# D3.2 Edge/Cloud orchestration tools I

| Document Identification | | | |
|---|---|---|---|
| **Status** | Final | **Due Date** | 31/05/2021 |
| **Version** | 1.0 | **Submission Date** | 01/06/2021 |

| | | | |
|---|---|---|---|
| **Related WP** | WP3 | **Document Reference** | D3.2 |
| **Related Deliverable(s)** | D2.3 Pledger Overall Architecture v1.0 | **Dissemination Level (*)** | PU |
| **Lead Participant** | ICCS | **Lead Author** | Alexandros Psychas |
| **Contributors** | ATOS | **Reviewers** | Olga Segou,<br>Ioannis Sarris (INTRA) |
| | | | Verena Stanzl (FILL) |

| **Keywords:** |
|---|
| Cloud, Edge, Orchestrator, Monitoring, Deployment, Scaling, Migration, Performance, Container, Kubernetes, Docker, Demo |

This document is issued within the frame and for the purpose of the PLEDGER project. This project has received funding from the European Union's Horizon2020 Framework Programme under Grant Agreement No. 871536. The opinions expressed and arguments employed herein do not necessarily reflect the official views of the European Commission.



(*) Dissemination level: **PU**: Public, fully open, e.g. web

# Document Information

| List of Contributors | |
|---|---|
| Name | Partner |
| Antonio Castillo Nieto | ATOS |
| Alexandros Psychas | ICCS |
| Orfefs Voutyras | ICCS |

| Document History | | | |
|---|---|---|---|
| Version | Date | Change editors | Changes |
| 0.1 | 18/02/2021 | Antonio Castillo Nieto (ATOS) | Document template and initial ToC provided |
| 0.2 | 12/4/2021 | Alexandros Psychas(ICCS) | Initial versions with updated ToC and Chapter 1,2 |
| 0.3 | 26/4/2021 | Antonio Castillo Nieto (ATOS) | Chapters 3 and 4 finalised |
| 0.4 | 10/5/2021 | Alexandros Psychas (ICCS) | Chapter 5 and 6 finalised |
| 0.5 | 20/5/2021 | Verena Stanzl (FILL) | Internal review 1 |
| 0.6 | 24/5/2021 | Olga Segou,<br>Ioannis Sarris (INTRA) | Internal review 2 |
| 0.8 | 24/5/2021 | Alexandros Psychas (ICCS) | Version for quality review |
| 0.9 | 28/05/2021 | Carmen San Román (ATOS) | Quality assurance review |
| 1.0 | 31/05/2021 | Lara López (ATOS) | Final version to be submitted |

| Quality Control | | |
|---|---|---|
| Role | Who (Partner short name) | Approval Date |
| Deliverable leader | Alexandros Psychas (ICCS) | 24/05/2021 |
| Quality manager | Carmen San Román (ATOS) | 28/05/2021 |
| Project Coordinator | Lara López (ATOS) | 31/05/2021 |

# Table of Contents

# List of Tables

# List of Figures

# List of Acronyms

| Abbreviation / acronym | Description |
|---|---|
| CPU | Central Processing Unit |
| DB | Database |
| DSS | Decisions Support System |
| EC | European Commission |
| E2CO | Edge to Cloud Orchestrator application |
| FC | Functional Component |
| GPU | Graphics Processing Unit |
| IaaS | Infrastructure-as-a-Service |
| Mx | Project Month x |
| MVP | Minimum Viable Product |
| QoE | Quality of Experience |
| QoS | Quality of Service |
| REST | Representational State Transfer |
| SaaS | Software-as-a-Service |
| SLA | Service Level Agreement |
| WP | Work Package |

# Executive Summary

This document acts as a **description report** accompanying deliverable "D3.2 – Edge/Cloud orchestration tools", one of the three **prototype** deliverables of Work Package "WP3 – Performance, QoS and orchestration mechanisms", and more specifically, that of Task "T3.2 – Edge/Cloud Orchestration tools". This is the first version of the deliverable, with an updated and final version to be provided in project month 30 (M30).

T3.2 focuses on the edge/cloud orchestration tools and the necessary refinements that need to be performed in order to provide the necessary Pledger functionality. Emphasis is given on the usability with REST-based interfaces and graphical support. Increased configurability and integration capability are also targeted, linking to the databases backend and supporting the on-the-fly installation of user-defined strategies for edge/cloud provider selection based on the results of "T3.1 – Performance Measurements and Classification". Another target is the improved application life cycle that supports the updating of already deployed software components. This allows a continuous operation of cloud applications and is a must for the commercialisation of the project results. This task also works on the challenge of predicting the dynamics of edge computing computation based on the Quality of Service (QoS) and Quality of Experience (QoE) requirements of applications, thus further enabling edge/cloud orchestration.

Following the activities of the Task, this deliverable is focused on the Pledger **Orchestration subsystem**, one of the core subsystems of Pledger. The subsystem is responsible for managing the orchestration of containerized applications (actions related to the deployment of applications, infrastructure scale up/down, etc.).

# 1 Introduction

## 1.1 Purpose of the document

The scope of the current deliverable (**D3.2**) is to provide functional and technical description of the main functionalities, functional components and services of the Pledger **Orchestration subsystem (T3.2)**. This deliverable accompanies the prototype during the first iteration of WP3 and provides installation and usage guidelines of tools of the subsystem as well as demos of the main functionalities.

## 1.2 Relation to other project work

The Orchestrator component and by extension T3.2 is tightly coupled with the other tasks (T3.1, T3.3) and components of the WP3. As depicted in the Section 2 (Figure 1) Orchestrator component is connected with both main subsystems in the Evaluation layer of the architecture. Orchestrator require data connections with both subsystems in order to function. Furthermore, the component as well as the deliverable is guided by the work performed in WP2 mainly as far as the architecture and Requirement analysis is concerned.

## 1.3 Structure of the document

This document is structured in 6 major chapters:

- **Chapter 2** presents the functional description of the tools in alignment with the overall architecture of Pledger.
- **Chapter 3** presents the technical description of the tools (data model, interfaces, and component architecture).
- **Chapter 4** presents the installation and usage guide of the tools.
- **Chapter 5** presents de Demonstration of the tool with the description of the validation scenarios.
- **Chapter 6** presents the conclusions and next steps for this first iteration of the tool.

# 2 Functional description

This chapter introduces the main functionalities and components of Pledger developed within Task "T3.2 – Edge/Cloud Orchestration tools" and provides useful information related to the development of the Pledger **Orchestration & Configuration subsystem.** The Orchestration and Configuration subsystem is part of the Pledger Core (Figure 1), as identified in Deliverable "D2.3 Pledger Overall Architecture" [1], which acts as the roadmap for all development and integration tasks of the project.

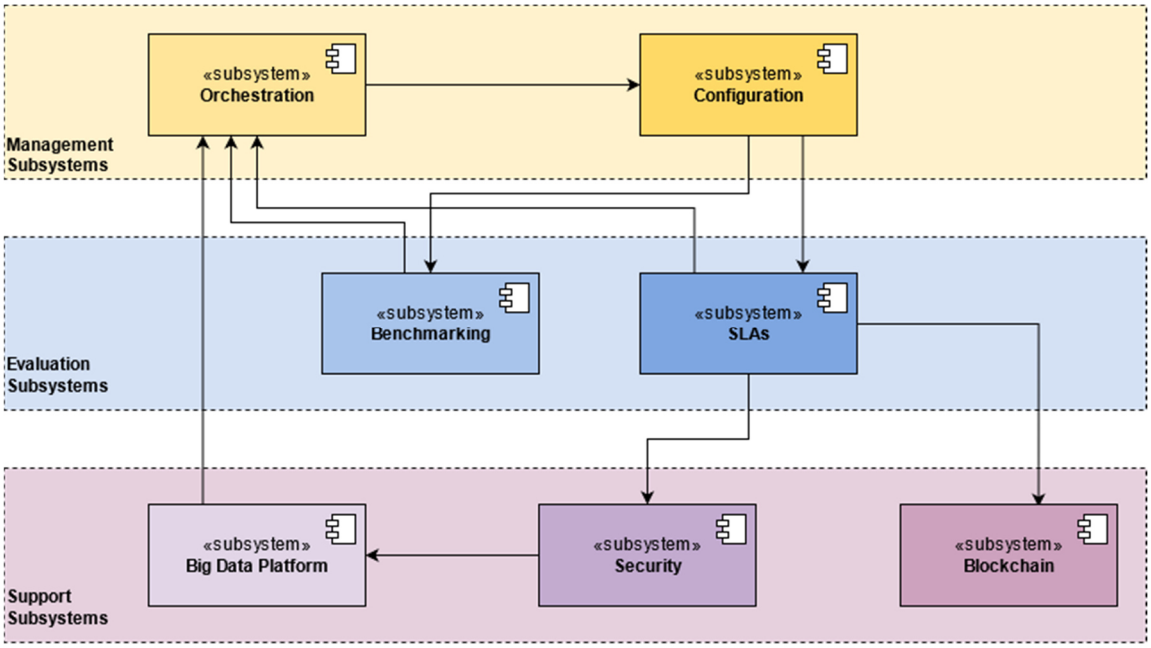



Figure 1: The Pledger Core Subsystems

As one of the **Management Subsystems** of Pledger, the Orchestration subsystem is focused in the management of Infrastructure as a Service (IaaS) and Software as a Service (SaaS), and acts as the link between the Decision Support System (DSS) and the underlying infrastructure, providing an abstraction layer on top of infrastructure management tools of choice to the DSS. This way, it allows to decouple the DSS from the technical implementation details and facilitates interoperability with diverse technical solution and makes it possible to transparently update and change low level infrastructure management tools. The DSS ("Recommender" in Figure 2) will implement the intelligence of decision-making and the Orchestrator will be the executor arm of DSS decisions (implementing on demand the necessary actions).

This subsystem manages orchestration of containerized applications (i.e. relying on Kubernetes implementation) to handle its management. The actions foreseen for this subsystem are related to the deployment of applications, infrastructure scale up/down and migration as well as operations in relation to the operational life-cycle of applications (start, stop, update and get current status of the application).

In addition, this subsystem also considers the management of complete clusters at different infrastructure levels (cloud, edge, on premise) as well as the deployment of cluster orchestration software (for single-node cluster) in the edge.

The Orchestration subsystem is in charge of selecting the best nodes of a cluster (best effort) to run an application based on the SaaS provider's preferences or the profile of the application (e.g. app requires Graphics Processing Unit (GPU) to run) or the recommendations added by DSS at runtime. The Orchestrator also receives input from a Monitoring Engine to collect different metrics of the

infrastructure in a time series database. With these functionalities we can settle the base for a QoS control system.

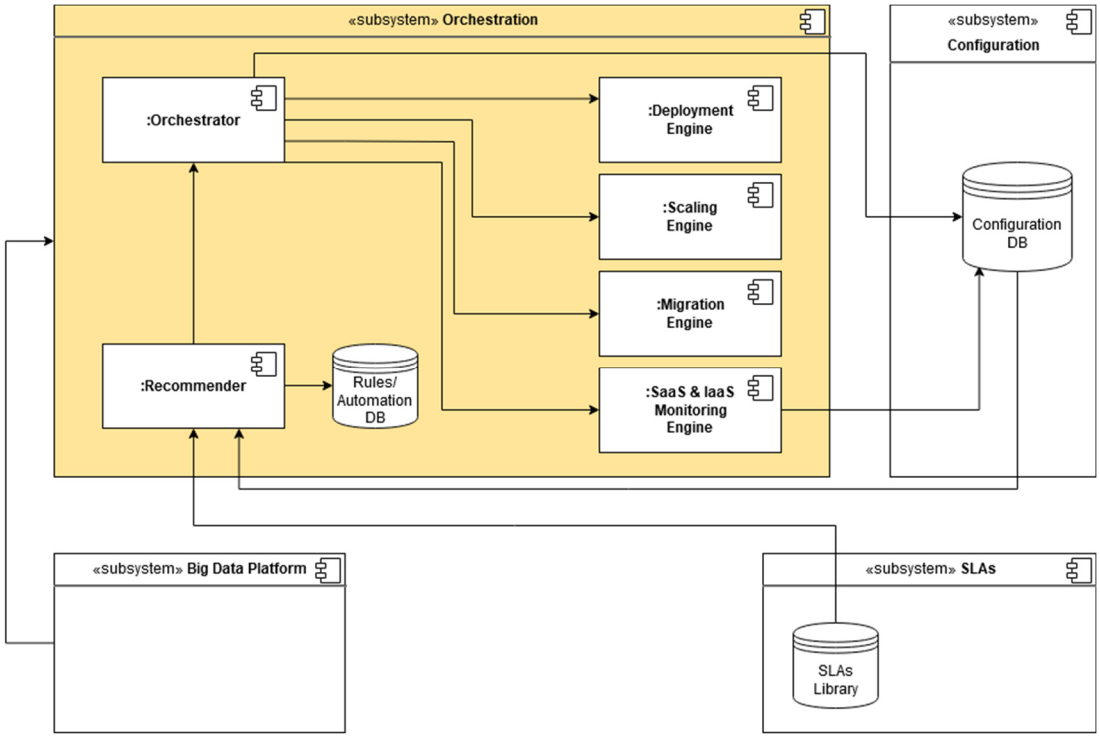



Figure 2: Orchestration subsystem Component Diagram and relation to other subsystems

More specifically, the subsystem comprises the following main Functional Components (FC):

Table 1: Orchestration subsystem Components

| ID | Component | Functionality |
|---|---|---|
| FC.2.1 | **Orchestrator** | This component is in charge of managing the information related to runtime environment specifications (profiles), such resources preferred/ required (like RAM, vCPU, etc.) for the applications, and of executing basic operations like start, stop, update or get status of applications in a cluster or infrastructure. |
| FC.2.2 | **Deployment Engine** | This component is in charge of deploying and removing the applications to the physical infrastructure, either in a swarm or different type of infrastructure. The "deployment" could be a first deployment, a redeployment due to runtime specification changes or an update of the version of the application. |
| FC.2.3 | **Scaling Engine** | This component implements a horizontal scaling (scale out/ in) of the application creating replicas of the same application (or subcomponents of the application in case of microservices, Y axis in scale cube[1]) or decreasing replicas. It also implements a vertical scaling (scale up) by calling the Migration Engine component to move the app to a node of the cluster with more resources (CPU, RAM, etc.) or with less resources (scale down). |

[1] https://uniknow.github.io/AgileDev/site/0.1.10-SNAPSHOT/scale-cube.html

| ID | Component | Functionality |
|---|---|---|
| FC.2.4 | **Migration Engine** | This component implements the migration of applications already deployed, by moving the app to different infrastructure with more resources (e.g. edge to cloud, other node type with more resources like RAM, CPU) or with less resource (e.g. cloud to edge). The reasoning for this migration covers different needs like colocation for decreasing latency, lift & shift for more computer capacity, etc. |
| FC.2.5 | **Monitoring Engine** | This component collects metrics of cluster infrastructure and stores these values in a time series database. It is also the query interface for metric values from other subsystems or components. |
| FC.2.6 | **Recommender** | The Recommender component is responsible for the provisioning of suggestions to the SaaS providers related to the app orchestration. In particular, the suggestions focus on the most efficient allocation within the available infrastructures, taking into account the infrastructure and app properties from the configuration subsystem, the infrastructure and app status from the IaaS Monitoring component and the Service Level Agreement (SLA) Manager, and the user preferences stored again in the configuration subsystem. |
| FC.2.7 | **Rules/ Automation DB** | The Rules/ Automation DB component stores the Recommender suggestions and includes rules to decline the decision-making process (with regard to the user preferences) and possibly provide (small) automations to allow edge nodes act autonomously, depending on the resources available, whenever a disconnection from the infrastructure occurs. |

**As far as the Orchestration Subsystem is concerned this deliverable will be focusing on FC.2.1-FC.2.5**, while FC.2.6-FC.2.7 will be presented as part of Task "T4.3 – Decision Support mechanisms for Edge/Cloud computation moving".

The **Configuration Subsystem** as part of the **Management Subsystems** is responsible to store the infrastructure and application configuration that is managed either manually by the IaaS/ SaaS providers or by tools that support automatic discovery features (e.g. the App Profiler).The main information stored is: users' configuration, infrastructure configuration, app configuration, multitenancy and authorizations for users to access specific infrastructures within resource limitations (e.g. resource quotas).

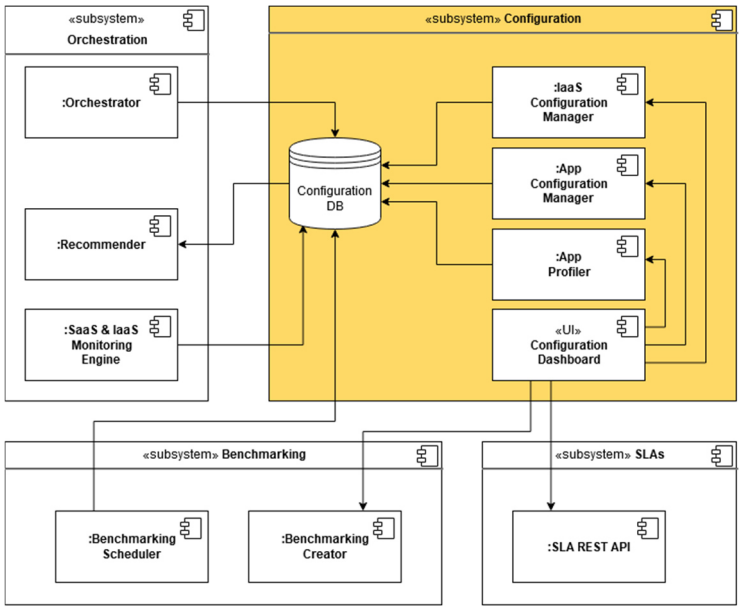



Figure 3: Configuration subsystem Component Diagram

More specifically, the subsystem comprises the following main components:

Table 2: Configuration subsystem Components

| ID | Component | Functionality |
|---|---|---|
| FC.1.1 | **IaaS Configuration Manager** | The IaaS Configuration Manager is responsible for the management of infrastructure configuration, spanning from the credentials to the main topology and properties of the infrastructure (such as the servers URI, the master/worker properties for Kubernetes, GPU type, CPU model, etc.) that could impact scheduling decisions, as well as resource limits configured from the IaaS providers. |
| FC.1.2 | **App Configuration Manager** | The App Configuration Manager is responsible for the management of the app configuration, which includes generic information used to match SaaS providers' preferences expressed in their profiles, along with those specific related to QoS and SLAs. QoS keys are listed for each application and SLA values/ thresholds are stored to allow the SLA Manager to check their violations and prioritize them. |
| FC.1.3 | **App Profiler** | The App Profiler component is responsible for the manual or automatic profiling of applications in order to provide additional information about apps in the form of key-value set that can be used during the decision-taking phase before scheduling. The SaaS provider can add application-specific properties (e.g. GPU intensive, CPU intensive). |
| FC.1.4 | **Configuration DB** | The Configuration DB is responsible for storing the aforementioned configuration information and sharing it through specific API to the other Pledger subsystems, such as the Recommender in the Orchestration subsystem, the Benchmarking and the SLA creators' components. |

| ID | Component | Functionality |
|---|---|---|
| FC.1.5 | **Configuration Dashboard** | The Configuration Dashboard is the UI provided to IaaS and SaaS users to allow the proper configuration of the aforementioned data and also includes reports to allow the SaaS users to have a detailed view of the infrastructures and apps status and the recommendations for the app orchestration. |

**As far as the Configuration Subsystem is concerned this deliverable will be focusing on FC.1.3 App Profiler**, the other functional components of this Subsystem are presented as part of Task "T3.1 – Performance Measurements and Classification".

# 3 Technical description

## 3.1 Baseline technologies and dependencies

### 3.1.1 Edge to Cloud Orchestrator component (E2CO)

The following baseline technologies are used within this component:

Table 3: Baseline technologies used by E2CO tool (Orchestrator subsystem)

| Name | Description | Version |
|---|---|---|
| **LifeCycle Manager** | An Edge-To-Cloud Resource Orchestration tool that provides life-cycle management for composed services described as collections of containers and relying in container orchestration tools such as Docker Swarm and Kubernetes. It is being developed by ATOS under an open source license (Apache License 2.0), and it has been used in another H2020 projects. | |
| **Prometheus [2]** | Prometheus is an open source monitoring system, written in Golang, and released with Apache License 2.0. This tool can be integrated with container orchestrators like OpenShift and Kubernetes, and it can get metrics from the infrastructures and applications. | Prometheus Operator Stack |
| **Kubernetes [3]** | Kubernetes is an open source container orchestration system for automating computer application deployment, scaling, and management, licensed under Apache License 2.0. Originally designed by Google, Kubernetes is now maintained by the CNCF (Cloud Native Computing Foundation). | Vanilla 1.19+ |
| **Grafana [4]** | Grafana is a data visualization and monitoring tool used to show data from external sources, like Prometheus. | |
| **Docker [5]** | Container engine for containerized apps running in the edge | 1.18+ |
| **Kafka [6]** | Apache Kafka is an open-source distributed event streaming platform used for high-performance data pipelines and streaming analytics. It is part of the Streamhandler platform that handles data management in Pledger, and as such, interfaces with multiple components in the Pledger Core and Use Case deployments. Kafka is selected for its high throughput and scalability. | |
| **MongoDB [7]** | MongoDB is an open source NoSQL database that uses JSON-like documents with optional schemas. | |

### 3.1.2 App Profiler component

The following baseline technologies are used within this component:

Table 4: Baseline technologies used by App Profiler

| Name | Description | Version |
|---|---|---|
| **Node-RED** | Node-RED is a flow-based development tool for visual programming developed originally by IBM (Apache License 2.0) for wiring together hardware devices, APIs and online services as part of the Internet of Things. Node-RED provides a web browser-based flow editor, which can be used to create JavaScript functions. | 1.3.4 |

| Name | Description | Version |
|---|---|---|
| **Docker** | Docker is a set of platforms as a service product that use OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries and configuration files; they can communicate with each other through well-defined channels. | 20.10.6 |
| **NumPy** | NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. | 1.20.1 |
| **Pandas** | Pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language. | 1.2.4 |
| **scikit-learn** | Scikit-learn is a free software machine learning library for the Python programming language. It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, k-means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy. | 0.24.2 |

## 3.2 Components Architecture

### 3.2.1 Edge to Cloud Orchestrator component (E2CO)

The functional components Orchestrator, Deployment Engine, Scaling Engine and Migration Engine are implemented in a single application (main module) coded in Golang language and containerized in one container (e2co-app). The datastores are implemented in plain text database file in the main module. The UI for this iteration is implemented in a single page application (SPA) with HTML and React JavaScript framework and containerized in another container (e2co-ui). The UI calls the REST API of the E2CO main module.

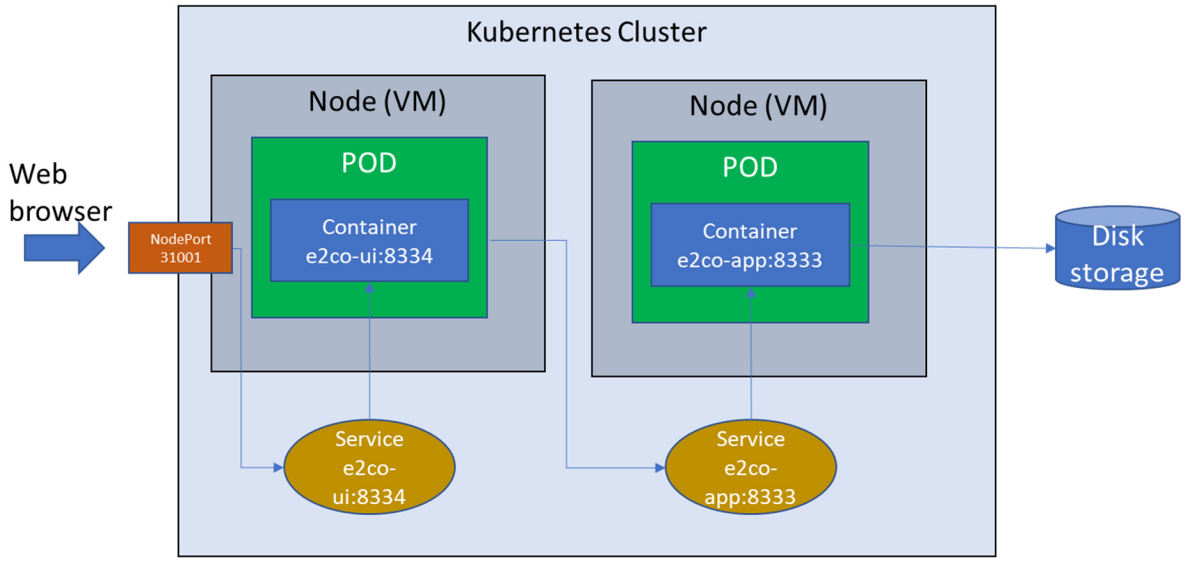



Figure 4: E2CO component of the Orchestrator subsystem deployment diagram in a Kubernetes cluster

Every container is deployed in a separated Pod and each Pod could be in the same node or in different nodes in the cluster.

### 3.2.2 App Profiler component Architecture

App profiler is the component responsible for profiling applications that are deployed and used as a Docker container. More specifically, AppProfiler extracts the resource usage measurements of a containerized instance and maps it to the resource usage of a known benchmark. Since benchmarks are created by design to be easily installed and performed, they can be used to assess the computational capabilities of specific hardware. This is not the case for general applications, thus mapping the applications to specific benchmarks through the App Profiler, can help the establishment of basic knowledge on how a specific application can perform on different systems based on known benchmarks.

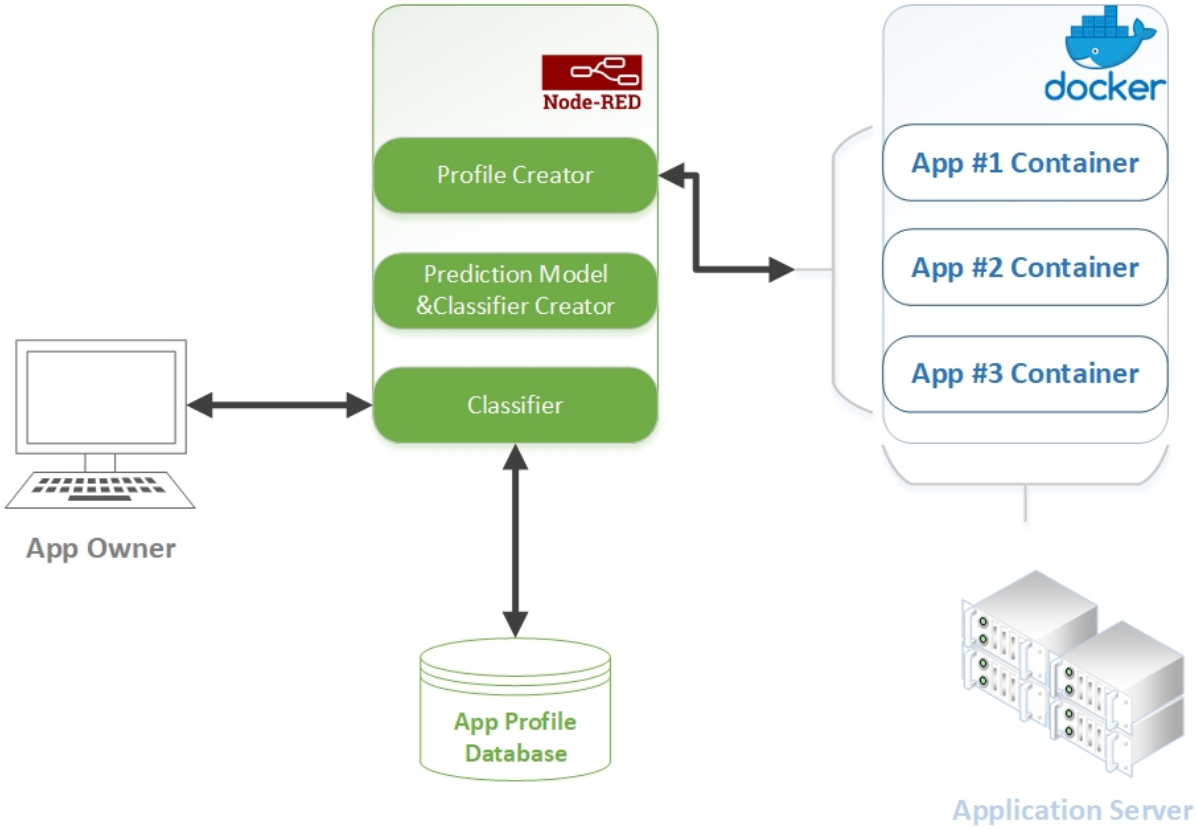



Figure 5: App Profiler high level architecture

As depicted in the image below Application profiler is Node-RED based tool that has three main subcomponents:

- ***Profile Creator***: Is responsible for the connection to the Docker API in order to extract container statistics based on the id of the container provided. If the profile is for a container running a benchmark, then the profile will be stored in the database in order to be used in the **Classifier Creator**. If the profile is for an application running in the container the profile will be used to classify the application with the **Classifier**.
- ***Prediction Model & Classifier Creator***: Is the component responsible for training the machine learning algorithm and providing the model which the **Classifier** will use to provide the mapping (application profile to benchmark).
- ***Classifier***: This component uses as input the profile created by **Profile Creator** and returns a benchmark that best represents this profile.

## 3.3 Interfaces provided

### 3.3.1 Edge to Cloud Orchestrator component (E2CO)

This section describes the REST interfaces provided by E2CO component. Interfaces provided by the other tools or platforms like, for example, Kubernetes or Prometheus, can be found in their respective web sites and documentations.

The REST API services exposed by the E2CO component are the following:

- **/api/v1/ime** Services used to manage the clusters / orchestrators managed by E2CO
- **/api/v1/apps** Applications managed by E2CO. These applications are deployed and launched in the clusters / orchestrators managed by the application.

**E2CO apps operations**

Table 5 E2CO apps operations with REST API

| Operation | Method | URI | Description |
|---|---|---|---|
| LIST | GET | /apps | Returns a list of all the current E2CO apps (from all infrastructures / clusters) |
| READ | GET | /apps/{id} | Returns an E2CO application. |
| READ | GET | /apps/{id}/all | Gets a E2CO application, including deployment info. |
| DELETE | DELETE | /apps/{id} | Delete an E2CO application |
| CREATE | POST | /apps | Create an E2CO application (a JSON file with the application details must be provided) and deploy it in the selected cluster. |
| UPDATE | PUT | /apps/{id} | Migrates / scales / updates an E2CO application |

**E2CO infrastructure operations (ime)**

Table 6: E2CO infrastructure operations with REST API

| Operation | Method | URI | Description |
|---|---|---|---|
| LIST | GET | /ime/e2co | Returns all clusters info. |
| LIST | GET | /ime/e2co-apps | Returns all clusters info with grouped applications. |
| READ | GET | /ime/e2co/{id} | Returns an E2CO cluster info. |
| DELETE | DELETE | /ime/e2co/{id} | Delete an E2CO cluster info. |
| CREATE | POST | /ime/e2co/{id} | Create an E2CO cluster info (a JSON file with the cluster details must be provided). |
| UPDATE | PUT | /ime/e2co/{id} | Update an E2CO cluster info. |

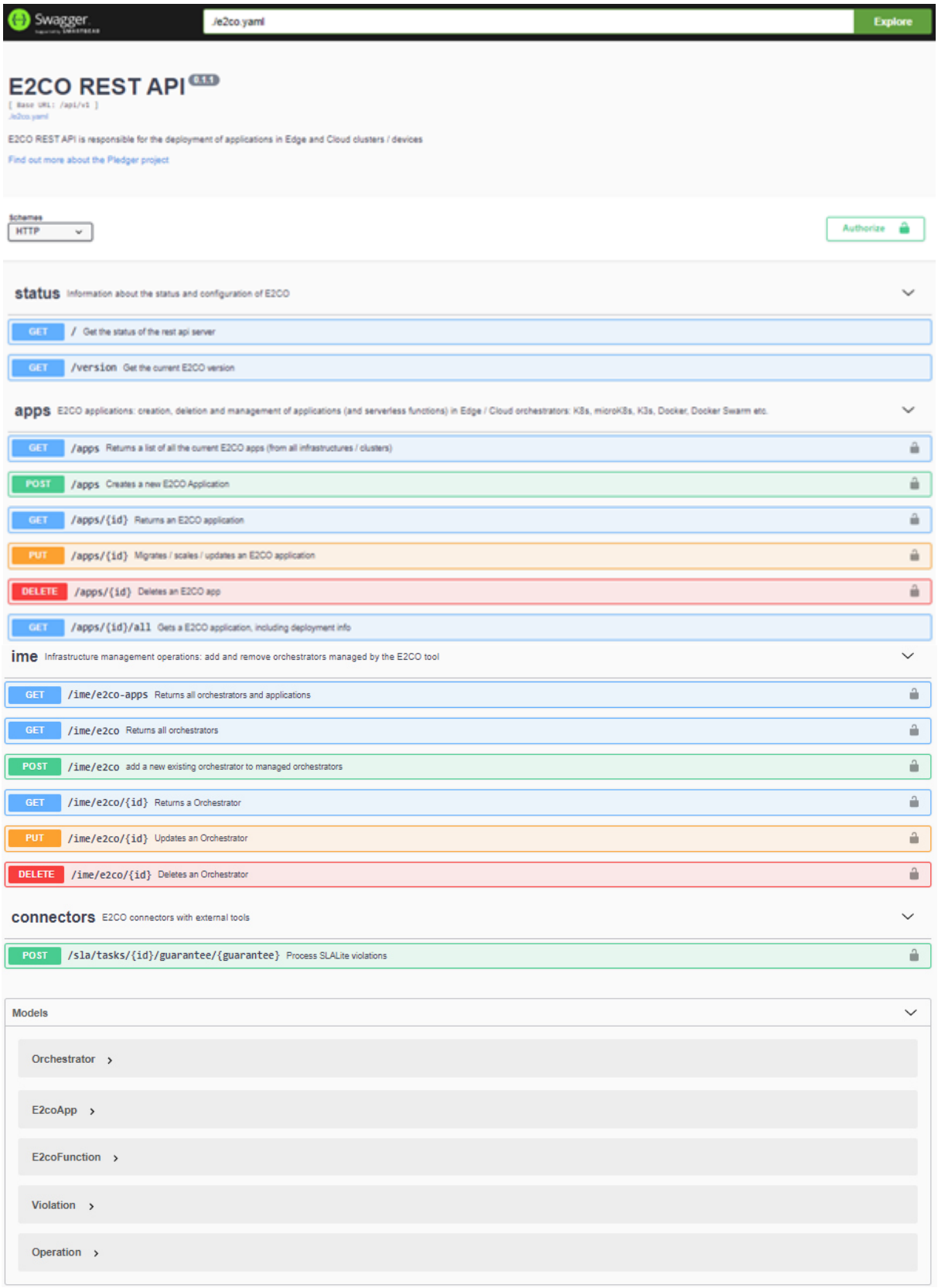



Figure 6: The E2CO component swagger interface for REST API

### 3.3.2 App Profiler API

The external API to automatically run the profiles and classifications is in a preliminary stage, the full integration of the REST API calls will be delivered in the second integration of the tool. The REST API services exposed so far by the App Profiler component are the following:

Table 7: App Profiler exposed REST API methods

| Operation | Method | URI | Description |
|---|---|---|---|
| Profile | POST | AppProf/profile | This method takes as input the specific target container and the exposed Docker API and returns the Profile |
| Classify | POST | AppProf /classify | This method takes as input the profile vectors and returns a known benchmark that produces the same hardware load footprint. |

## 3.4 Data models

### 3.4.1 Edge to Cloud Orchestrator component (E2CO)

The main entities are the cluster infrastructure information and the application manifest. The first one describes the kind of cluster technology (Kubernetes, docker, VM, etc.) and the parameters to connect to the control plane of the cluster (endpoint URL, credentials, tokens, etc.).

The application entity describes what resources are needed to run the application like the destination cluster infrastructure, container image, number of instances, storage volumes, environment variables, communication protocols and ports. There is also a section to indicate the QoS expected for the execution of this application. In this section the consumer can define metrics and thresholds to fulfill (constraints) by the infrastructure, importance intervals of the violations and some rule-based actions to run in case of violation of any constraint.

The cluster description is defined in JSON format with the following schema:

```
{
  "id": "id of the cluster infrastructure",
  "name": "name of the cluster",
  "description": "description of the cluster infrastructure",
  "location": "some geospacial reference like region, country, city, zone, etc.",
  "type": "type of the cluster infrastructure: k8s (Kubernetes), docker, etc.",
  "so": "operating system in case of an edge node",
  "defaultNamespace": "namespace of a Kubernetes based cluster",
  "restAPIEndPoint": "url of the cluster API server",
  "ip": "IP in case of an edge node to create remote shell connection",
  "connectionToken": "credential token to the cluster API server (service account)",
  "slaLiteEndPoint": "url of the SLA component REST API",
  "prometheusPushgatewayEndPoint": "url of a pushgateway program to get edge metrics",
  "prometheusEndPoint": "url of a Prometheus instance (metric collector) for querying metrics"
}
```

The application description is defined in JSON format with the following schema:

```
{
  "name": "name of the application",
  "namespace": "namespace in case of Kubernetes cluster",
  "idE2cOrchestrator": "id of the target cluster to deploy the app",
  "qos": [{
        "name": "guarantee id (metric name)",
        "constraint": "[metric] comparison_op threshold",
        "importance": [{
                “name”: “name of severity category”,
                “Constraint”: “comparison_op threshold”
        }],
        “penalties”: [{
                “type”: “type of penalty”,
                “value”: “value of penalty”,
                “unit”: “unit of penalty”
        }],
        “actions”: [{
                “type”: “type of action, e.g. SCALE OUT, MIGRATE”,
                “value”: “value for the action, e.g. number of replicas to
scale”,
                “unit”: “unit of the value”
        }]
   }]
  "replicas": number of instances of the app to create at deployment time,
  “maxReplicas”: maximum number of instances of the app in the cluster,
  "containers": [ {
      "name": "name of the container for the app",
      "image": "image of the container as defined in the image repository",
      "ports": [ {
          "containerPort": port of the containiner,
          "hostPort": port of the node,
          "protocol": "TCP/IP protocol (tcp or udp)"
       } ],
      "volumes": [ {
          "name": "name of the volume",
          "mounthPath": "full path of the mounted volume"
       } ],
      "environment": [ {
          "name": "environment variable name",
          "value": "variable value”
      } ]
   } ]
}
```

This is an example of a cluster definition:

```
{
  "id": "cluster1",
  "name": "cluster1",
  "description": "Kubernetes cluster",
  "location": "Madrid",
  "type": "k8s",
  "so": "ubuntu18",
  "defaultNamespace": "core",
  "restAPIEndPoint": "https://192.168.1.131:16443/",
  "ip": "192.168.1.131",
  "connectionToken": "eyJhbG....wm9AKA",
  "slaLiteEndPoint": "http://localhost:8090",
  "prometheusPushgatewayEndPoint": "",
  "prometheusEndPoint": "https://192.168.1.131:9090/"
}
```

This is an example of an app definition:

```
{
  "name": "nginx-app",
  "namespace": "development",
  "idE2cOrchestrator": "maincluster",
  "qos": [{
        "name": "throughput",
        "constraint": "throughput < 70000",
        "importance": [
            {
              "Name": "Mild",
              "Constraint": " > 70000"
            },
            {
              "Name": "Serious",
              "Constraint": " > 700000"
            }],
        "penalties": [{
              "type": "",
              "value": "",
              "unit": ""
          }],
        "actions": [{
              "type": "",
```

```
                "value": "",
                "unit": ""
            }]
        },
        {
          "name": "responseTime",
          "constraint": "responseTime < 100"
        }],
    "replicas": 2,
    "containers": [    {
        "name": "nginx",
        "image": "nginx",
        "ports": [        {
            "containerPort": 80,
            "hostPort": 80,
            "protocol": "tcp"
          }      ],
        "volumes": [        {
            "name": "name",
            "mounthPath": "mounthPath"
          }      ],
        "environment": [        {
            "name": "TEST_VALUE",
            "value": "1.2.3"
          }      ]
      }  ]
}
```

### 3.4.2 App Profiler component Data Model

As mentioned in Section 3.2.2, App Profiler extracts usage metrics from Docker API in order to create the profile. Docker Container usage metrics Schema that is used by the App Profiler can be found: https://docs.docker.com/engine/api/v1.41/#operation/ContainerExport .

This is an example of a typical JSON produced by the Docker API and consumed by the App Profiler:

```
{
    "read": "2021-05-17T14:29:57.429750966Z",
    "preread": "2021-05-17T14:29:56.425005308Z",
    "pids_stats": {
        "current": 15
    },
    "blkio_stats": {
```

```
        "io_service_bytes_recursive": [],
        "io_queue_recursive": [],
        "io_service_time_recursive": [],
        "io_wait_time_recursive": [],
        "io_merged_recursive": [],
        "io_time_recursive": [],
        "sectors_recursive": []
    },
    "num_procs": 0,
    "storage_stats": {},
    "cpu_stats": {
        "cpu_usage": {
            "total_usage": 5094039708,
            "percpu_usage": [],
            "usage_in_kernelmode": 2750000000,
            "usage_in_usermode": 8120000000
        },
        "system_cpu_usage": 121604730000000,
        "online_cpus": 8,
        "throttling_data": {
            "periods": 0,
            "throttled_periods": 0,
            "throttled_time": 0
        }
    },
    "precpu_stats": {
        "cpu_usage": {
            "total_usage": 5094039708,
            "percpu_usage": [],
            "usage_in_kernelmode": 2750000000,
            "usage_in_usermode": 8120000000
        },
        "system_cpu_usage": 121596700000000,
        "online_cpus": 8,
        "throttling_data": {
            "periods": 0,
            "throttled_periods": 0,
            "throttled_time": 0
        }
    },
    "memory_stats": {
        "usage": 48508928,
```

```
        "max_usage": 48640000,
        "stats": {},
        "limit": 33626771456
    },
    "name": "/portainer",
    "id": "e687e706c2e8a42f5945136027eb962f2c31350aedecd4138a6d18dec92c176b",
    "networks": {
        "eth0": {}
    }
}
```

# 4 Installation and usage guides

## 4.1 Requirements

### 4.1.1 Edge to Cloud Orchestrator component (E2CO)

The following software components and credentials must be in place to install the E2CO component:

- Kubernetes cluster ver. 1.19+ (this cluster will be the broker cluster)
- Credentials of K8S cluster with permission to deploy applications.
- Persistence storage for plain text database file in K8S cluster.
- Kafka client certificate KeyStore and passwords as secrets object in K8S cluster.
- Kafka endpoints as IP address of Kafka servers.
- The container images of the subsystem in a container image repository. It could be the image repository of the project (JFROG) in the following url: 116.203.2.204:443/plgregistry
- A monitoring system or metric collector (Prometheus) in the K8S broker cluster.

### 4.1.2 App Profiler component

The required software and configuration in order to run the App Profiler are listed below:

- Docker Engine ver. 18 or higher
  - Exposed Docker API in specific IP and Port
- Node-RED ver. 1.2 or higher
  - Additional Libraries for Python Machine learning module:
    - Numpy
    - Pandas
    - SciKit-Learn
  - Additional Node-RED flows:
    - node-red-contrib-machine-learning
    - node-red-dockerode

## 4.2 Installation

### 4.2.1 Edge to Cloud Orchestrator component (E2CO)

The first step is to download the source code from the Pledger Gitlab repository:

```
git clone https://gitlab.com/pledger/edge-to-cloud-orchestrator.git
```

to the next step is to build the docker image of the UI and the core tier:

```
cd edge-to-cloud-orchestrator
docker build --rm -t e2co .
cd ./UI
docker build --rm -t e2co-ui .
cd ..
```

To install the E2CO component, the command line tool **kubectl** from K8S is required. It is also possible to install this subsystem in the testbed of the project using the project's CI/CD tools (Jenkins pipelines).

To customize the installation to a specific environment, the "kustomizer" tool may be used along with the corresponding yaml files. Some examples of this customization can be seen in the subdirectory "overlays" of the project.

For example, the command to install this subsystem in the testbed is the following:

```
kubectl apply -k overlays/testbed
```

For more details of manifest files (yaml) can be found in:

- https://gitlab.com/pledger/edge-to-cloud-orchestrator/-/tree/master/base and
- https://gitlab.com/pledger/edge-to-cloud-orchestrator/-/tree/master/overlays/testbed.

There are several environment variables to configure before installing the subsystem. These variables are declared in the manifest files of the subsystem (yaml files)(Table 8).

Table 8: E2CO tool environment variables for configuration

| Name | Description | Default | Mandatory | Example |
|---|---|---|---|---|
| E2CO_REPOSITORY | DB database file with full path | :memory: | No | /opt/e2co/data/e2co.db |
| E2CO_SLA_ENDPOINT | URL of the SLA subsystem REST API | | Yes | http://slalite-app:8090 |
| E2CO_WARMING_TIME | Time (seconds) between operations on the same app or cluster | 180 | No | 180 |
| E2CO_KAFKA_CLIENT | Kafka client implementation | | No | kafka2 |
| E2CO_KAFKA_ENDPOINT | Kafka brokers addresses | | No | - |
| E2CO_KAFKA_NOTIFIER_TOPIC | Kafka topic for consuming sla violation alerts | | No | sla_violation |
| E2CO_KAFKA_AGREEMENT_TOPIC | Kafka topic for producing agreements info | | No | sla_agreement |
| E2CO_KAFKA_KEYSTORE_PASSWORD | Kafka KeyStore password for client certificates | | No | password |
| E2CO_KAFKA_KEY_PASSWORD | Kafka private key password for client certificates | | No | password |
| E2CO_KAFKA_TRUSTSTORE_PASSWORD | Kafka truststore password for server public certificates (CA) | | No | password |
| E2CO_KAFKA_KEYSTORE_LOCATION | Kafka KeyStore location in container filesystem | | No | /var/kafka_keystore/kafka.client.keystore.p12 |

| Name | Description | Default | Mandatory | Example |
|---|---|---|---|---|
| E2CO_KAFKA_TRUSTSTORE_LOCATION | Kafka truststore location in container filesystem | | No | /var/kafka_truststore/kafka.client.truststore.pem |

### 4.2.2 App Profiler component

Installation guides for the Docker[2] and Node-RED[3] can be found in their official websites.

After setting up of Docker and Node-RED through the Node-RED UI(default http://localhost:1880) import the flows and python libraries mentioned in the section 4.1.2.

Finally install the App Profiler by importing the Flows from the Pledger Code repository: https://gitlab.com/pledger/app-profiler

## 4.3 Usage

### 4.3.1 Edge to Cloud Orchestrator component (E2CO)

We can manage the subsystem via REST API calls or by means of the UI component (web console). For the list of API calls you can find the swagger documentation at an annex of this document and in the following URL of the UI component: https://cluster-external-ip:e2co-ui-svc-port/#/ (e.g. http://localhost:31001/#/).

The main operations for this subsystem are the lifecycle management of applications and clusters (CRUD operations). Every cluster info represents an infrastructure provided by the provider to deploy an application provided by a consumer of Pledger system.

We can also invoke the REST API via swagger HTML interface published in this URL: https://cluster-external-ip:e2co-ui-svc-port/swaggerui/#/ (e.g. http://localhost:31000/swaggerui/#/). We can see an example in the demo section of this document.

### 4.3.2 App Profiler component

App Profiler is operated through the UI of Node-RED which by default is accessed by a web browser using this URL: http://localhost:1880.

## 4.4 Licenses

**E2CO** component has an Apache license type version 2.

**App Profiler** component has an Apache license type version 2.

## 4.5 Source code repository

The source code repository for **E2CO** is located in GitLab.com in private repositories.

The URL of the repository of this subsystem is this: https://gitlab.com/pledger/edge-to-cloud-orchestrator

The source code repository for **App Profiler** is located in GitLab.com in private repositories.

The URL of the repository of this subsystem is this: https://gitlab.com/pledger/app-profiler

[2] https://docs.docker.com/engine/install/ubuntu/
[3] https://nodered.org/docs/getting-started/local

# 5 Demonstration

As far as the demonstrator is concerned, the Orchestrator component is integrated with the QoS and SLA Assessment tools. In this sense, the scenarios for both deliverables (D3.2, D3.3[8]) present the same demonstrator highlighting the involvement of both Orchestration and QoS assessment tools, in order to demonstrate unified scenarios that present the individual functionalities, but most importantly a more complete and integrated system.

## 5.1 Scenario description

In order to identify which of the general tools are involved in every step of the scripts the annotation **Orchestrator** and **SLA Framework** are used.

**Scenario 1:**

**Objectives**: Demonstrate configuration of infrastructure and applications (resources and QoS). Deployment of applications in the infrastructure. Check QoS and take remediation action at runtime.

**Script**:

1. Add a cluster infra configuration (**Orchestrator**)
2. Add an app deployment config with QoS params and actions (**Orchestrator& SLA Framework**)
3. Deploy the app with the Orchestrator in the new cluster (**Orchestrator**)
4. Check QoS threshold (metric) from Monitoring with SLA tool (**SLA Framework**)
5. Send a violation alert from SLA to message middleware (**SLA Framework**)
6. Orchestrator reads violation alert from middleware and executes an action (*scale out*) (**Orchestrator**)

**Scenario 2:**

**Objectives**: Demonstrate placement of applications between Edge and Cloud as a runtime adaptation.

**Script**:

1. Add cluster and edge infra configuration (**Orchestrator**)
2. Add an app deployment config with QoS params and actions (**Orchestrator& SLA Framework**)
3. Deploy the app with the Orchestrator in the target location (**Orchestrator**)
4. Check QoS threshold (metric) from Monitoring with SLA tool (**SLA Framework**)
5. Send a violation alert from SLA to message middleware (**SLA Framework**)
6. Orchestrator read violation alert from middleware and execute an action (*placement*) (**Orchestrator**)

## 5.2 Validation and Verification

This first iteration of Orchestration subsystem addresses the Minimum Viable Product (MVP) requirements for this first iteration of the tool. The System User Cases (SUC) that were implemented in this iteration were the following:

- SUC.02: Deploy App components.
- SUC.03: View available IaaS resources.
- SUC.05: Select fitting IaaS resources.
- SUC.06: Get App & App instances status info.
- SUC.07: Control App & App instances.
- SUC.08: Get notified about unwanted events.

- SUC.09: Handle unwanted events.
- SUC.11 (partially): Choose/ perform recovery action(s), in this integration two recovery actions are performed based on the alerts, SaaS providers cannot choose a strategy yet.
- SUC.13: Start/ pause/ stop App instances.
- SUC.15 (partially): Choose to scale up/ down (components of) App, same us SUC.11.
- SUC.19: IaaS monitoring.

These SUC were defined in D2.3 PLEDGER Overall Architecture

For the verification of this first iteration we present a demonstration with two different and integrated scenarios to show the implementation of the MVP's SUCs (at least the first version of them) and with the evidences in video recording format.

For the validation of the European Commission (EC) of the work done we made this document to help in the process of validating of this first iteration of the Orchestration tools describing the work done, conclusion and next steps.

## 5.3 Demo

### 5.3.1 Scenario 1: SCALE OUT

**1. Add a cluster infra configuration**

The description file for add a new infra cluster of Kubernetes for this demo scenario is the following:

```
{
    "id": "mycluster",
    "description": "Pledger Testbed",
    "type": "k8s",
    "defaultNamespace": "core",
    "restAPIEndPoint": "https://kubernetes.default:443",
    "connectionToken":  "eyJh… sYWA",
    "slaLiteEndPoint": "http://slalite-app:9080",
    "prometheusEndPoint": ""
  }
```

We can add this new infra configuration via API REST call using the swagger interface of the orchestrator as show below (url http://<Pledger_cluster_IP>:31000/swaggerui/#/ime/addOrchestrator)

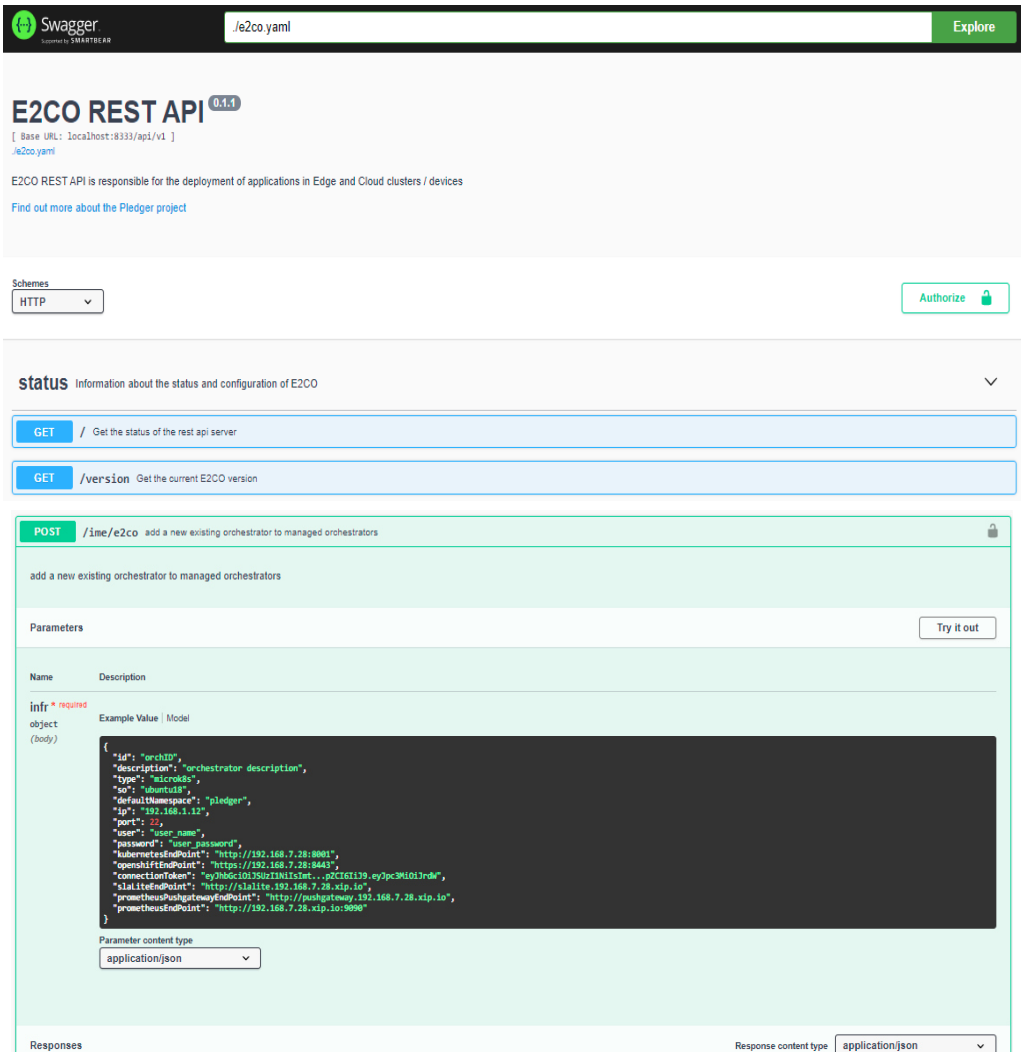



Figure 7: PLEDGER Orchestrator swagger interface for REST API

We can also use the Orchestrator UI or another tool that can make API REST calls.

## 2. Add an app deployment config with QoS params and actions

For deploy a new application we use a description file like the follow:

```
{
  "name": "myapp",
  "namespace": "core",
  "idE2cOrchestrator": "mycluster",
    "qos": [{
        "name": "scrape_duration_seconds",
        "constraint": "scrape_duration_seconds < 0.4",
        "actions": [{
           "type": "SCALE OUT",
           "value": "1",
```

```
            "unit": ""  }]
        }],
    "replicas": 1,
    "maxReplicas": 3,
    "containers": [    {
        "name": "nginx",
        "image": "nginx",
        "ports": [        {
            "containerPort": 80,
            "hostPort": 80,
            "protocol": "TCP"
          }      ]
      } ]
}
```

Similarly, we can create this app deployment using Orchestrator REST API with Orchestrator UI, swagger HTML interface or any other external tool.

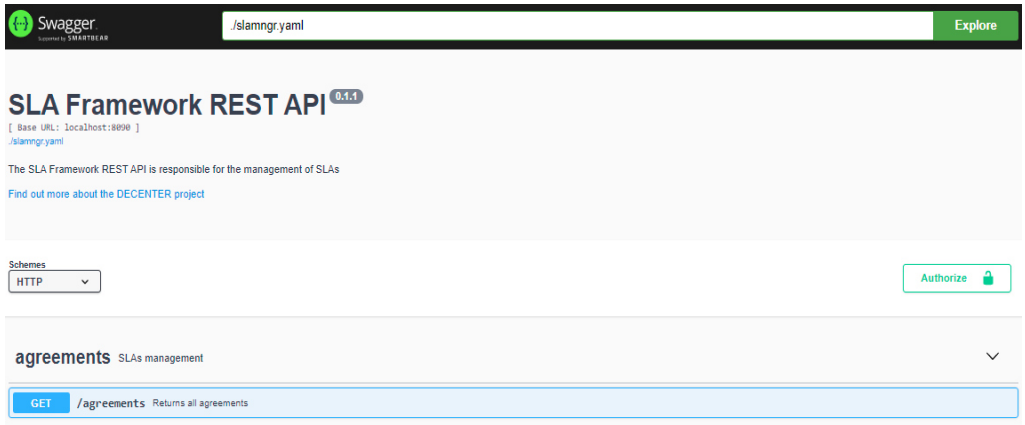
The highlight point of this JSON file for SLA subsystem is the part of QoS Here the consumer defines the requirements to fulfill at runtime for the infrastructure expressed by means of metrics and thresholds.

For this demo we add a rule-based action or actions to execute when the agreement is not met (scale out) to allow the system to recover the QoS expected.

3. **Deploy the app with the Orchestrator in the new cluster**

The Orchestrator subsystem deploys the app in the Kubernetes cluster and send an API rest call to the SLA subsystem to create the SLA agreement of the new app with the QoS defined by the consumer.

We can reproduce this call using the swagger interface of the SLA like this example or with other external tools (URL http://<Pledger_cluster_IP>:32000/swaggerui/):



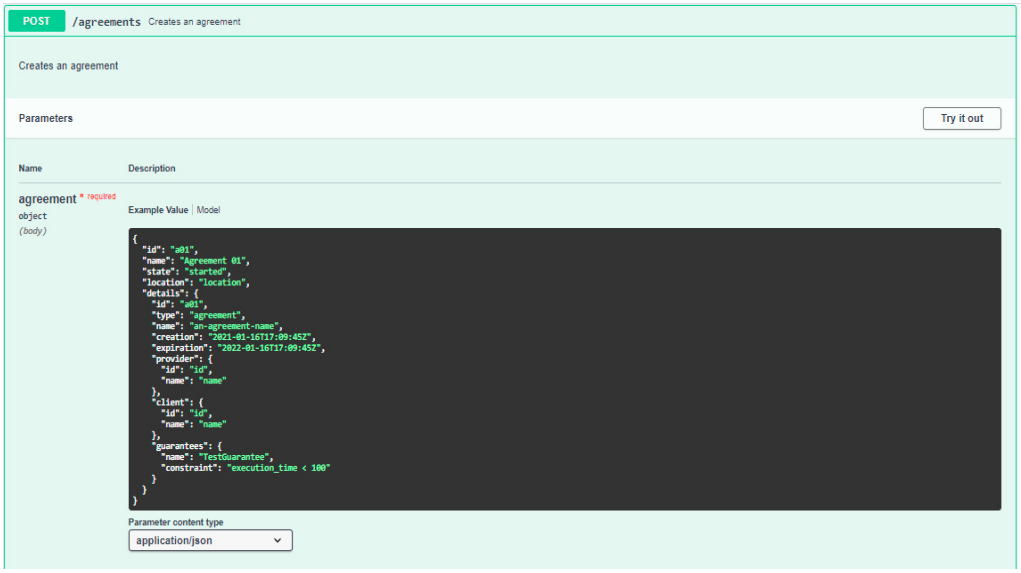



Figure 8: PLEDGER SLA tool swagger interface for REST API

**4. Check QoS threshold (metric) from Monitoring with SLA tool**

The SLA tool starts a monitoring loop to check the metric values every sleep time parameter. When it detects a violation of the metric (threshold value reached) it creates a violation alert with the details of the agreement violated.

**5. Send a violation alert from SLA to message middleware**

To notify to the rest of the system about a violation event we use a message broker system (Kafka) as a core component of Pledger. The SLA tool publish the violation message with the topic "sla_violation" and any component subscribed to this topic could receive the notification.

**6. Orchestrator read violation alert from middleware and execute an action (*scale out*)**

For this first iteration of the Pledger tools the orchestrator will be subscribed to this alert and it will implement an adaptation action based on the rules defined by the consumer. For this demo scenario it will be a scale out of the number of instances of the app until the maxReplicas parameter (max instance per app in the cluster).

Video recording link in the PLEDGER channel in YouTube:

https://www.youtube.com/channel/UCXV6V9rJ0ZvWhXeoWvDsArQ – Test Scenario 1 Demo WP3

### 5.3.2 Scenario 2: PLACEMENT

**1. Add cluster and edge infra configuration**

This first step is like the first step of scenario 1 but in this case, we must define two infrastructure locations:

- EDGE: docker engine runtime
- CLOUD: Kubernetes cluster

The description file for the Edge is the following:

```
{
    "id": "mycluster2",
    "name": Remote edge docker",
    "description": "Pledger remote testbed",
    "type": "docker",
    "so": "windows",
    "user": "remote_user (windows account)",
    "password": “remote_password",
    "restAPIEndPoint": "http://192.168.1.24:2375",
  }
```

**2. Add an app deployment config with QoS params and actions**

The deployment config is like the scenario 1 but we implement a different action (placement) to move the app from the edge to the cloud (from mycluster2 to mycluster).

This the QoS part defined by the consumer in the app descriptor:

```
  "qos": [
        {
          "name": "scrape_duration_seconds",
          "constraint": "scrape_duration_seconds < 0.04",
          "importance": [
            {
              "Name": "Mild",
              "Constraint": " > 0.04"
            },
            {
              "Name": "Serious",
              "Constraint": " > 0.4"
            }],
          "actions": [{
               "type": "MIGRATE",
               "value": "mycluster",
               "unit": ""
          }]
        }
    ]
```

**3. Deploy the app with the Orchestrator in the target location**

This step is like the Scenario 1 but in this case the Orchestrator deploys the app in a remote site using Docker HTTP/S API.

4. **Check QoS threshold (metric) from Monitoring with SLA tool**

This step is like the Scenario 1 but we could check the metric values in the metric collector of the PLEDGER cluster or in the remote site. For this iteration we will use the metric collector of PLEDGER broker cluster (Prometheus based).

5. **Send a violation alert from SLA to message middleware**

This step is the same as Scenario 1.

6. **Orchestrator read violation alert from middleware and execute an action (*placement*)**

In this step the Orchestrator undeployed the app from the remote edge (docker engine) and redeploys the app in the cloud (Kubernetes cluster).

Video recording link in the PLEDGER channel in YouTube:

https://www.youtube.com/channel/UCXV6V9rJ0ZvWhXeoWvDsArQ – Test Scenario 2 Demo WP3

# 6 Conclusions and next steps

In this deliverable D3.2, the work performed in WP3 Task T3.2 during M6-M18 was documented and the goal "MS4 – First iteration of WP3 prototypes" was achieved. This is an excellent starting point for the next breakthrough at M33, "MS7 - Second iteration of WP3 prototypes".

On this first version of the tool, the Orchestrator subsystem addresses the majority of MVP requirements. In this iteration, the following System User Cases (SUC) were implemented:

- SUC.02: Deploy App components.
- SUC.03: View available IaaS resources.
- SUC.05: Select fitting IaaS resources.
- SUC.06: Get App & App instances status info.
- SUC.07: Control App & App instances.
- SUC.08: Get notified about unwanted events.
- SUC.09: Handle unwanted events.
- SUC.11 (partially): Choose/ perform recovery action(s), in this integration two recovery actions are performed based on the alerts, SaaS providers cannot choose a strategy yet.
- SUC.13: Start/ pause/ stop App instances.
- SUC.15 (partially): Choose to scale up/ down (components of) App, same us SUC.11.
- SUC.19: IaaS monitoring.

In this deliverable, the Orchestrator and App profiler PLEDGER subsystem was presented along with instructions for installation and management. Furthermore, the functionalities of the tools were demonstrated through specific scenarios.

As far as the evolvement of this Task (T3.2) the next steps are, to address all the remaining requirements:

- SUC.04: Get recommendations for IaaS resources
- SUC.10: View suggested recovery actions
- SUC.14: Choose to migrate (parts of) components of App
- SUC.21: Get IaaS evaluation
- SUC.22: Compare different IaaS providers

Furthermore, effort will be placed on integration of the Orchestrator also with Benchmarking subsystem in order to obtain important information on the infrastructure the Orchestrator operates on. Also integrating with DSS to acquire valuable information about migration, deployment and scaling of the applications, to achieve the desired QoS. Finally, App profiler API will be finalised to achieve integration with DSS and ease of use for the tool.

# 7 References


[1] PLEDGER. D2.3 – Pledger Overall Architecture v1.0. Voutyras, Orfefs. 2020

[2] Prometheus home page. https://prometheus.io, retrieved 2021-05-25

[3] Kubernetes home page. https://kubernetes.io, retrieved 2021-05-25

[4] Grafana home page. https://grafana.com, retrieved 2021-05-25

[5] Docker home page. https://www.docker.com, retrieved 2021-05-25

[6] Kafka home page. https://apache.kafka.org, retrieved 2021-05-25

[7] MongoDB home page. https://www.mongodb.com, retrieved 2021-05-25

[8] PLEDGER. D3.3 – QoS and SLA assessment and negotiation tools v1.0. Castillo, Antonio. 2021