in TaaS subsystem is the NGSI-LD socket notifier.

3.4.1 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. A socket bridge API called *NGSI-LD socket notifier*, receives these notifications through Stellio. The notifier dispatches them to the front-end using Web Sockets, according to the user/tester and the right TaaS instance. 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.

3.5 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.

3.5.1 GraphWalker as a Restful service

The following diagram in Figure 1 shows all REST API Commands used to control GraphWalker (GW) and interact with the model. Below we also describe the commands in detail.

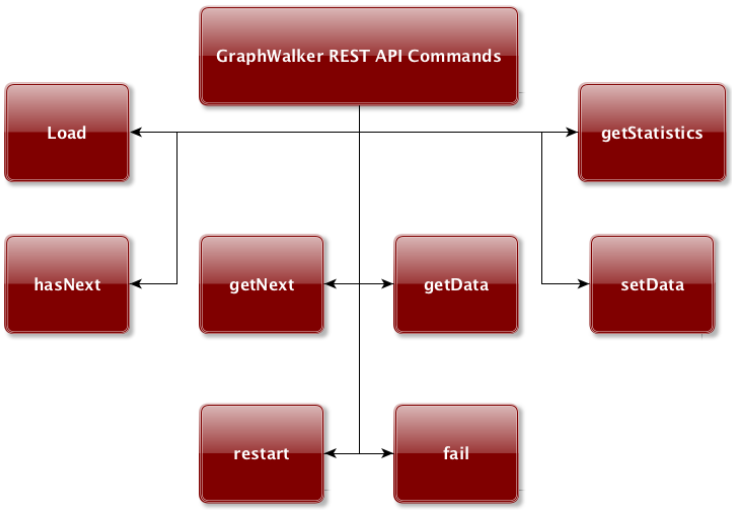


Figure 1. GraphWalker REST API commands

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**".

3.6 OSLC Bridge Subsystem

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).

3.6.1 OSLC Bridge Microservice

The OSLC Bridge brings a common standardized interface connected to product lifecycle management tools. The full specification is available at Open Services Specifications Website[9].

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

- Core Specification, which defines the overall approach to OSLC-based specifications and capabilities. These capabilities are often needed across several domains, and provides 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.

3.7 Uncertainty Subsystem

The uncertainty subsystem includes two parts:

- uncertainty generation at design-time.

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

- uncertainty detection at run-time.

The first part, which employs various techniques such as Genetic Algorithm (GA) and Reinforcement Learning (RL), aims to generate unforeseen situations automatically at design-time so as to collaborate with the validation subsystem.

The second part uses the time-series data from a CPS to detect anomalies that were not known at the design-time. The anomaly detection is further connected to the uncertainty-aware robustness assessment, which aims to assess the robustness of a CPS against various uncertainties occurring in the run-time, e.g., unpredicted loading and unloading time of a passenger. The defined microservice in the uncertainty subsystem is the uncertainty detection, which is explained in the following subsection.

3.7.1 Uncertainty detection

The uncertainty detection is responsible for detecting anomalies and poor robustness of the SUT. The anomaly detection employs a digital twin and various machine learning techniques, e.g., Generative Adversarial Network (GAN), to detect anomalies based on the operational data provided by the monitors. The uncertainty-aware robustness assessment employs various statistical tests to comprehensively quantify the robustness of the SUT under different uncertain situations caused by several uncertain factors, which can be specified with uncertainty datatypes in the uncertainty libraries, such as uncertain mass of the passengers.

3.7.2 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 validation agent and uncertainty agent 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 their corresponding microservices in order to be executed.

4 DESIGN-OPERATION CONTINUUM ENGINEERING 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 provides the microservices to Stellio and generates the Adeptness plan, 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.

Here, for all subsystems defined in Adeptness, the workflow at configuration and runtime is established, and dependencies along these subsystems is highlighted.

The following subsections provide a detailed sequence of the phases described in Figure 2.

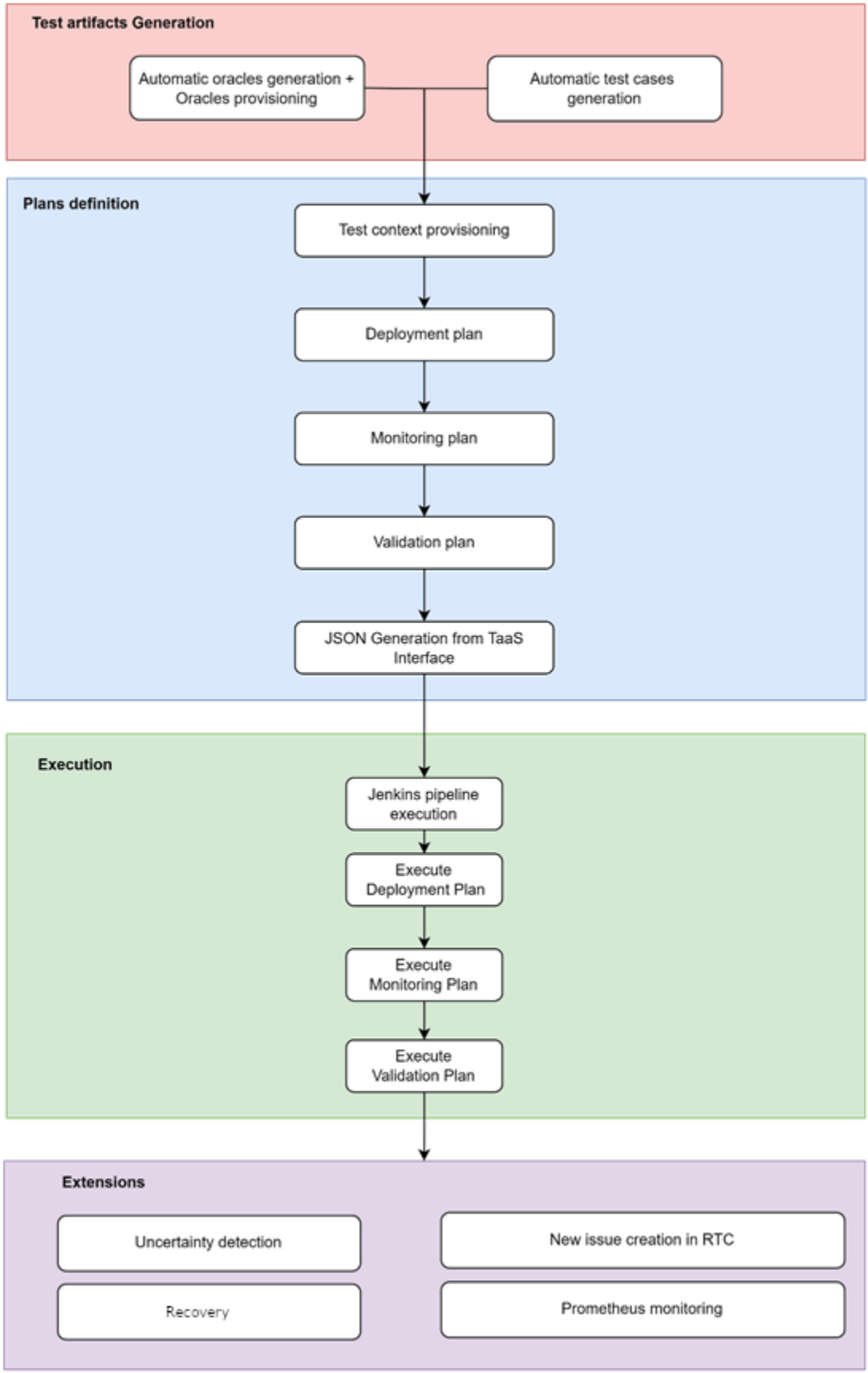


Figure 2. Workflow definition

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 3):

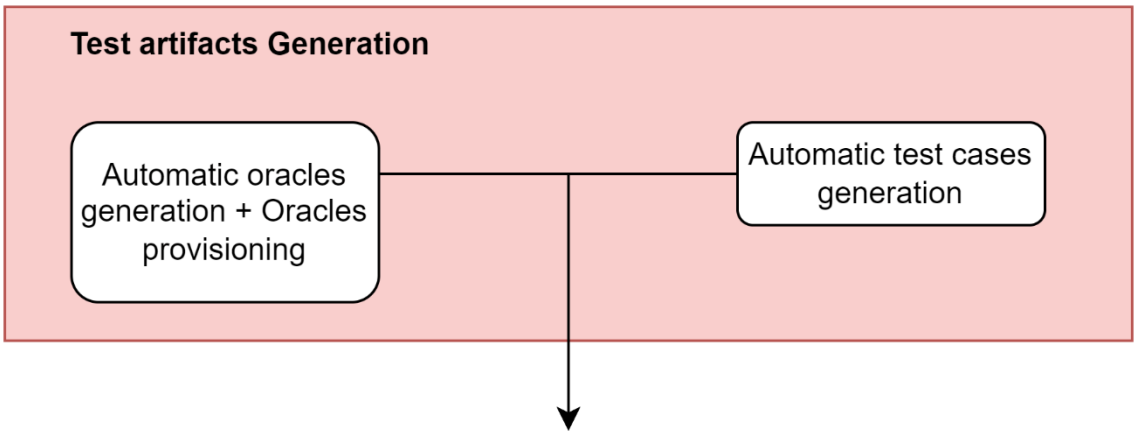


Figure 3. Workflow to Test Artifacts Generation

4.1.1 Test Case Generation & provisioning

The Test Generator microservice take 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*[10] 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 4.

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

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

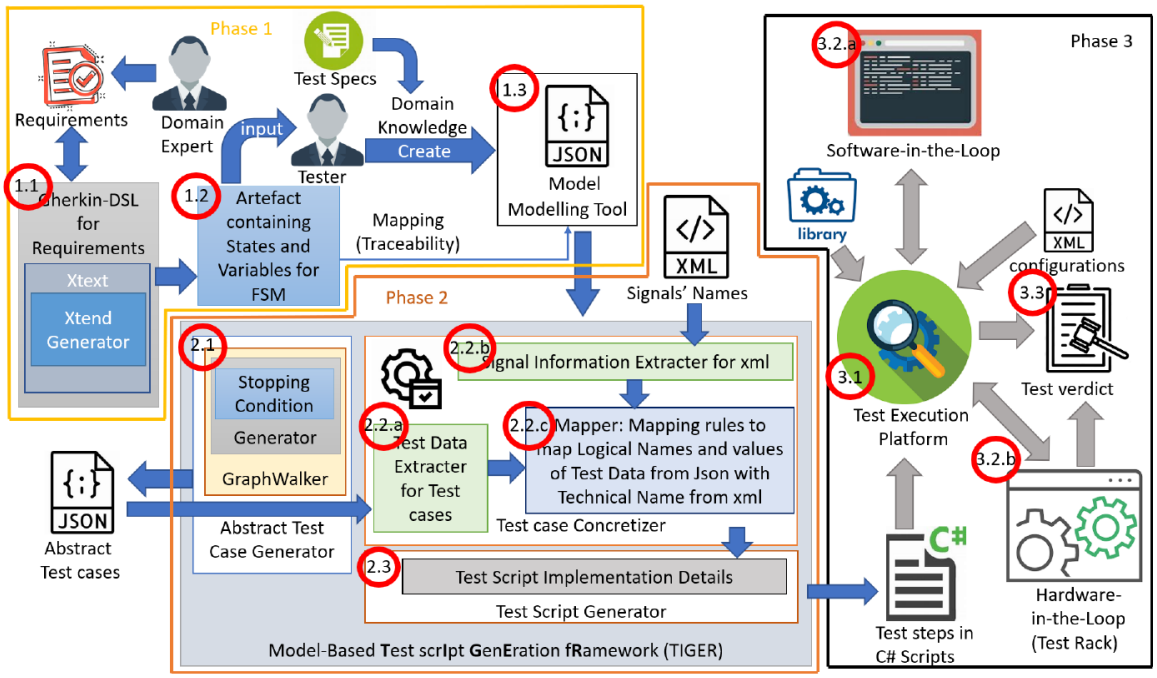


Figure 4. Sequence of the Test Case Generation

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[11] generator to extract the information from the Gherkin-like DSL produced in the Eclipse-based Xtext editor[12]. 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.

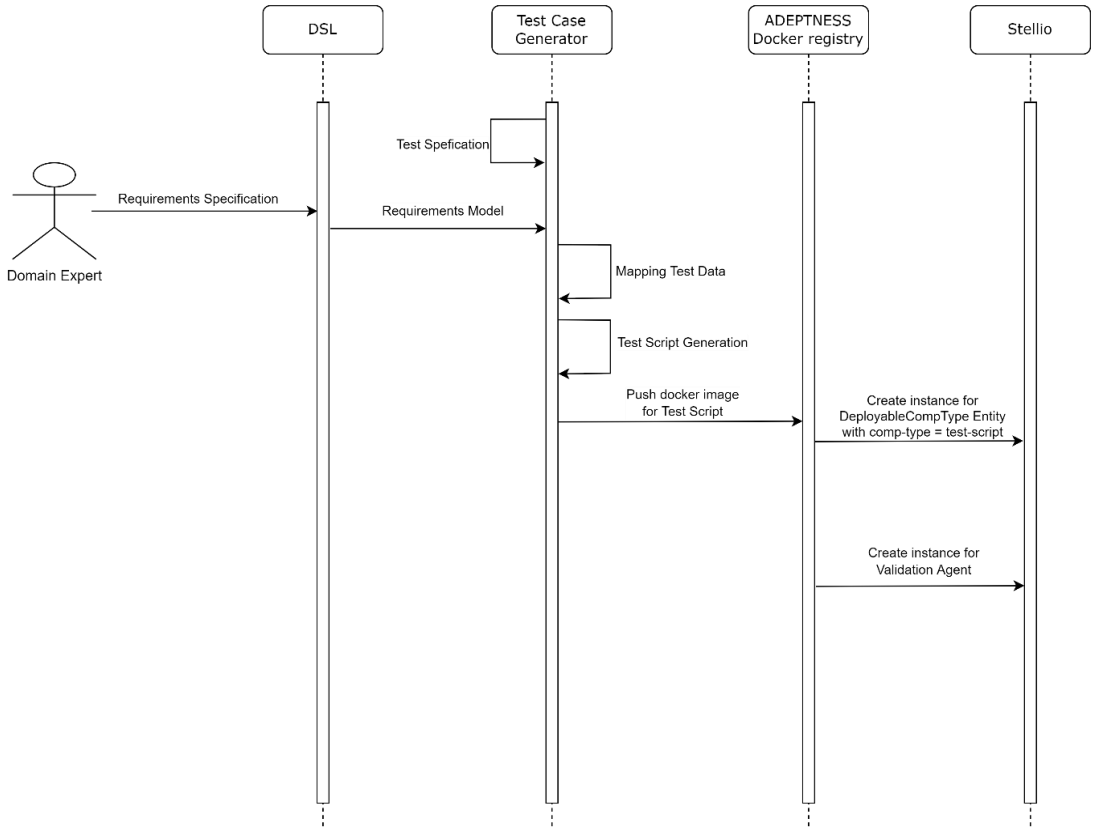


Figure 5. Test Case Generation and Provisioning Workflow

The sequence in Figure 5 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.

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

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

- Generate the target test scripts.

4.1.2 *The generated scripts and location of input data is then packed into Docker images, and provided to the Docker registry.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 2). 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.

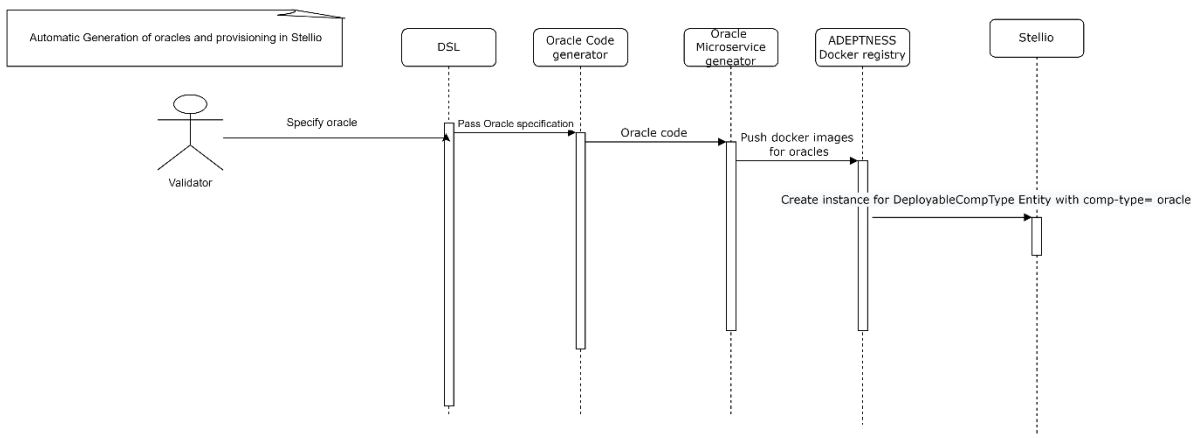


Figure 6. DSL-based Oracle Generation Workflow

At planning 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.

The definition has four key steps (see Figure 7):

- Resource Provisioning: Here other common components are defined as well. Mainly, deployable components, nodes, the CPS and CPSoS, etc. The SUT is defined during the validation plan definition step.
- Deployment Plan: Provisioning of the deployment plan, including the definition of the agents responsible of downloading and creating the resources in the target edge nodes.
- Monitoring Plan: Provisioning of the monitoring agents and variables to observe the SUT.
- Validation Plan: Provisioning of the agents, oracles and external tools, including test inputs, responsible for validating the SUT.
- JSON Generation from TaaS Interface: This step includes the aggregation of the deployment, monitoring and validation plans to the Adeptness plan, and the generation of the files and configurations required to run the plan.

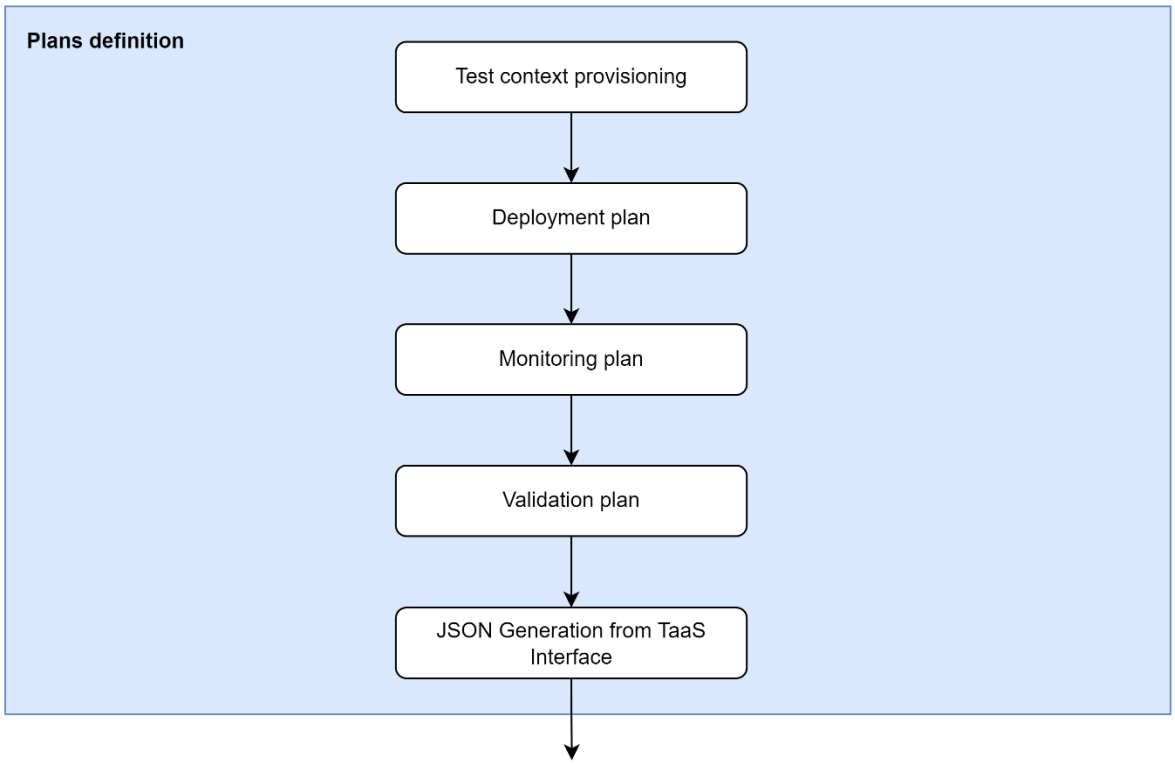


Figure 7. Planning Workflow

The following subsections provide details about the sequences in each of the planning 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 interface, 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 Specification

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 or container image (referencing MinIO[13] or a URL), and the kind of component, such as a CAN 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 the association between physical SUT and logical entities.

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.

For the specification, the user connects to the front-end, TaaS in Adeptness project, and provides a unique descriptive name, and a short description of the CPSoS.

The TaaS tool parses this information and creates a new CPSoS entity in Stellio context broker.

This sequence is depicted in Figure 8. In D5.1 – Adapted version of TaaS – Section 3.1.1 explains the visual interface in detail.

[13] https://min.io/

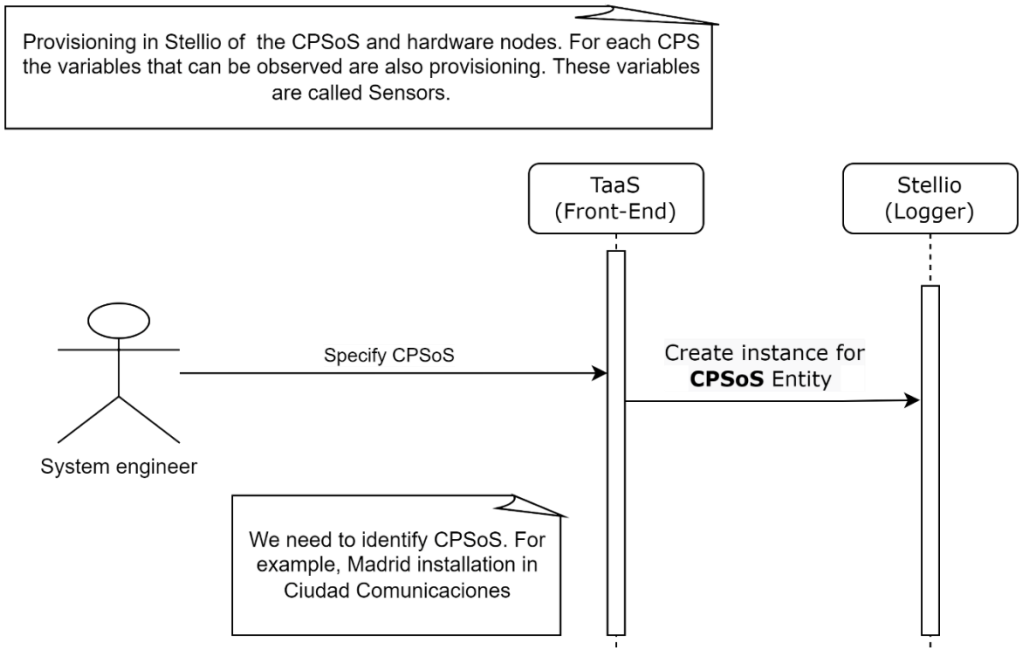


Figure 8. Sequence diagram for provisioning CPSoS in Stellio

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 be defined. After that, a unique descriptive name for CPS and a short description shall be provided, in order to identify the CPS and map accordingly to the edge node. See Figure 9 for the representation of the sequence diagram associated to the CPS specification.

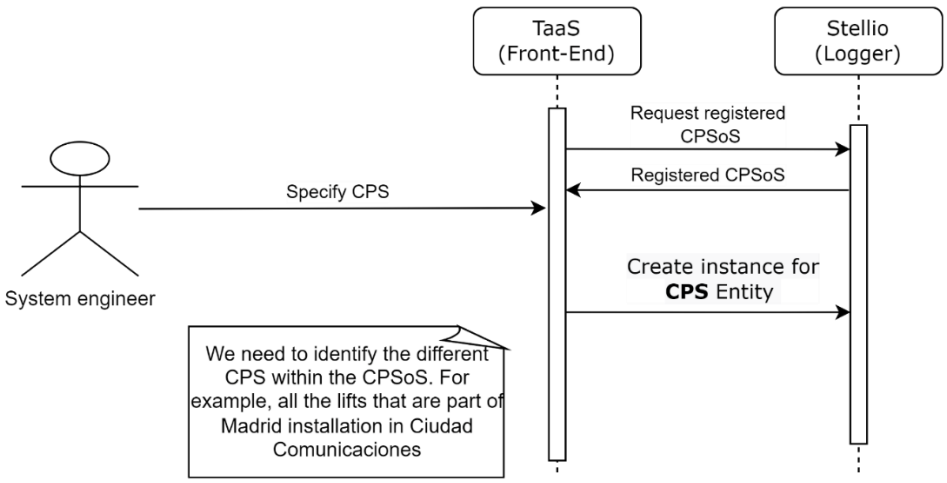


Figure 9. Sequence diagram for provisioning CPS in Stellio

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, such as data location. Finally, the sensor shall be associated to one of the defined monitoring agents, which is the way of notifying to Stellio at runtime the sensed data in the SUT.

The Figure 10 displays graphically the sequence described above.

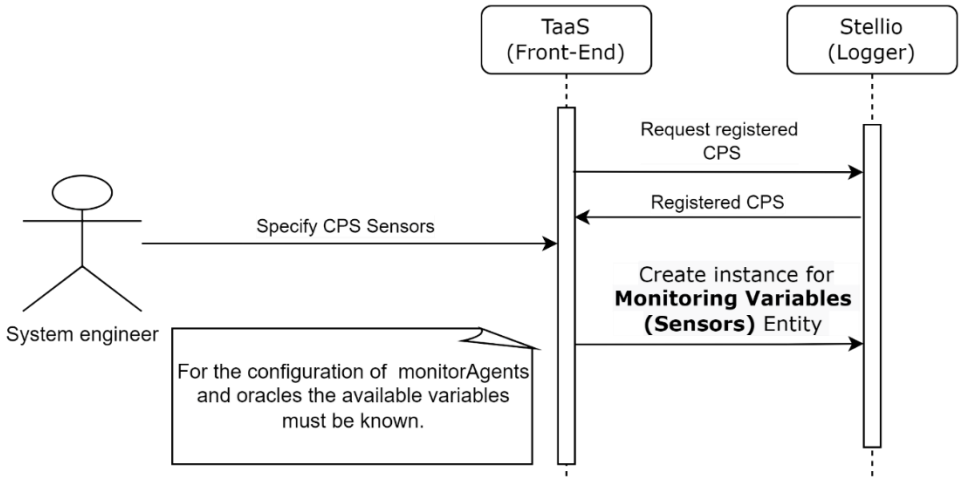


Figure 10. Sequence diagram for provisioning Sensors in Stellio

The sensors can be grouped intro Sensor Groups. This allows sharing common technical characteristics to all contained sensors such as publishing rate. The groups shall be defined after at least one leading Sensor is defined.

Hardware Nodes

The logical entity Nodes can be directly specified to Stellio through TaaS front-end. The Figure 11 provides the sequence for creating the hardware nodes.

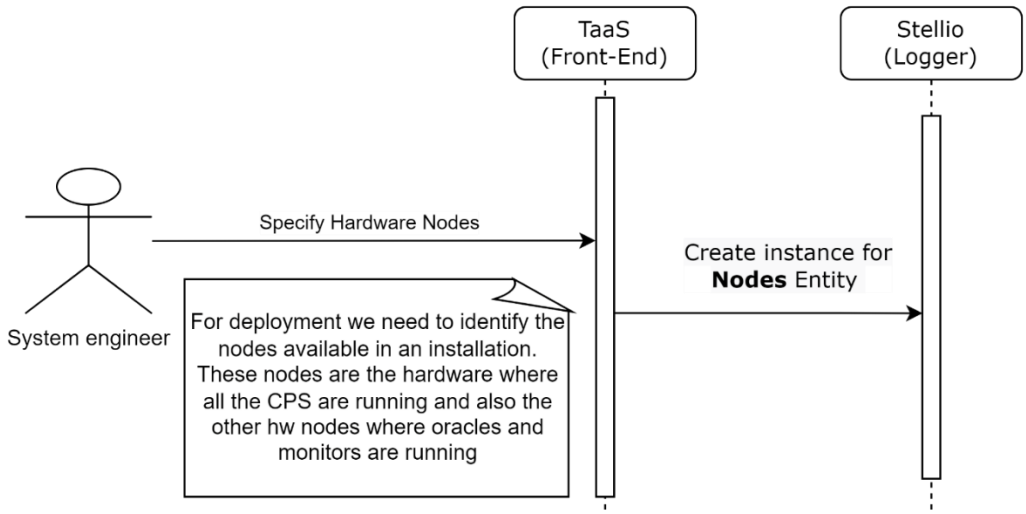


Figure 11. Sequence diagram for provisioning Nodes in Stellio

4.2.1.2 Deployable Components Specification

The next logical step is the specification of deployable components in Stellio through TaaS. TaaS provides an interface to declare in the Logger the following deployable component types:

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

This specification allows the specification and connection at deployment, monitoring and validation plans.

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 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 3.3.5), and provisioned to Stellio using the TaaS front-end specification available in Section 4.1.2.

The Monitor Agents are configured to read the proper SUT variables during the execution of the validation plan. These are specified at provisioning time 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 interface for specifying all these deployable components is common for all available types: monitor agent, system under test, external tool, validation agent, oracle or test input. This is due to the Entity stored in Stellio is the Deployable Component Entity.

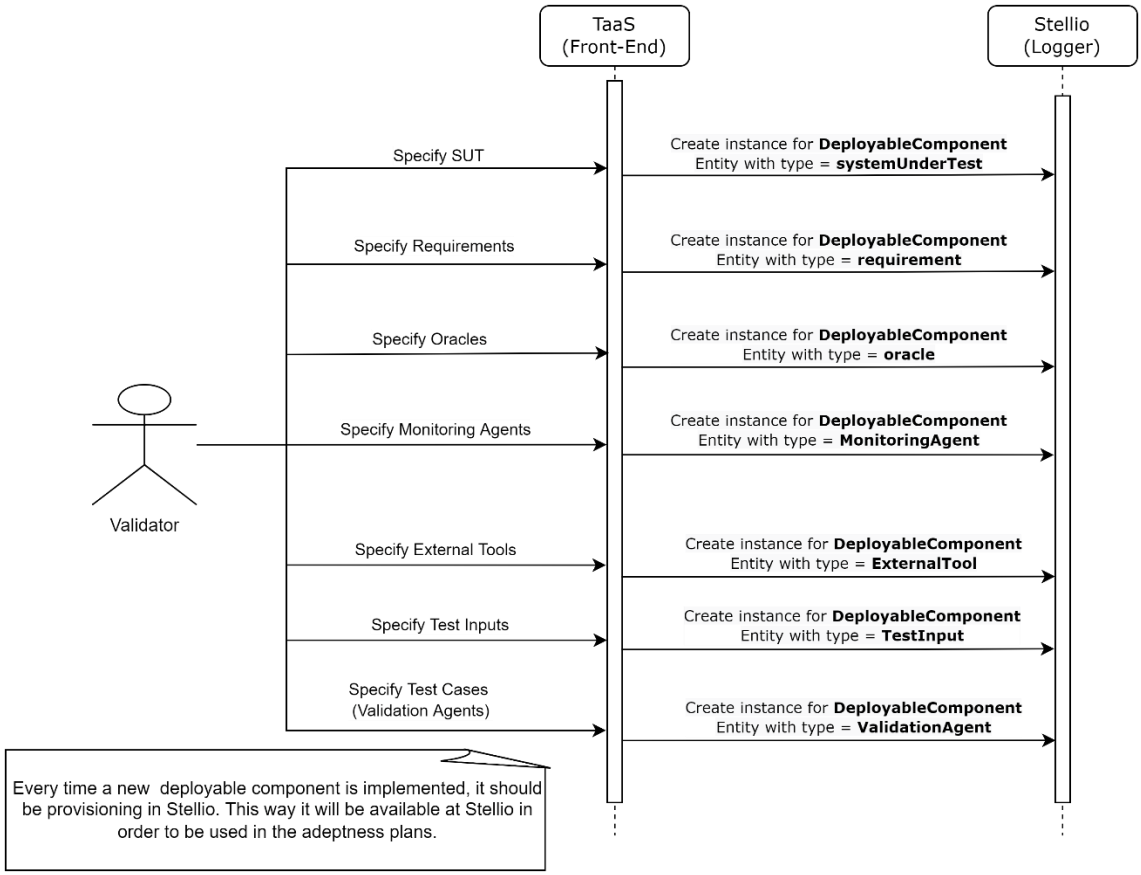


Figure 12. Specification of deployable component in Stellio

4.2.2 Creation of the Deployment plan

The deployment plan selects which images, registered in the system, are selectable for being deployed in the target system and how these connect to the SUT:

Deployment Association.

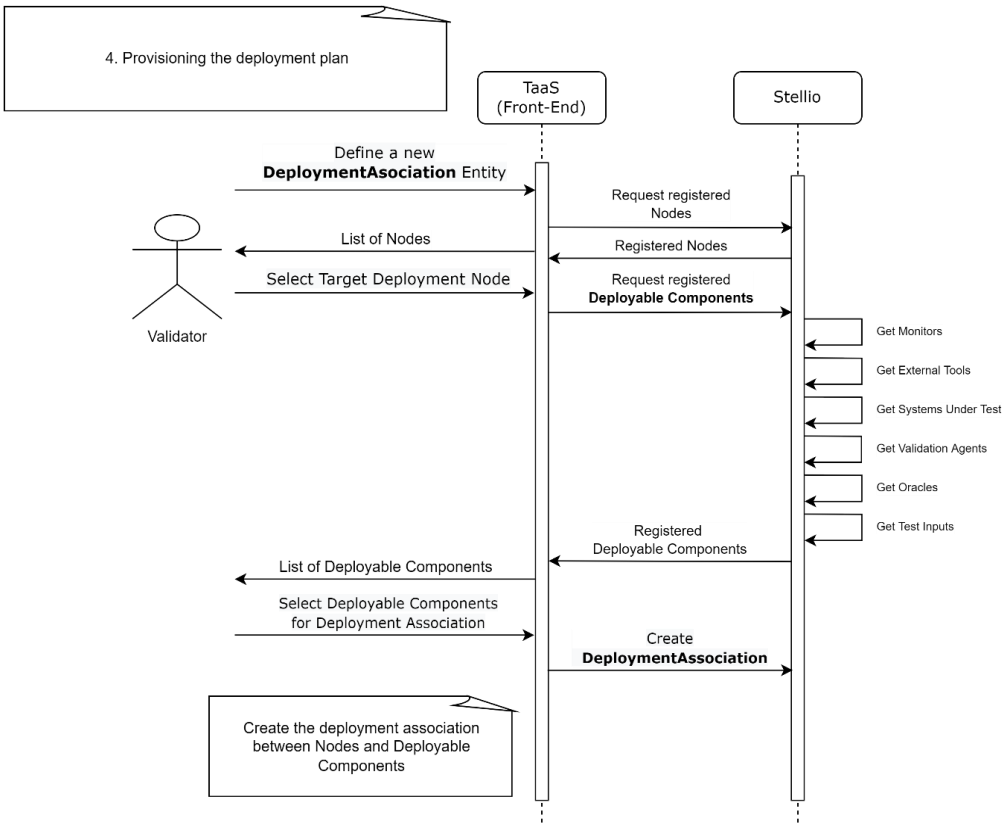


Figure 13. Deployment association creation sequence diagram

Components of type "Deployable" need to be associated with a node in order to retrieve, for each of them, rest port and IP, when TaaS will generate the deployment, monitoring and validation plans.

A deployment association consists of a linking of deployable components to (edge) nodes. The deployment association is later used as an entity to declare a deployment plan in the targeted edge nodes, by selecting the corresponding deployment associations to include in the deployment plan.

Regarding the deployment association, in a first step, the user (in this case, the validator) requests the edge nodes defined in the system. A deployment association can be only linked to a single edge node.

In the second step, the user selects using TaaS the available deployment components specified in the system. These deployment components can be either monitors, external tools, SUTs, validation agents, oracles or test inputs. Only one deployable component is part of the deployment association.

The deployment association simply links single deployable components to edge nodes. 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 deployment associations associate edge nodes to single deployable components, such as monitors, external tools, SUTs, validation agents, oracles or test inputs. In order to ease the generation of deployment plans, the specification of deployment plans consists only on selection previously defined deployable components-edge node associations. A deployment

plan can store as many deployment associations as defined in the system, containing the deployable components required for the deployment at validation targets. This can comprehend one or more edge nodes.

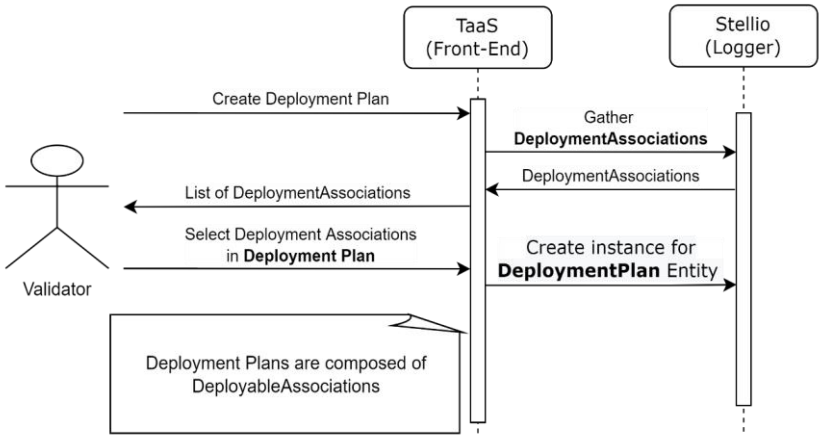


Figure 14. Deployment plan creation sequence diagram.

Deployment Plans are generated from a list of associations between Monitors 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.

In a second step, the user can select the set of deployment associations, which will be included in the plan. The deployment associations are simply the relationship between nodes and deployable components, in any of the available deployable component types: monitoring agents, SUTs, external tool agents, validation agents, oracles or test inputs.

The deployable associations can be then be included in the deployment plan, and thus, at deployment guided by the deployment orchestrator, be downloaded to the target node.

4.2.3 Creation of the Monitoring Plan

The monitoring plan depends on the deployment plan. Only available deployed monitor agents can participate on the monitoring plan. In Figure 15 contains the details on how a monitoring plan is defined.

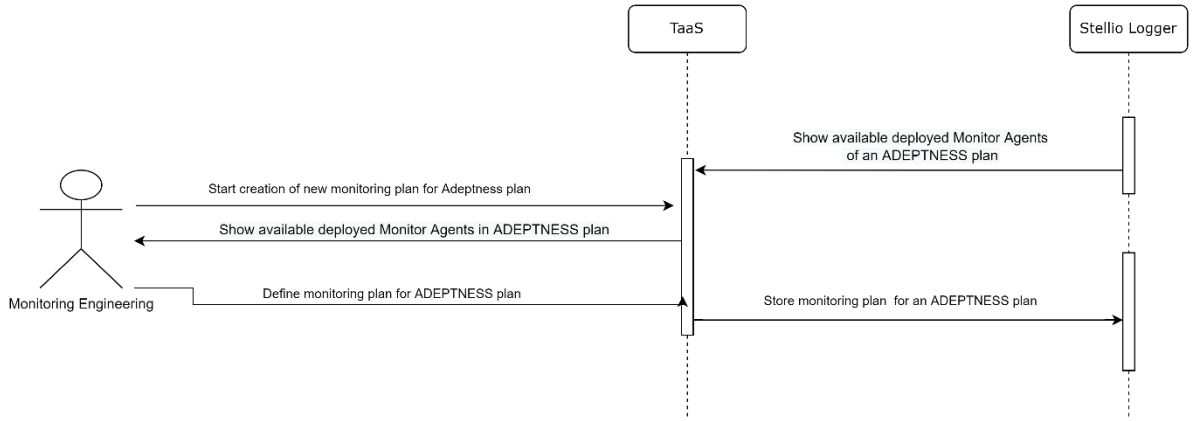


Figure 15. Monitoring plan configuration workflow

The available deployable components, already specified in the deployment plan, can be used for creating the monitoring plans. The monitoring orchestrator uses the generated JSONs as input. These are pushed by the automation server to it's API..

A monitoring plan includes at least information about the connection settings, such as MQTT port, the target node in which the plan will be executed, the associated monitoring agents, and the list of sensors with associated monitoring variables.

For the configuration of the monitoring plans, the TaaS front-end provides the interface to the user to create them.

For creating monitoring plans, TaaS requires to gather the available deployed monitor agents. These agents have been previously been defined in the context provisioning.

With the available monitoring agents, the monitoring engineer can select a subset of these, and create a monitoring plan, naming it accordingly.

The monitoring plan will be then be registered in Stellio.

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 16.

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.

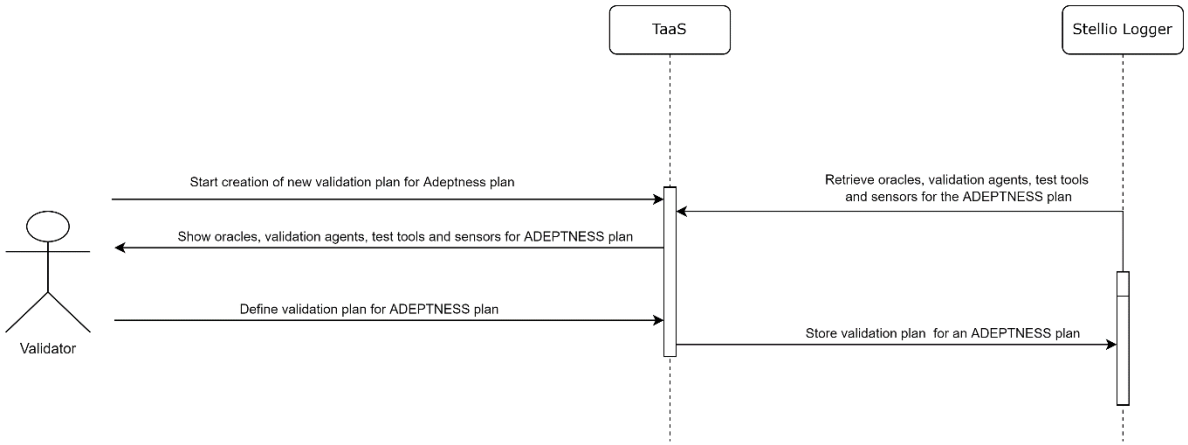


Figure 16. Validation plan configuration workflow

When a call is received on the validation plan, it is passed to the parser component of the microservice. The parser lends the parsed plan to the validation agent manager that manage the validation agents required for the execution of the validation plan.

The validation orchestrator receives validation plans and performs a parsing procedure for interpreting the contained validation elements.

The validation agent manager will then perform the required process to configure and start the validation on the validation agents and subscribe to the topics where the verdicts generated by each of the validation agents will be published.

4.2.5 JSON Generation from TaaS interface

As of now, JSON file plans are generated from TaaS but this should be done by the Stellio logger as soon as possible for a better tasks distribution between microservices. Thus, the generation is executed automatically from the *AdeptnessPlan* component page, which regroups all three ready Deployment, Monitoring and Validation plans, onto test launch click, by the user. However, each plan can still be generated specifically without starting a test, from each plan component page. This can be useful as a debugging feature especially in the early days of this workflow usage.

In order to build the files, TaaS needs to retrieve all needed components within each plan first, by following the graph data model relationships. From the *AdeptnessPlan* at the top, to the *OracleInputs* at the bottom which are bound to *Sensors*, *Monitors*, etc. In the meantime, an algorithm starts building the JSON files from the received data, in respect with the expected format as shown on the Validation plan object sample below:

```
urn: 'urn:ngsi-ld:ValidationPlan:valplan07-12',
agents: {
  'urn:ngsi-ld:ValidationAgent:ValWithFile': {
    endpoints: {
      rest_endpoint: '123.456.789:(restPort not found)',
      mqtt_endpoint:
        'adms/v2/validation-agent/urn:ngsi-ld:ValidationAgent:ValWithFile',
    },
    tests: {
      'urn:ngsi-ld:TestCase:TEST-16-12': {
        sut: {
          name: 'urn:ngsi-ld:SystemUnderTest:SUT-07-12-n°3',
          type: 'algorithm',
          configured_on: 'urn:ngsi-ld:Node:Edge2',
        },
        external_tools: {
          'urn:ngsi-ld:ExternalTool:ExternalToolWithInput': {
            tool_type: 'elevate',
            endpoints: {
              rest_endpoint:
                'Could not retrieve rest endpoint for urn:ngsi-ld:ExternalTool:External
              mqtt_endpoint:
                'adms/v2/external-tool/urn:ngsi-ld:ExternalTool:ExternalToolWithInput',
            },
            tool_settings: '{}',
          },
          'urn:ngsi-ld:ExternalTool:ExternalToolRMWithinput': {
            tool_type: 'elevate',
```


Figure 17. Validation plan object generated sample

All plans are extracted to the already defined plans specified in the previous subsections: the deployment plan, the monitoring plan and the validation plan, and creating a new entity, responsible of instantiating the plans at the orchestrators, and downloading the deployable components to the specified edge nodes.

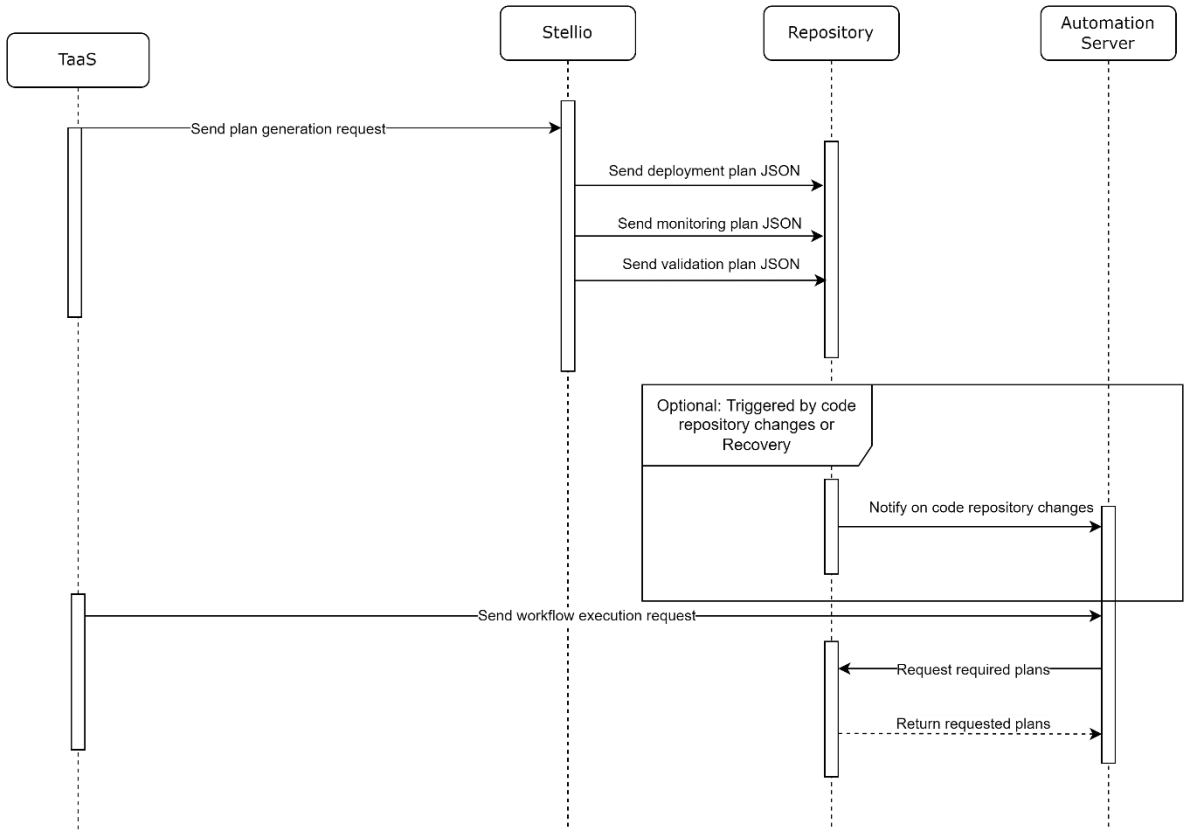


Figure 18. JSON Generation Workflow

The request is directly sent from TaaS, which contacts with Stellio Logger to explore the deployment, monitoring and validation plans, and format them accordingly in JavaScript Object Notation (JSON)[14] format.

The generated plans are then stored in a repository (in our case, MinIO), which acts a cloud fileserver for the execution of the plans in the next workflow step.

In the case of modifications on the plans or when needed a recovery strategy to re-establish the health of the edge nodes, the repository notifies the Automation Server about the changes on the code.

The next step is the installation of the Adeptness plan into the orchestrators, through the Automation Server, which initiates the plan execution pipelines.

4.3 Execution Workflow

The Execution workflow is the one, which targets the SUT for the validation and monitoring in the CPSoS.

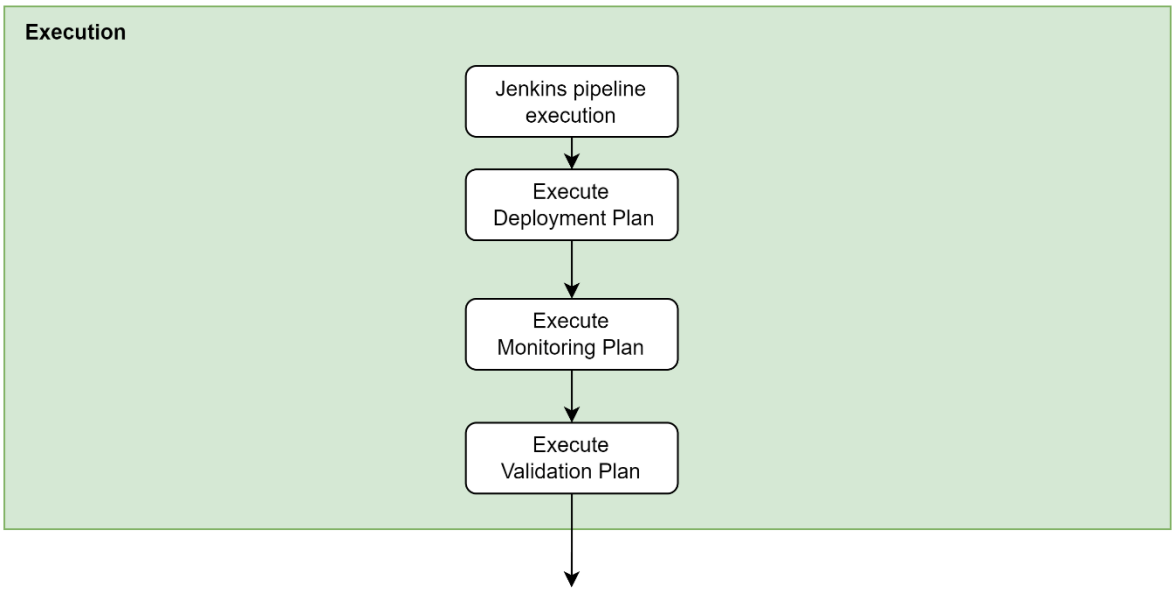


Figure 19. Execution Workflow.

In the previous stage, the plan definition, the user stablished the set of deployable components and sequence of the validation strategy and monitoring variables. In this stage, we execute the available Adeptness plan. This can follow a SiL strategy, in which the importance is set into the logic beyond the controlling software, including algorithms, or HiL strategy, in which the target SUT plays a main role in a real system. Both strategies can be scheduled to be executed at this stage. This highly depends on the instantiation of the deployable components, and the existence of one or more validation artifacts, such as external tools, test inputs, or oracles.

An Adeptness plan can contain both SiL and HiL execution pipelines, only SiL or only HiL, depending on the design-operation continuum validation strategy.

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

4.3.1 Jenkins Execution Pipeline (Trigger Adeptness plan)

The Jenkins execution pipeline is executed from the Automation Server, which comprehends, essentially, gathering the Adeptness plan, and place in the corresponding Orchestrators: Deployment, Validation and Monitoring.

There are three pipelines executed sequentially:

- Load and Configuration of the Deployment plan in Deployment Orchestrator, including the configuration and instantiation of deployable components in edge nodes.
- Load and Configuration of the Monitoring plan(s) in Monitoring Orchestrator, once deployable components, specially agents, are instantiated in edge nodes. This pipeline highly depends on the existence of monitoring agents. If no agents are configured, then the pipeline moves to the validation plan.
- Load and Configuration of the Validation plan(s) in Validation Orchestrator. The deployment orchestrator should have started the corresponding validation-related deployable components in the deployment plan.

The full sequence is provided in Figure 2020.

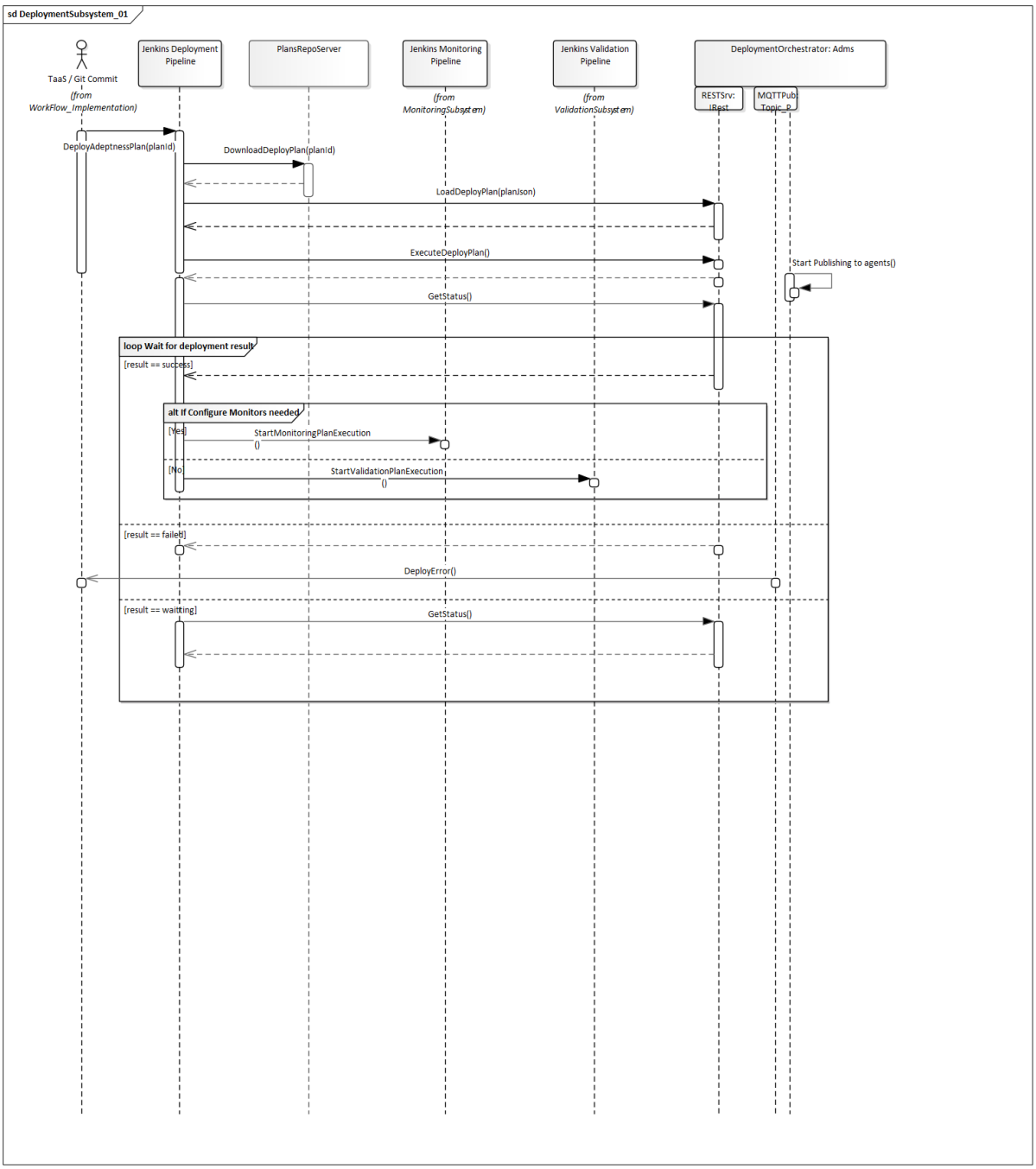


Figure 20. Adeptness plan download and execution sequence.

4.3.2 *Deployment Plan execution*

The deployment plan contains the procedure to deploy all agents established in the Adeptness plan. This includes the monitoring and validation agents' instantiation in the target platform (near SUT).

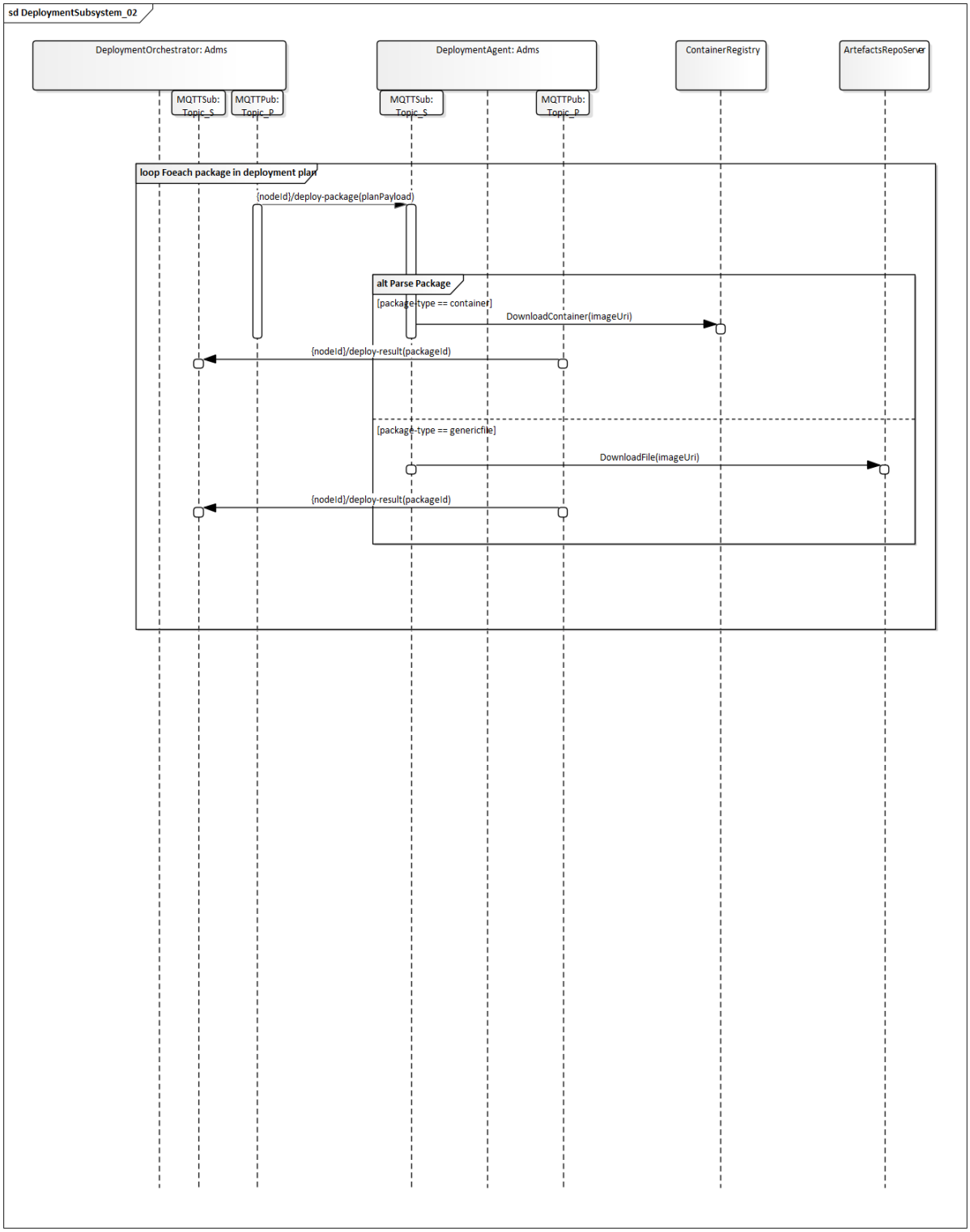


Figure 21. Download and deployment sequence in edge nodes of deployable agents.

Once the Automation Server has ordered the execution of the plan already uploaded in the orchestrator, the orchestrator will extract from the plan the nodes it has to communicate with and prepare the MQTT messages to send to the deployment agent in each node the orders to download the components and launch them 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. In turn, the orchestrator will subscribe to the result topic (**adms/v2/deployment-orchestrator/{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. Finally, the orchestrator will end the deployment phase and go back to the "*configured*" state.

4.3.3 Monitoring Plan execution

On start-up, the monitoring orchestrator is considered to not be configured and will be awaiting a plan. In this "*unconfigured*" state, the orchestrator can receive the following requests:

- A request to load a monitoring plan, which will be internally stored.
- A request to launch a monitoring plan, which will be responded to with an error, as no plan has been sent.

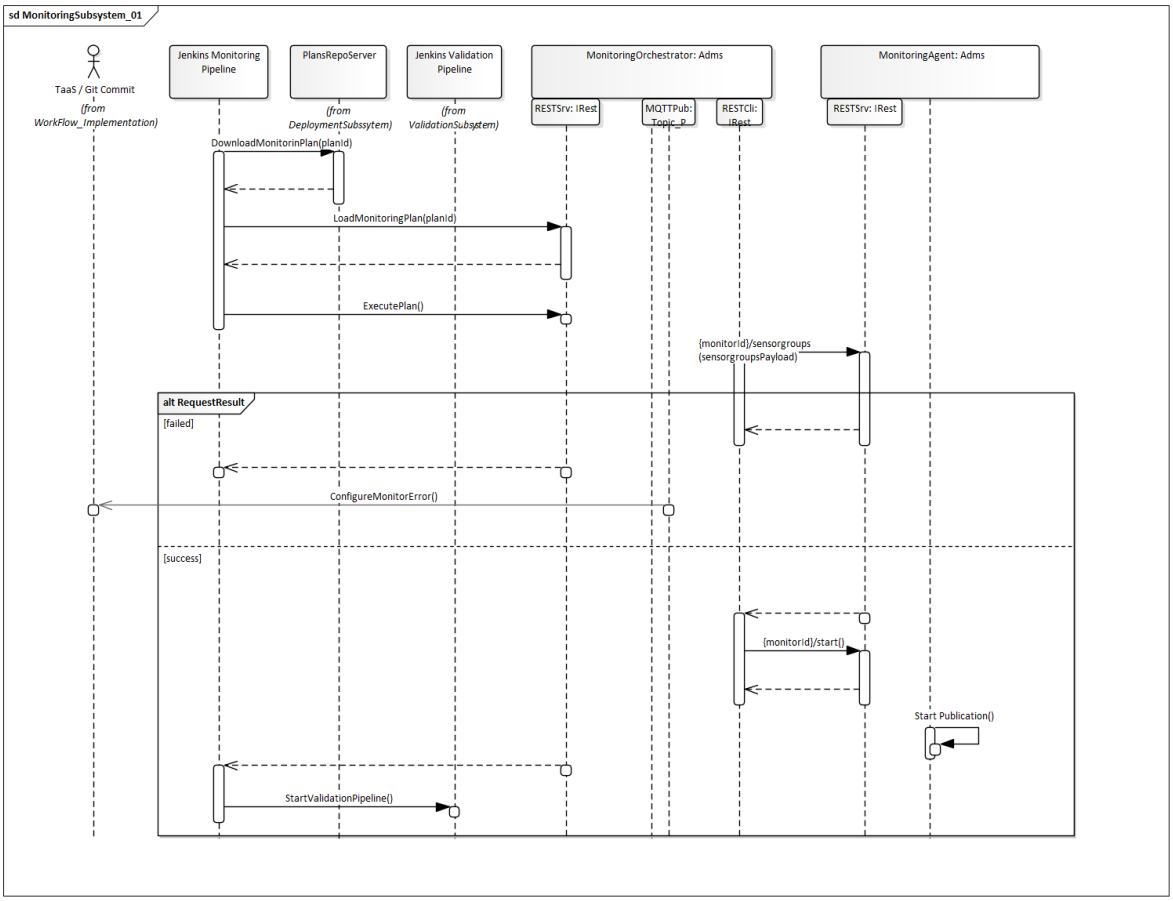


Figure 22. Monitoring plan download and configuration sequence.

When a plan is received and the orchestrator is configured, it will be able to receive the following requests:

- A request to load a new monitoring plan. If the previous plan was launched, then it will be temporarily stored internally.

- A request to launch the monitoring plan will make the orchestrator launch the plan by accessing each of the monitors specified in the plan. Through the rest interface offered by each monitor, the orchestrator will be able to configure the variables that the monitor should sample and publish via MQTT.

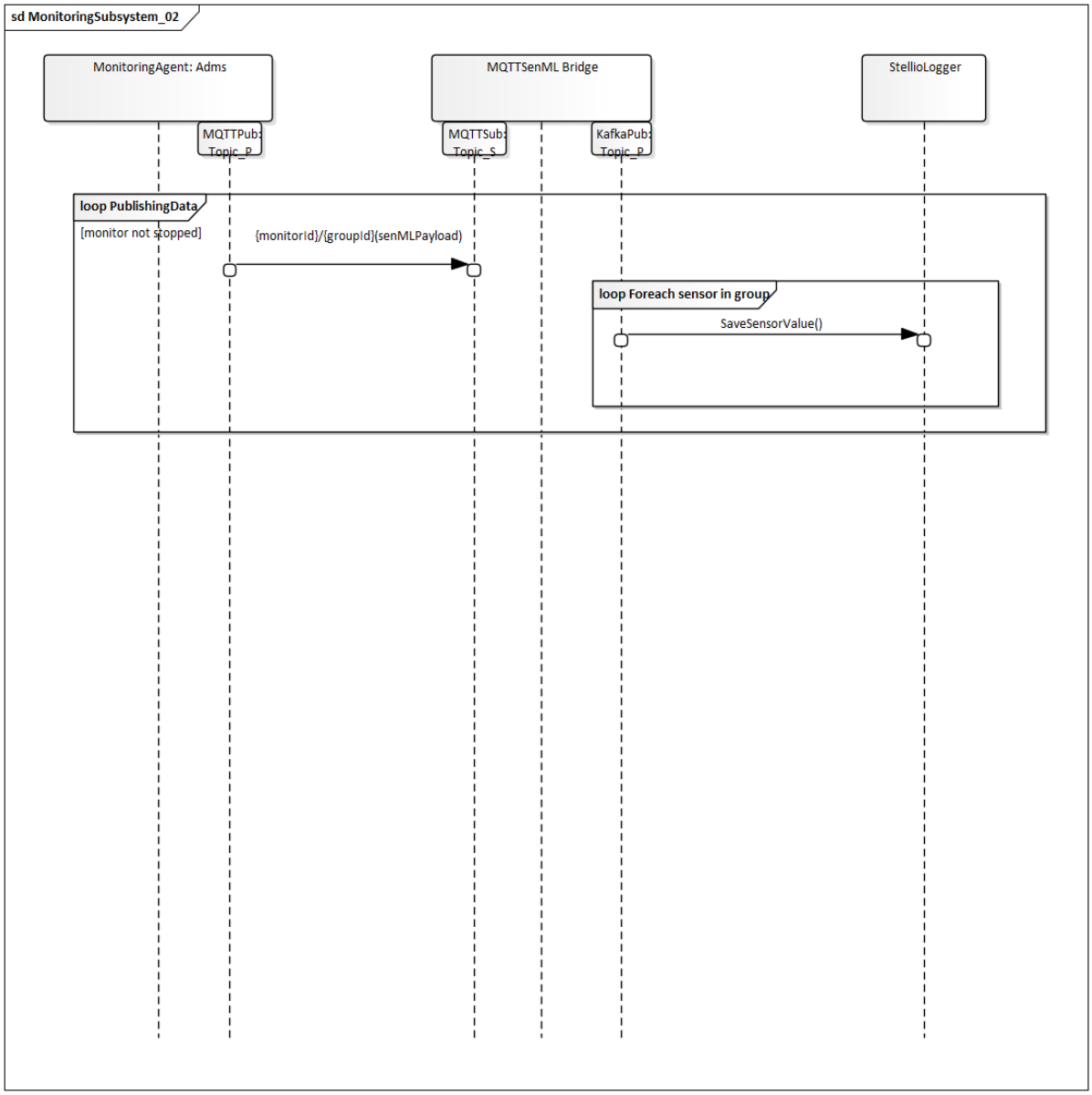


Figure 23. Execution workflow for monitoring plan at runtime

At runtime, the execution workflow for monitoring (see Figure 23), the monitoring agents publish regularly using MQTT the collected monitoring variables using SenML format. The received of this information is the MQTT Broker, which sends to Stellio context broker the sensor values received in MQTT Broker through the MQTT Bridge.

4.3.4 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 automation server, the validation orchestrator receives a validation plan containing all the required information for performing a validation procedure via a REST API call. 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 automation server, 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.

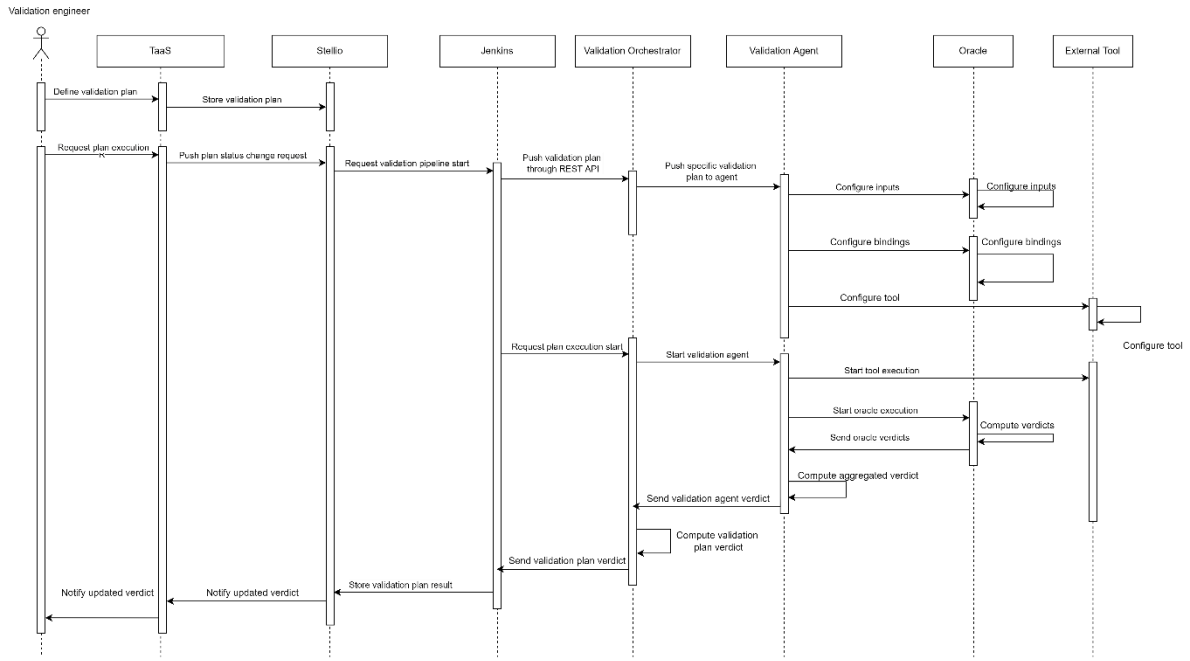


Figure 24. Validation workflow configuration and execution

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.

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 automation server 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.

In the following subsections, Elevator external tools and Train Control Management System (TCMS) system provide the realisation of the validation strategy for two domains in CPSoS.

4.3.4.1 Elevator

In the Elevation use case, we would have a validation agent per test level. The validation orchestrator will forward the subset of the validation plan corresponding to each of the validation agents, and delegate the execution of the validation to each of them.

In this case, the plan will require the configuration of an external tool containing the Elevate simulator, where a simulation using the SUT to be evaluated will be performed. On SiL test level, Elevate will perform a simulation of the Lift system and will run the SUT. Operational values will be published over MQTT via the developed trace library. These values will be collected and evaluated by oracles that will obtain a verdict for the behaviour of the system running on SiL.

After a successful evaluation on SiL, the evaluation will take place on HiL. On HiL, the evaluation will take place on Orona's hardware in the loop test bench. In this case, operational values will be mainly collected from the CAN bus and additionally from the instrumentation of the SUT with the trace library.

During operation, continuous monitoring and validation will be performed in order to detect anomalous behaviour of the system.

4.3.4.2 Train Control Management System (TCMS)

The test scripts generated by TIGER can be executed on SiL and HiL levels after including specifically designed libraries and integrating a configuration file in our case company's C# generic test framework. The configuration files and libraries provide an adaption layer for the execution of generated test scripts by configuring the environment and simulations at SiL and HiL level. After the execution of generated test scripts on either SiL or HiL level, the test framework generates a test verdict containing a detailed list of passed and failed test steps.

The test of the TCMS System uses test setups using the following test equipment: Virtual CCU, Virtual Train and Test rack. The Virtual CCU is a tool, which runs at a test PC, based on the Soft-TCMS technology and exposes a virtualized environment for the SW applications at a CCU. It provides the same functional

environment as the real CCU (process scheduling, OS calls, communication means) including the capability for signal stimulation, observation and debugging. The Virtual CCU can be built directly from the TCMS project files. The test rack is a setup, which hosts the TCMS System of one vehicle comprising the real TCMS devices. In addition, the test rack hosts the test execution environment and supporting tools (e.g. remote controllable switch to power TCMS devices). Like the Virtual CCU, the Virtual Train is a tool, which runs at a test PC, based on the Soft-TCMS technology and exposes a virtualized environment for the SW applications at a CCU as well as the configuration of the additional TCMS devices (switches, gateways) including the TCMS bus systems and the test execution environment.

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: Generation at design-time and detection at runtime: generates profiles related to unforeseen situation 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 255.

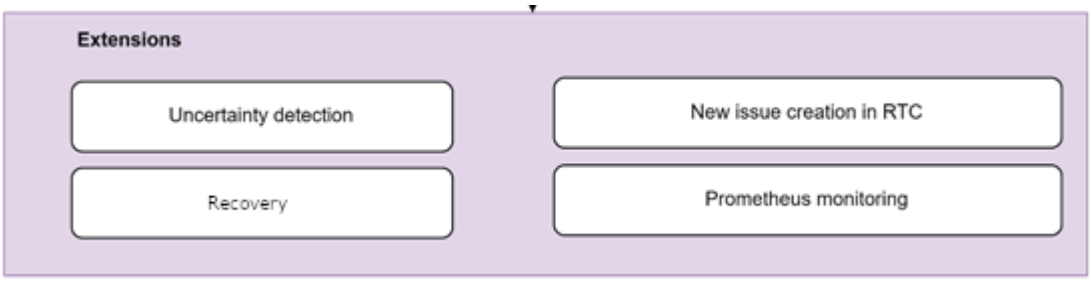


Figure 25. Subsystems for knowledge extraction from operational data

These subsystems are explained in detail in the following subsections.

4.4.1 Uncertainty Subsystem: Generation + Detection

The uncertainty subsystem 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 determined. If a poor robustness is due to, e.g., software updates, then the corresponding recovery action should be taken.

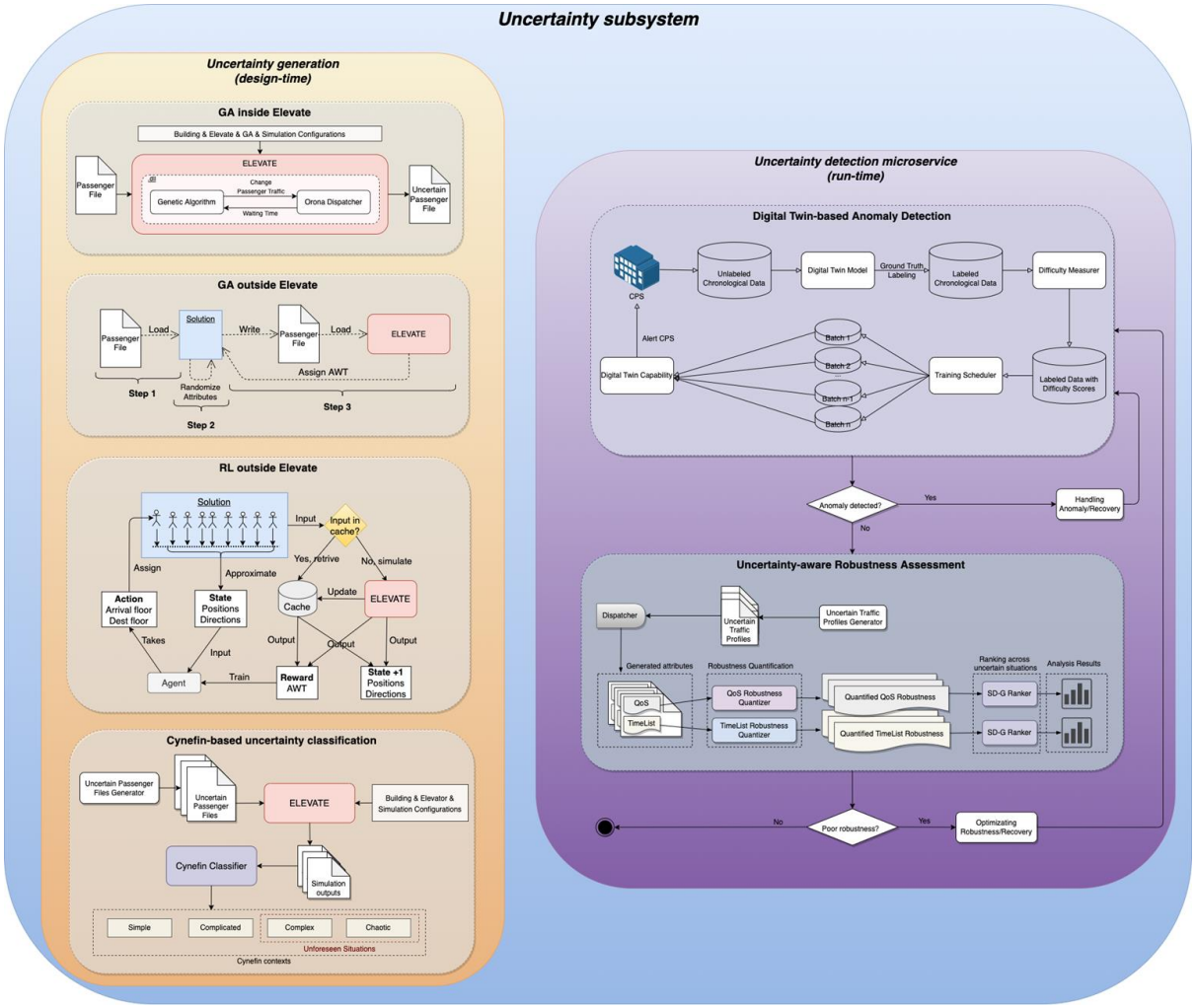


Figure 26. Overview of uncertainty subsystem

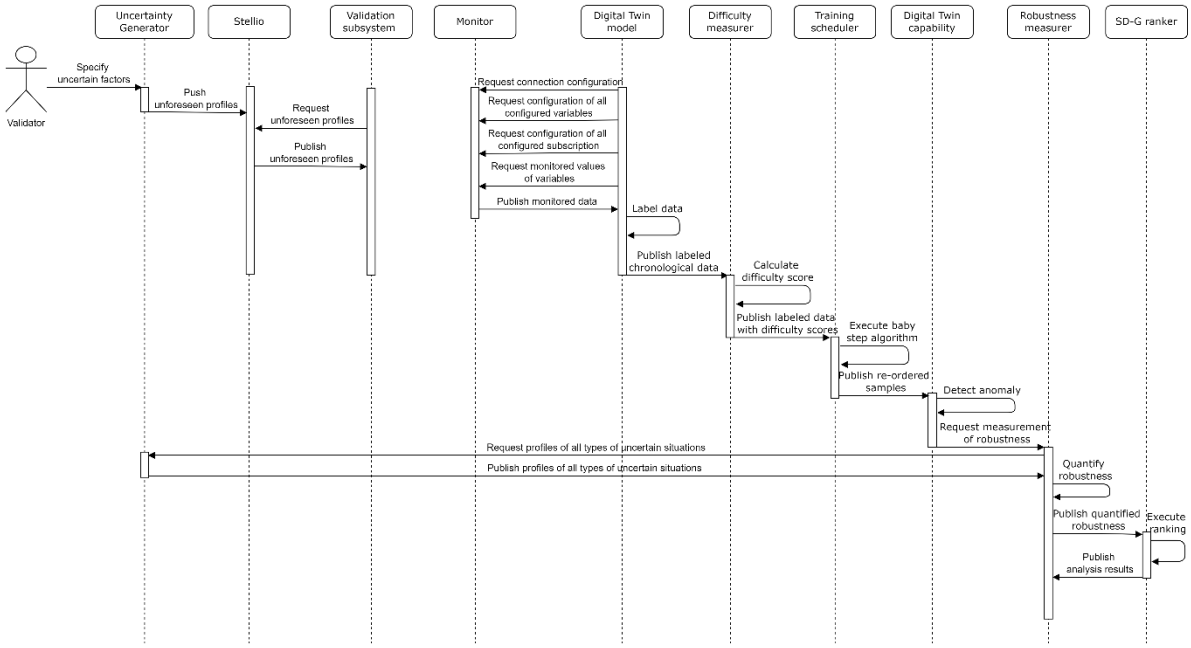


Figure 27. Sequence diagram for the workflow of uncertainty subsystem

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 run-time

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 CPS, 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.

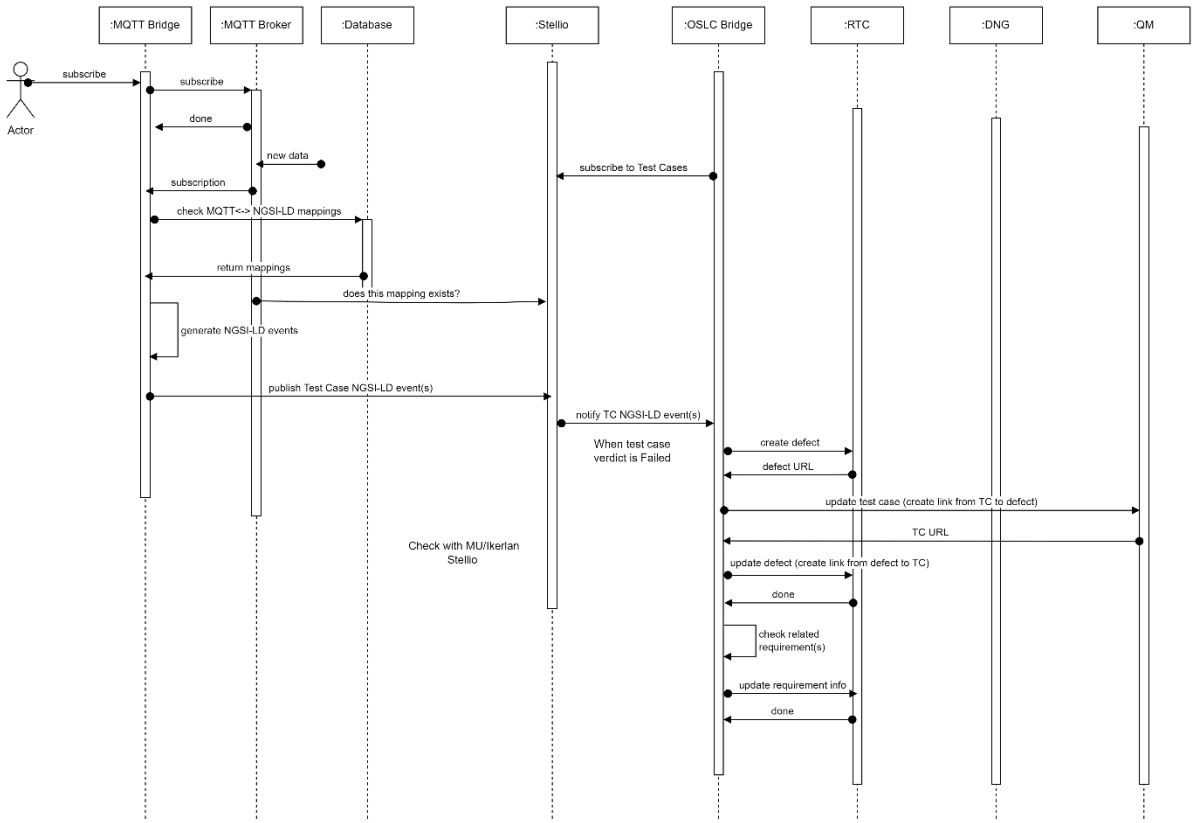


Figure 28. OSLC Traceability from the Stellio subscription

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.

The Figure 288 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.

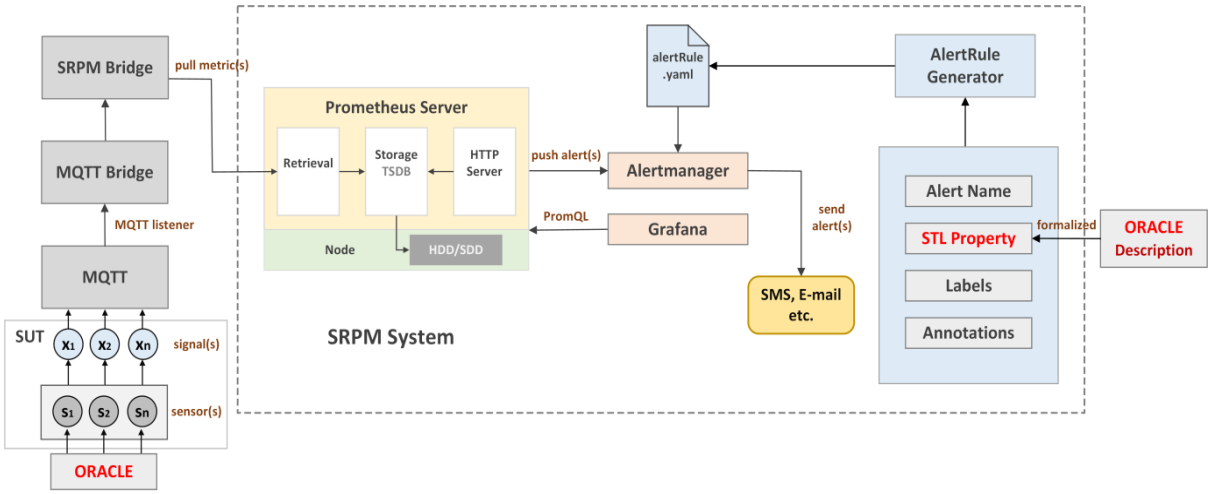


Figure 29. Architecture of SRPM and SRPM Bridge to MQTT

As shown in Figure 29, 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*[15] (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 - bridge the MQTT and Prometheus

The implementation of bridging the MQTT and Prometheus illustrated in Figure 30.

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

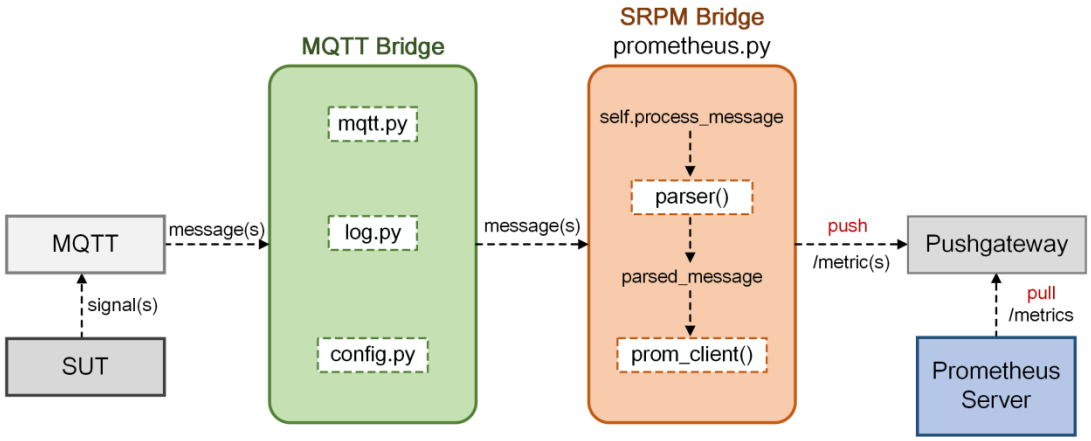


Figure 30. Implementation of bridging the MQTT and Prometheus

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

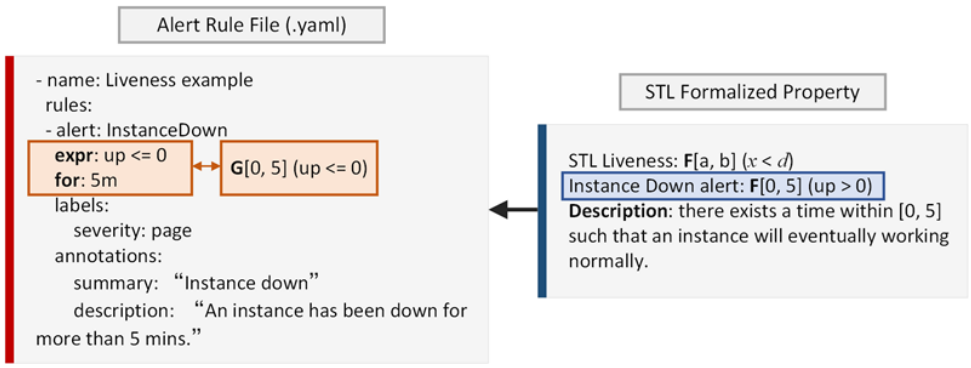


Figure 31. Example of transformation for Alerting Rules in Prometheus

As shown in Figure 31, 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 32.

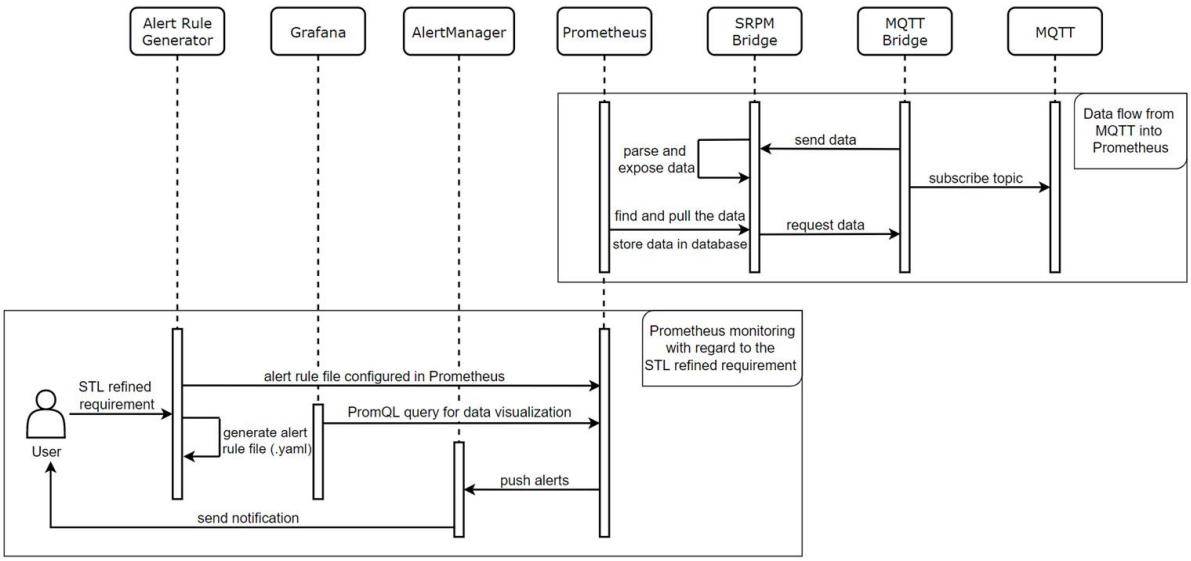


Figure 32. Sequence diagram for the Prometheus workflow based on the configuration capabilities

5 CONCLUSIONS

This document represents the first 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.

This deliverable and the deliverable D5.2 – Toolchain first release – will follow up with a refined deliverable, named D5.5 – Workflow and toolchain documentation final release – with the experience of adopting Adeptness approach to industrial use cases, after gathering feedback on the proposed workflow and toolchain.

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

This document reflects the views of the author(s) and does not necessarily reflect the views or policy of the European Commission. Whilst efforts have been made to ensure the accuracy and completeness of this document, the Adeptness consortium shall not be liable for any errors or omissions, however caused.